# Rethinking User Interfaces for Feature Location

Fabian Beck*, Bogdan Dit†, Jaleo Velasco-Madden‡, Daniel Weiskopf* and Denys Poshyvanyk‡
*VISUS, University of Stuttgart, Germany
†Boise State University, Boise, ID, USA
‡The College of William and Mary, Williamsburg, VA, USA
Email: fabian.beck@visus.uni-stuttgart.de, bogdandit@boisestate.edu, jnvelascomadde@email.wm.edu,
weiskopf@visus.uni-stuttgart.de, denys@cs.wm.edu

***Abstract*—Locating features in large software systems is a fundamental maintenance task for developers when fixing bugs and extending software. We introduce *In Situ Impact Insight* (I3), a novel user interface to support feature location. In addition to a list of search results, I3 provides support for developers during browsing and inspecting the retrieved code entities. In situ visualizations augment results and source code with additional information relevant for further exploration. Developers are able to retrieve details on the textual similarity of a source code entity to the search query and to other entities, as well as the information on co-changed entities from a project's history. Execution traces recorded during program runs can be used as filters to further refine the search results. We implemented I3 as an Eclipse plug-in and tested it in a user study involving 18 students and professional developers that were asked to perform three feature location tasks chosen from the issue tracking system of jEdit. The results of our study suggest that I3's user interface is intuitive and unobtrusively supports developers with the required information *when* and *where* they need it.**

## I. Introduction

Feature location [1] and impact analysis [2] are two fundamental activities that need to be performed during a maintenance task (e.g., implementing a new feature, fixing a bug), and involve finding the source code entities that need to be changed for the maintenance task. These activities are difficult and time consuming, especially for large software systems. Therefore, developing techniques to support developers during maintenance tasks is paramount, and a lot of effort has been dedicated towards developing such techniques [3]. The typical output of such techniques is a list of code entities (e.g., methods) that should be considered by the developers. But for developers, a list of methods is not the end but rather the start of the feature location process. Currently, the only further tool support addresses the reformulation of queries [4], [5], [6], [7] and the refinement of the search space [8]. Developers are left without much help to determine which of the methods really need to be changed and why; most previous feature location approaches considered developer interaction as a 'black box'.

In this paper, we introduce *In Situ Impact Insight* (I3), a novel user interface to better support developers when performing feature location. The main contribution is that I3 explains the search result to the users, allows for filtering of the results with respect to different execution scenarios, and relates code entities to each other (Figure 1). I3 uses visualization and other advanced user interface elements to present search results. In situ visualization augments the source code and

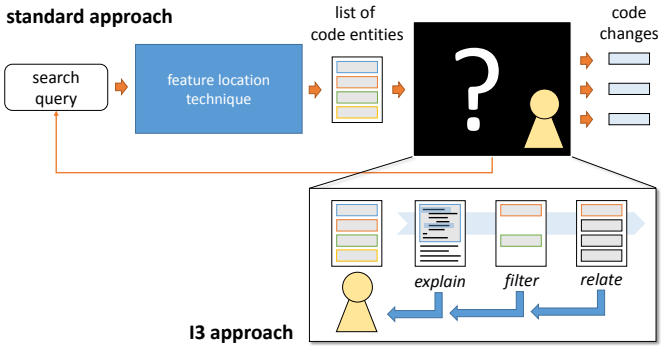



Fig. 1. Standard feature location process and our proposal of supporting the developer in understanding and navigating the search results.

the list of results with small diagrams. Detail views describe the textual similarities between a search query and a method, or between two methods. Also, developers can gain insights about change impact based on the change history of methods.

Our approach is based on state-of-the-art feature location integrating textual, historical, and dynamic information [9], [10] (Section III). We modeled the iterative feature location process as inter-connected cognitive tasks (Section IV). The user interface, implemented as an Eclipse plug-in, is designed to support these tasks (Section V). In a user study involving 18 participants, we tested the usefulness of I3 on three realistic feature location tasks for the popular text editor jEdit (Section VI). The feedback of the participants indicates that I3's interface is easy to use and unobtrusively supports developers with the required information from different data sources (Section VII). I3 drastically differs from previous feature location interfaces in the general approach of presenting the results to the developers (Section VIII).

## II. Related Work

The feature (or concept) location process is a fundamental first step towards addressing software maintenance tasks, and researchers have approached this problem from different directions. A systematic survey of feature location techniques is described by Dit et al. [3]. These automatic or semi-automatic techniques use textual [11], [12], [13], [14], [15], [16], [17], [18], dynamic [19], [20], historical information [21], [22], or hybrid approaches [23], [24], [12], [13], [4], [20], [25], [26], [27], [28], [29], [30], [31].

Researchers already addressed the user interfaces and visualization of the results produced by feature location tools. Sando [5], [7] and CONQUER [4], [6] help refine the query by suggesting co-occurring terms and synonyms. MFIE [8] is a faceted search approach that allows for focusing the search result with respect to a variety of facets. Our approach, I3, however, is most closely related to FLAT$^3$ [32] and its extension ImpactMiner [10]: execution traces are used to filter the search results and co-change information to find related code entities. The main difference of I3 to previous approaches is that the goal is not to come up with the perfect list of search results, but to consider the list as only the starting point of feature location. A detailed discussion, pointing out differences in the use of data sources, data representations, and evaluation, is provided as part of Section VIII.

Another avenue pursued consisted of conducting user studies to evaluate how developers perform feature location [33], [34], [35], [36], [37], [38], [39], [40], [41]. For instance, Ko et al. [35] conducted an exploratory study investigating how developers understand unfamiliar code during maintenance tasks, and reported that developers seek, relate, and collect relevant information when performing these tasks. Similar findings are reported by Wang et al. [41] who present a generalized model (consisting on phases, patterns, and actions) on how developers perform feature location. Li et al. [37] investigated help-seeking task structures and strategies, information sources, process models, and information needs of developers during software development tasks.

A sizeable body of work has been devoted to investigate the vocabulary of the source code [42], [43] in order to support developers during the process of formulating and reformulating queries. Haiduc et al. [44] used machine learning to propose a technique that recommends a reformulation strategy for a particular query based on the properties of the query. Stolee et al. [45], [46], [47], [48] proposed an approach that uses an SMT solver and symbolic analysis to identify the program fragments that adhere to query defined by developers as a set of inputs and expected outputs. Ge et al. [5] enhanced the Sando search tool [7] to recommend queries based on the dictionary of the software system, the term co-occurrence matrix, verb-direct-object pairs, a software engineering thesaurus, and a general English thesaurus. CodeExchange [49] aims at constructing a network of social and technical information relevant to the code. Although I3 was not designed to automatically suggest reformulations, it provides visual feedback to help with the task, for instance, by (i) highlighting the words from the query that were found in the code base and by (ii) showing the frequency of search terms and the terms of a particular method.

Our approach uses in situ software visualizations (i.e., small visualization embedded into the code and other elements of the IDE providing additional information where required) to preview textual search similarities and change history of methods. In general, small visualizations embedded in text are known as *Sparklines* [50]. While pretty printing and code highlighting are common visual aids in text editors, recently, in situ visualization were also used to augment source code, for instance, to encode static software metrics [51], dynamic software metrics [52], runtime consumption [53], or values of numeric variables [54]. However, to the best of our knowledge, in situ visualizations have never been used for feature location.

## III. Feature Location Technique

Our I3 approach implements a feature location technique that uses textual, dynamic, and historical information [9], [10]. Since I3 is integrated into Eclipse as a plug-in, the implementation is based on Java. This section briefly summarizes the technical realization of the feature location engine and data retrieval, while the following sections describe how the data is presented to the users.

**Information Retrieval:** We rely on Information Retrieval (IR) to provide a list of methods ranked based on their textual similarity to a search query [9], [55]. First, we take advantage of the Eclipse Java Development Tools (JDT) to extract all the methods from a software project. Second, using Lucene [56], we build a corpus at method-level granularity, preprocess it (i.e., remove special characters, split identifiers [57], remove English and Java stop words, and apply the Porter stemmer) and generate the indexed corpus. Third, given a search query, we use Lucene to compute textual similarities between the query and the methods of the software system. We finally retrieve a sorted list of search results.

**Dynamic Analysis:** In order to allow filtering of the results produced by the IR technique [9], [19], we provide developers with a mechanism to use the Eclipse Test and Performance Tools Platform (TPTP) to collect full traces of specific executions of software (i.e., they capture all the methods executed from the time the software starts until it is stopped by the developer). Once the list of methods from an execution trace is extracted, the developer can choose to rank these executed methods based on their similarity to the search query. Note that the use of dynamic information is optional.

**Mining Software Repositories:** The historical information about a software system, which is used as input for I3, is extracted in a similar way as described in our previous work [9], [58]. First, we retrieve all the commits and metadata from the version control system (in our case, SVN), including commit log message, author, list of modified files, and the actual changes. Second, for each commit, we identified the list of methods that had their content modified, using a tool based on the Eclipse Abstract Syntax Tree (from Eclipse JDT). Third, the list of methods modified for each revision is used as input to compute which methods frequently changed together (i.e., co-change relation).

## IV. Cognitive Task Analysis

Feature location is not the end in itself, but rather a part of a larger sense-making process that enables a developer to make informed decisions on code changes. To better understand the process, we specialize cognitive tasks taken from a more general model [59] for feature location. This *Cognitive Task Analysis* enables us to understand key activities behind feature location and provide good theoretical basis for designing a

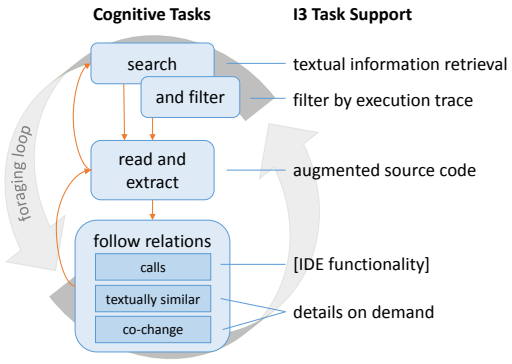



Fig. 2. Foraging loop of a *Cognitive Task Analysis* of a sensemaking process [59] applied to feature location.

user interface and visualizations supporting these activities, which can be considered as a form of support for distributed cognition [60]. In particular, we fit the *foraging loop* designed by Pirolli and Card [59] for intelligence analysis to feature location. It describes the process of seeking, understanding, and relating information. In its original version, it consists of the tasks *search and filter*, *read and extract*, *search for relations*, and *search for information*. In case of feature location, we identified the following inter-connected tasks (Figure 2):

**Search (and Filter):** The feature location process starts with a search, which is covered by the IR method in our approach. Optionally, filtering the results may restrict them to a smaller subset, here provided through execution traces.

**Read and Extract:** Following search and filter, initial information can be extracted from the search results by reading method names. But to really understand whether a method is relevant, developers need to look into the source code. This process could lead to reformulating the search query. The task of understanding the search results is facilitated in our approach by visually augmenting the results and the code.

**Follow Relations:** Search results are not independent of each other, but connected. While the IDE already allows following relations such as method calls, our approach adds other relations that can be relevant for feature location—textually similar or co-changed methods. Following relations generally leads back to task *read and extract*.

These tasks largely conform to those originally proposed [59]. Only *search for information* is only indirectly covered by the reformulation of a search query, which we only see as a variant of the formulation of the original search query being part of *search (and filter)*. Pirolli and Card [59] further describe a *sensemaking loop*, which follows after the *foraging loop*. Since it includes building a mental model or theory of the collected information, it, however, reaches beyond what our feature location approach may support. Applying this theory in our context is supported by Lawrance et al. [61] who found in a study that information foraging as formulated by Pirolli and Card [62] much better describes the strategies developers use for debugging, in particular, in the early stages of navigating through the code before starting to fix the bug.

## V. I3 Visual User Interface

The main contribution of this paper is proposing a novel user interface for feature location that handles and visualizes several sources of information relevant for the task. Results need to be represented, explained, and made explorable. As a consequence, our implementation of the proposed approach closely merges into an IDE, namely, Eclipse: in addition to a search view, the source code in the editor view becomes visually augmented with information that explains the search result and points to other possibly relevant code. Figure 3 gives an overview of the user interface. On the right side, a search view allows to begin the feature location activity by entering a search query (A). Results are presented in the list below (C) and can be filtered by execution traces (B). Search results are also indicated in the editor view by highlighting query terms. The method list and the editor view are enriched with in situ visualizations (D; Figure 4) that provide a preview of additional information related to textual similarity to the search query and change history of the methods. Moreover, details of these visualization techniques can be retrieved on demand in tooltip dialogs (Figure 5 and 6).

### A. Search Interface

As indicated in Figure 2, the starting point of each feature location process is a *search* of a textual description of the task, formulated as a query by the developer. Also, longer descriptions of the tasks can be used (e.g., description of a bug report, phrases from documentation). For this reason, the search field is a multi-line text field (Figure 3, A). During the feature location process—in particular, as a consequence of the task *read and extract* (Figure 2)—the search query might be corrected, specified in more detail, or alternated otherwise. After running the search, terms excluding stop words are printed in bold to give feedback to the users which terms have been really considered by the search engine.

The results, in the form of a list of methods, are presented below the search field. The methods are ordered by similarity ranking to the search query as provided by the Lucene search engine [56]. Supporting the *filter* task (Figure 2), they can be restricted to collected execution traces, available in a drop-down combo box (Figure 3, B): methods that were not executed in the respective recorded program run are removed from the list of results. For each method, the name, type of parameters, and return type are provided in the first line of each list entry. The second line further references the class and package containing the method. In front of the method and class names, Eclipse-specific icons are used to encode visibility properties and other modifiers. At the right side of an entry in the result list, a small visualization indicates that more details are available on demand (see Section V-B). By clicking on one of the search results, the corresponding method is opened and centered in the source code editor. All code belonging to methods that are part of the search results is highlighted with a blue background to discern it from other code. Terms in the source code that directly match one of the terms in the query are highlighted with darker blue. Further,

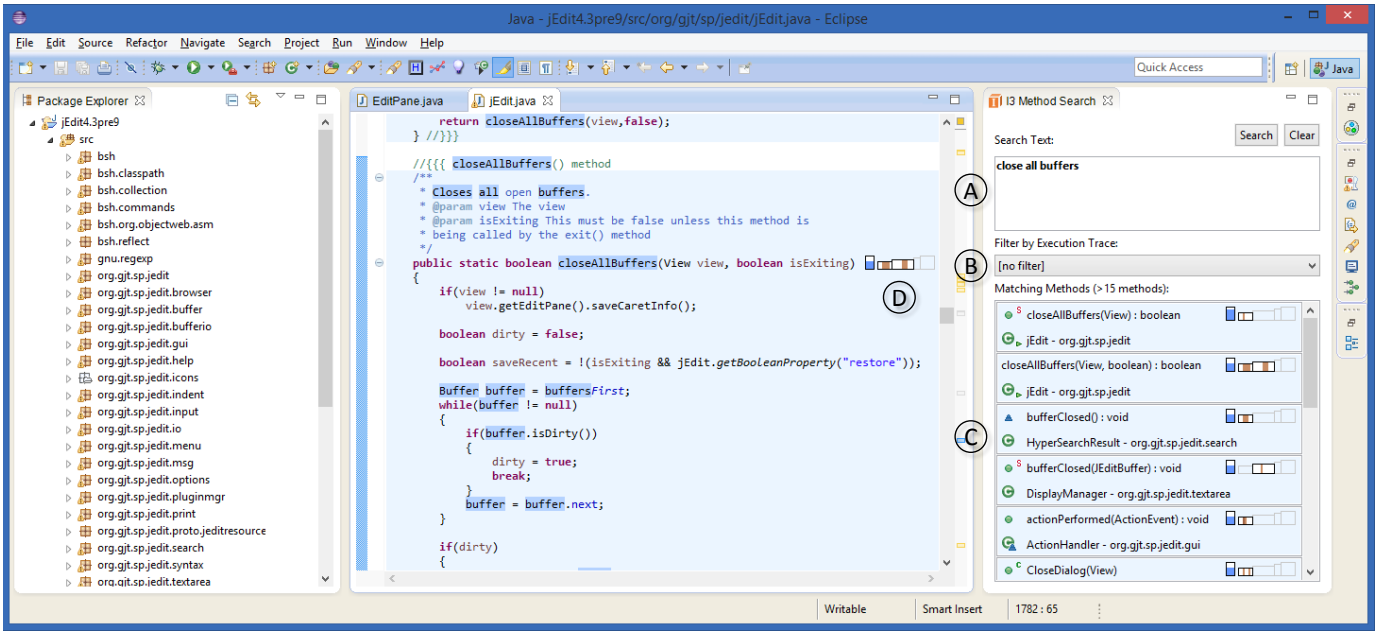



Fig. 3. User interface of I3 enriching Eclipse with an additional view for method search consisting of a search field (A), a filter selection (B), and a list of search results (C); search results are also highlighted and visualized in the editor (D).

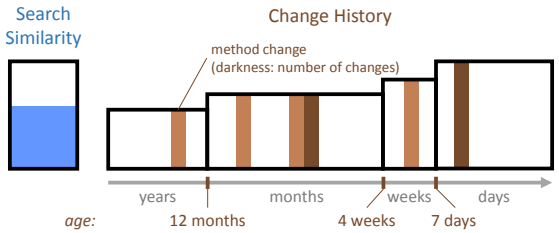



Fig. 4. Enlarged and annotated version of an in situ visualization of a method.

every method signature is augmented with an situ visualization equivalent to those used in the search results (Figure 3, D).

### B. In Situ Visualization

The goal of the in situ visualizations is to provide a preview of additional information that is available on demand for each method. Hence, they should not replace the detailed information but just allow the developer, when browsing the list of search results or source code, to quickly decide whether retrieving those details might be interesting. Being word-sized graphics, the in situ visualizations can be considered as types of Sparklines [50]. Each visualization is divided into two parts as shown in Figure 4:

**Search Similarity:** The blue bar on the left encodes the similarity of the method to the search query. Since Lucene rankings are not limited to a specific interval, we map the ranking value to the height of the bar based on a hyperbolic function, where a similarity of 0 produces an (invisible) bar of height 0, higher values asymptotically approach the maximum height indicated by the surrounding black border line.

**Change History:** The other part of the visualization encodes more complex information, namely, the full change history of the method as recorded. Since recent development is often more relevant than changes that occurred a long time ago, the timeline does not use a uniform resolution, but displays the younger history in greater detail. In particular, the timeline is split into four parts, the rightmost one showing the changes of the past 7 days (resolution: 1 day), followed to the left by the changes of the last 4 weeks (resolution: 1 week), the last 12 months (resolution: 1 month), and the last years (resolution: 1 year). The different areas of the timeline are framed by black borders (or gray border in case the method was not changed in the respective time interval) and drawn with decreasing height. Time steps are visualized as vertical lines color-coding the number of changes (the darker, the more changes; white, no changes).

While the similarity to the search result is displayed for every method, the change history part is only displayed when there is at least one commit stored for the method. Based on the two parts, two different tooltip dialogs provide details on demand when hovering the mouse over blue bar or the brown timeline. These dialogs contain lists of methods that support the *follow relations* task (Figure 2).

### C. Textual Similarity Details

Details for the blue bar give more background on the vocabulary used in the method as shown in Figure 5. On the left, the similarity value of the method to the search query can be retrieved next to the blue bar. Below, all keywords contained in the identifiers and comments of the method are listed with decreasing frequency. While keywords matching the search query are highlighted, the keyword frequency is encoded in the font size and subscript number. On the right, a list of methods is shown that are textually similar to the current

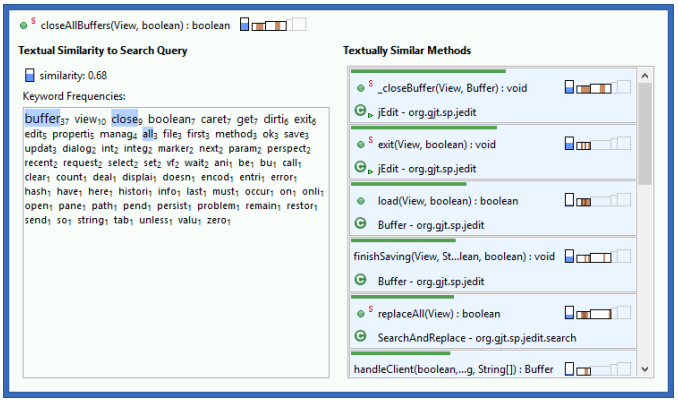



Fig. 5. Tooltip dialog showing textual similarity details of a method displayed when hovering the search similarity visualization of the method.

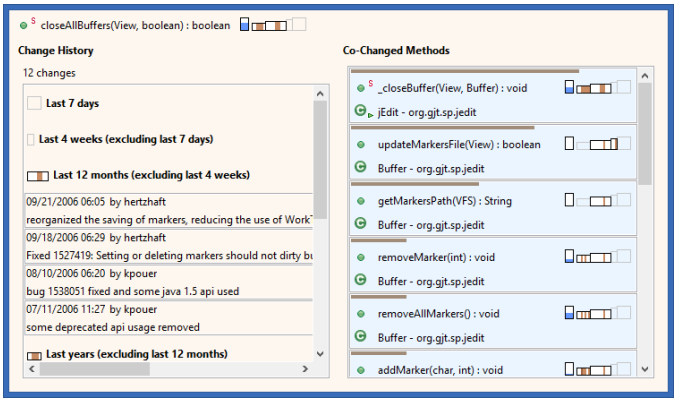



Fig. 6. Tooltip dialog showing change details of a method displayed when hovering the change history visualization of the method.

one: the keywords used in the method are taken as a query for starting a new search, for which the results are shown in the list. Here, the similarity to the specific search query is encoded in the green horizontal line at the top of the results—the blue bars in the search result still encode the similarity to the initial search, which is still active in background. Again, methods in the list can be clicked to jump to their source code representation.

### D. Co-change Details

In contrast, the tooltip dialog that appears when moving the mouse over the timeline visualization of a method provides details on past changes of the method (Figure 6). The list on the left further explains the data already displayed in the visualization: the changes grouped by periods starting with the most recent ones at the top. For each change, date, author, and commit message are displayed. The list on the right relates to other methods that were changed frequently together with the method in the past. The values encoded in the top brown bars and used for sorting refers to the so-called *confidence* of the co-changes [63]—the percentage of changes of the method where the other method changed as well. Also this list can be used for navigating to the source code of the related methods.

## VI. User Study Design

We performed a user study to test how I3 would be applied in practice by software engineers. The primary goal for the study design was to create a realistic software engineering scenario with non-trivial complexity. Hence, we used a real and sufficiently complex software system as a sample project and took real maintenance tasks from the issue tracking system as sample tasks for the study. Moreover, we did not reduce the evaluation to a single (quantitatively measurable) research question, but aimed at broader understanding of the feature location process as applicable through our approach. In particular, we designed the evaluation to answer the following research questions (RQs):

- **RQ1:** What information sources do developers use to identify the methods to change?
- **RQ2:** Which interaction strategies are applied for the feature location process?
- **RQ3:** Is the visual interface easy to understand and use? What are the obstacles?
- **RQ4:** How can I3 be applied in practice? What are the missing features?

The study consisted of (i) an introductory phase familiarizing the participants with I3, (ii) the main part where participants solved maintenance tasks, and (iii) a final questionnaire to collect feedback. During the main part, we collected the participants' answers and comments to the provided tasks and logged their interactions with I3. The questionnaire asked for quantifiable ratings and qualitative free text statements. The experiment was performed as a distributed user study with participants running the software on their own computers. We invited colleagues, industrial professionals, and current and former graduate students to participate. We designed the experiment to last 90 minutes. However, to reflect a realistic scenario where software engineers could freely schedule their tasks, we did not impose hard time constraints. The materials of the experiment, including the software, tutorial, questionnaire, and aggregated results are available online[1].

### A. Study Preface

After a short introduction, participants were asked to read a tutorial designed as a set of self-explaining slides. Participants were allowed to use it as a reference throughout the experiment. Then, participants started Eclipse as provided in a software and data bundle. The workspace already included the sample software project and I3 was initialized with a search index, execution traces, and historical data, so that participants did not need to set up anything. To familiarize participants with the tool and to check whether they understood its features, we asked them to conduct a number of simple tasks: to perform a search, to filter methods, to name the most frequent keywords of a method, to find textually similar methods, to retrieve certain changes of a method, and to identify co-changed methods. The participants had to fill in their answers for these tasks in a text field, which we used to check correctness.

[1]https://sites.google.com/site/i3featurelocation/

### B. Main Part

The main part of the user study consisted of performing feature location tasks very close to a realistic software engineering maintenance scenario. We decided to use fixing issues, such as bugs or simple feature improvements, reported in the issue tracking system of the sample project: First, the issue description provides a well-defined starting point for feature location. Second, it is a realistic scenario that the developer fixing the issue is not familiar with the code and, in particular, is not the person who initially wrote the code. Third, for already closed issues, there exists a reference solution of which methods were needed to be changed to fix the issue.

As a sample project, we selected jEdit, a text editor written in Java. It is a reasonably large software system, but not too complex for participants to understand the source code. It has been already used in related studies [33], [64], [36], [38], [32], [40]. Code, change information, bug reports, and execution traces are available as a benchmark dataset [58]. From this benchmark, we used version 4.3 pre9 of jEdit with a commit history from 2000-01-15 to 2007-01-20 (SVN revision 1–8676; we removed duplicate revisions included in the dataset). To simulate that participants continue development of jEdit 4.3 pre9, we artificially set the date to 2007-01-20 in I3 because the timeline visualizations depict the recent history in higher resolution.

We selected three issues from the benchmarks as sample tasks for the participants of the study to solve. Criteria for the selection were that the issues cover different features of jEdit, that the issue reports provide a comprehensive descriptions of the requested fixes, that the fixes were non-trivial (at least five methods changed in the reference solution), that the fixes did not require to add considerable amount of new code, and that the issues had been fixed after 2007-01-20, but the related code has been already available in version 4.3 pre9. The following issues were finally retrieved as a set of tasks for the study:

- **Task 1 (Issue 1671312):** The issue describes that regular expressions used in the search dialog do not find matches of zero length.
- **Task 2 (Issue 1999448):** It is reported that joining lines is not working efficiently if a folding mode is used for collapsing parts of the text.
- **Task 3 (Issue 1730845):** The issue is requesting the feature of selecting an entire line or multiple lines through clicking in the gutter (i.e., the area at the left side of the editor usually containing line numbers).

The specific question we asked was "Which methods would you think need to be changed for fixing the above issue?" and participants provided their answers as a list of methods. As additional questions, we further wanted to know, which specific sources of information were relevant for solving the task and which Eclipse functionalities were used as relevant additional help. To warrant that participants had understood the procedure, we started with a simpler, yet also realistic warm-up task that follows the same question design (results from this questions are not evaluated in the following).

### C. Questionnaire

The study closed with a questionnaire for general feedback. First, we asked the participants to formulate what they did and did not like about I3 as free text and wanted to know if they could imagine an approach like this in their daily development work. Then, we explored the different features of the approach based on—among other questions—rating scales, one for the intuitiveness of the features, one for the usefulness of the features. We also wanted to learn whether the participants considered the tasks as realistic software development scenarios. Finally, we collected some personal background of the participants, in particular, their professional occupation and experience in software development.

### D. Participants

Twenty participants took part in and completed the user study. Based on the results from training tasks, for which we know the correct answers, we excluded 2 participants because they only answered less than 5 out of the 7 questions correctly. In the following, we only analyze the answers of the remaining 18 participants: The largest group of participants are PhD students (8), followed by professional software developers (3), Master students (3), Bachelor students (2), professors (1), and researchers (1). They judged their software development expertise as high: average of 3.9 on a scale from 1 (very low) to 5 (very high). Also, their Java experience was not much lower (average of 3.6 on the same scale).

## VII. Results

For analyzing the results of the study and answering the research questions, we applied mixed methods: descriptive statistics for evaluating quantifiable measures and qualitative methods summarizing free text answers. All averages reported in the following are mean values aggregated from rating scales. The scales were all only labeled for the two extreme values. Hence, they can be interpreted, at least in approximation, as interval scales. Additional histograms are provided for the most important rating scales to further clarify.

The participants agreed that the issue fixing tasks are realistic: on average 4.4 on a scale from 1 (I do not agree) to 5 (I agree). They attested the tasks to be of medium difficulty: 2.7 on a scale from 1 (too difficult) to 5 (too easy). Since there are different ways to implement a fix, it is difficult to judge whether the answers provided as a list of methods are correct. Using the solution by the original developers, we at least consider the methods being part of this reference as correct, but cannot make any statements about the correctness of other methods (the agreement values provided in the following can be interpreted as estimated lower bounds of precision values). For Task 1, participants answered 3.2 methods on average of which 43% also appeared in the reference; for Task 2, 3.1 methods and 35% agreement; for Task 3, 3.3 methods and 45% agreement. Hence, the tasks seem to have a comparable level of difficulty. Abstracting from methods to classes shows that the answers not matching the reference are often in the context of the reference answers: a higher agreement is reached if we

TABLE I

RELEVANCE RATINGS OF DATA SOURCES ON A SCALE FROM 1 (TOTALLY IRRELEVANT) TO 5 (HIGHLY RELEVANT) DISCERNED PER TASKS AS WELL AS AVERAGED; HISTOGRAMS SHOW THE DISTRIBUTION OF THE ANSWERS, PERCENTAGES QUANTIFY RATINGS OF AT LEAST 4 (RELEVANT).

| | list of methods | execution trace | source code | list of keywords | text. similar methods | changes | co-changed methods |
|---|---|---|---|---|---|---|---|
| *Task 1* | 4.0 | 3.3 | 4.1 | 2.4 | 3.3 | 2.7 | 3.2 |
| *Task 2* | 4.6 | 3.1 | 3.9 | 2.7 | 2.9 | 1.9 | 2.9 |
| *Task 3* | 4.1 | 3.1 | 4.2 | 2.6 | 3.2 | 2.1 | 2.8 |
| ***avg*** | **4.2** | **3.2** | **4.1** | **2.5** | **3.1** | **2.2** | **2.9** |
| *histogram* | 1 2 3 4 5 | | | | | | |
| *relevant (≥4)* | 81% | 44% | 74% | 26% | 44% | 17% | 39% |

check whether participants found the same classes as changed in the reference (Task 1: 83%; Task 2: 66%; Task 3: 50%). However, we did not analyze the 'correctness' of the answers further because there is not only one correct solution, but multiple ones that are subjective, context-dependent, and do not necessarily have considerable agreement [65], [66].

### *RQ1: Information Sources*

Investigating the importance of different sources of information used in I3, we analyze the ratings of relevance that participants gave. Table I summarizes these relevance ratings per task and data source across all participants applying an average measure. The results show that the list of methods as retrieved through textual search and the source code form the most relevant source of information (average relevance of 4.2 and 4.1, being considered as relevant in 81% and 74% of the cases); this matches the design decision to put textual search and code highlighting of search results in the main focus of I3. Less important, but still rated as relevant in about a third to half of the cases were execution traces (44%), textually similar methods (44%), and co-changed methods (39%); again this matches the role of the information in the user interface because this information is only presented through additional selection. Although not totally irrelevant, the list of keywords and the change history of a method are rated with lower relevance (average relevance of 2.5 and 2.2, rated as relevant in 26% and 17% of the cases). The histograms reveal different kinds of distributions for the answers: while the list of methods and the source code are rated with a high agreement among the participants, the other data sources show a broader distribution. This could indicate that particularly those data sources are more difficult to interpret and helped only some of the participants. Comparing the tasks, we mostly observe consistent results, only with a few exceptions: the list of methods in Task 2 and the list of changes in Task 1 seem to be considerably more relevant than in the other tasks.

> ***Result 1:*** The relevance ratings of the data sources conform to their role in the user interface of I3. The answers further confirm that not only IR seems to reveal relevant results, but filtering by execution traces, textually similar methods, and co-changed methods are rated as important as well.

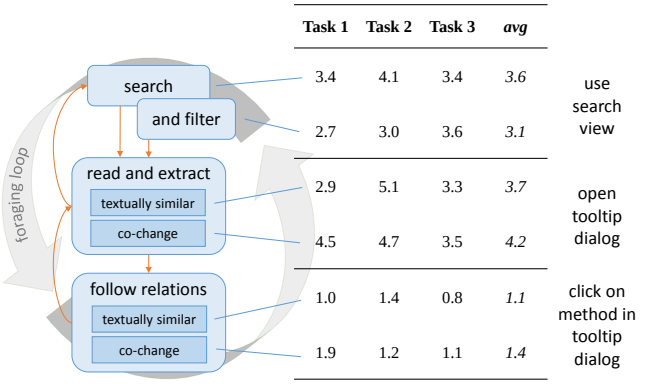



Fig. 7. Model from *Cognitive Task Analysis* (Figure 2) adapted for analysis and augmented with average usage frequencies for the three tasks of the study.

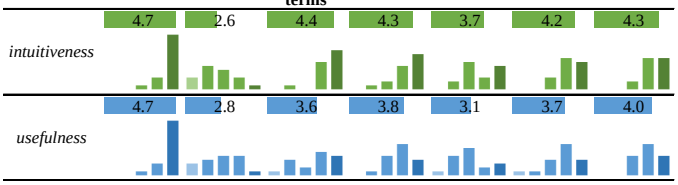
### *RQ2: Interaction Strategies*

To analyze the applied usage strategies, we map the specific usage interactions recorded in log files to the high-level cognitive tasks as derived from *Cognitive Task Analysis* (Section IV). Since there are multitudes of ways in Eclipse to navigate through the code, we only focus on the means of interaction provided by I3. As a consequence, we slightly adapt Figure 2 by removing the parts reflecting IDE functionality and specifying the *read and extract* task in more detail. Figure 7 shows the adapted version enriched with usage frequencies of the related functionality. Please note that *'textually similar'* and *'co-change'* refer to opening tooltip dialogs of the respective type for *'read and extract'* while the same terms are mapped to clicking on a related method in these tooltip dialogs for *'follow relations'*.

The average usage frequencies reveal that a search query was (re-)formulated 3.6 times per task and filters were set 3.1 times (filters needed to be set again when changing the search query). This indicates that feature location is applied as an iterative process. A manual analysis of the of applied filters shows that the relative high frequencies for filtering are only partly an effect of consistent usage of this functionality: some participants gave up on using the feature at all while others tested different filters one after the other (also those for other tasks than the current one). Tooltip dialogs are opened on average about 4 times per task for each type of dialog (3.7 times for textual similarity details and 4.2 times for co-change

TABLE II
INTUITIVENESS AND USEFULNESS RATINGS OF USER INTERFACE ELEMENTS ON A SCALE FROM 1 (NOT INTUITIVE/USEFUL AT ALL) TO 5 (VERY INTUITIVE/USEFUL) AS AVERAGES AND HISTOGRAMS.

| | search result | filtering | high-lighting of search terms | blue bar | timeline vis. | textual similarity details | history details |
|---|---|---|---|---|---|---|---|
| *intuitiveness* | 4.7 | 2.6 | 4.4 | 4.3 | 3.7 | 4.2 | 4.3 |
| *usefulness* | 4.7 | 2.8 | 3.6 | 3.8 | 3.1 | 3.7 | 4.0 |



details). When a tooltip dialog is opened, a relation is followed in the form of clicking on a related method in about third of the cases (1.1 of 3.7 times for textual similarity details and 1.4 of 4.2 times for co-change details). We, however, can only speculate what participants did in the other two thirds of the cases—reading the information on the left, studying the list of methods on the right, or they might have opened the dialog just by accident. The usage frequencies are in general quite consistent with respect to the three tasks, only that Task 2 tends to require more user interaction with I3, which could point to a higher degree of difficulty.

For each task, we asked for an extra free text answer to retrieve built-in functionality of Eclipse that participants considered as relevant to solve the task. In 69% of the cases, extra features were named, which we manually classified. The usage of Eclipse functionality was quite diverse: no item occurred in more than third of the cases. Most regularly, participants used local search, opening a declaration of a code entity, and the call hierarchy (answered in 30%, 22%, and 20% of the cases). Other functionality is mentioned only occasionally, such as the outline view (11%), the package explorer (7%), the type hierarchy (4%), finding references (2%), and JavaDoc (2%). Interestingly, no participant named the global search functionality of Eclipse. The answered functionalities, with the exception of JavaDoc, serve for code navigation and can be classified into *global* and *local* navigation (global: package explorer, global search; local: call hierarchy, find references, local search, open declaration, outline, type hierarchy). If participants named additional Eclipse functionality (69%), these almost always included local navigation (65%), but only rarely global navigation (7%).

> ***Result 2:*** Developers use I3 iteratively to locate features as intended, where the first part of the foraging loop is repeated more often than the second part. In the context of issue fixing tasks, the new means of code navigation largely replace the Eclipse functionality for global code navigation and complements local code navigation.

*RQ3: Visual Interface Analysis*

To investigate specific elements of the I3's user interface, we asked participants to rate the intuitiveness and usefulness on a numeric scale (Table II). With respect to intuitiveness, we intended to find whether interactions matched the users' expectations, visualizations were easy to understand, and I3 in general had a flat learning curve. Most features reach good scores $\geq 4$ on a scale from 1 (not intuitive at all) to 5 (very intuitive). Only filtering by execution traces and the timeline visualizations receive lower ratings (2.6 and 3.7); the histograms show a broad spectrum of ratings for these two categories. Analyzing the free text answers describing obstacles of understanding, we find a simple explanation for the low rating of filtering: some participants just did not understand where the execution traces came from (these were recorded before the experiment and we may have not explained this clearly enough in the preface of the user study). In contrast, for the timeline visualization, participants did not state to have problems to understand them—their complexity might just required additional explanations as provided in the tutorial. As the gapped histogram indicates, participants are divided in their rating of intuitiveness.

The rated usefulness of the user interface elements, as also provided in Table II, connects the elements to the feature locations tasks, however, cannot be independent of the data sources discussed in RQ1. As expected, the usefulness of the elements largely agrees with the relevance of the underlying data sources; for instance, the displayed search result containing the list of methods is rated as most useful (4.7). Further, the history details (4.0), the blue bar in the in situ visualization (3.8), textual similarity details (3.7), and the highlighting of search terms in the code (3.6) tend to be useful. Again, the filtering by execution traces and the timeline visualization form an exception with a lower usefulness rating around 3, but with a broad spectrum of answers. For the in situ visualizations, we further explicitly asked whether participants liked them: a manual sentiment analysis of the free text answers revealed that 11 participants were positive about the visualizations, but only one participant was negative (the other participants were indifferent or did not provide any interpretable answer). While participants seem to agree that the visualizations are non-obtrusive and provide a good anchor for retrieving details on demand, in particular for the timeline visualization, they do not agree whether the visualization as such is useful or not (both statements can be found in the answers).

> ***Result 3:*** The general intuitiveness of I3 is rated high, only filtering by execution traces and timeline visualization seem to need some explanation. The presentation of the search results and the details on demand provided as tooltip dialogs are considered as useful.

*RQ4: Practical Applicability*

Generally, we convinced participants that I3 is useful in practical application: with an average of 4.2 on a scale from 1 (definitely not) to 5 (definitely) participants would use an approach like ours in their daily work. Summarizing some of the high-level textual feedback, one participant said that already "*the IR-based search tool is much better than the state-*

*of-the-art practice*" and "*I don't have to go to another view to get the information I want.*" Another participant liked that "*everything is an single view*" and somebody remarked that the tool "*is well integrated into Eclipse*". Many participants also explicitly highlight the way of retrieving relations between methods, for instance, that it is "*very nice to see corresponding methods instantly*" or "*I liked the wide array of ways the tool allows you to draw relationships between methods.*" Negative comments often referred to lacking intuitiveness of filtering by execution traces: one participant further explained that "*it is a lot to ask users to execute their program before searching*." Suggested features to further improve the approach were to add way to filter by a specific package, to integrate the possibility to exclude keywords, to cluster corresponding methods, to visualize method calls as a tree or graph, to select specific time frames for co-changes, to always show the timeline visualization (not only when searching), and to manually mark or hide irrelevant search results. Participants see the main area of application of I3 (among the suggested answers) in adapting existing features (17 of 18 participants), in fixing bugs (14), finding code for reuse (13), implementing a new feature (10), and restructuring/refactoring the program (7). Three participants additionally recommend to use I3 in general to understand a new codebase or project.

> ***Result 4:*** The answers of the participants suggest that our approach can be leveraged in practice, mainly, to adapt existing features, to fix bugs, and to find code for reuse.

## VIII. Discussion

To reflect the result of our work in a broader context, we finally discuss limitations of the user study and compare I3 systematically to related feature location approaches.

### A. Limitations of the User Study

The user study was designed to cover a realistic scenario and participants confirmed the realism of the tasks. Nevertheless, certain factors limit the validity of our study. First, we were only able to test three tasks on a single open source software project; for instance, the relevance of data sources, quality of the search results, or the applied usage strategies could be different for other scenarios. Second, participants were not familiar with the code nor the project; although this is a realistic scenario, it might not be the standard case. The preloaded set up of the data introduced an artificial element and led to some obstacles (e.g., some participants did not understand how the execution traces were generated). Third, participants were mostly graduate students; more experienced, professional software developers could have different requirements for a feature location tool. Finally, the number of participants is still too small to test numerical differences for statistical significance—the insights of the user study have to be generally considered as preliminary before more authoritative evidence is provided.

Moreover, the user study did not compare I3 to other approaches like the standard search of Eclipse or competing advanced feature location tools. Hence, we cannot claim that our technique is better than others, but only that it can be leveraged for feature location, that it received positive feedback, and that our tool largely replaced the build-in global navigation and search features of the IDE during the experiment. While, initially, we planned to conduct a comparison of tools in a comparative controlled experiment, we finally discarded the plan for several reasons. First, a fair comparison is hard to design due to many differences in implementation of the tools. Second, the results would be difficult to interpret as the approaches do not differ in one, but a multitude of variables and characteristics. Third, for valid statistical analyses, the study would have required a larger number of participants and a more streamlined experiment setup (i.e., shorter and less realistic). Instead, we performed a non-comparative study that allowed us to inspect the individual characteristics of our approach in more detail. We believe, such an experiment—at least as a first evaluation—is likely to provide deeper insights than a comparative study of same scale.

### B. Qualitative Comparison of Tools

Nevertheless, differences of our approach to others can be qualitatively discussed in greater depth. Additional to the review of related work in Section II, we compare the approach to those state-of-the-art feature location techniques that include an advanced user interface. The following discussion is based on the dimensions of software visualization as introduced in the taxonomy by Maletic et al. [67]: *task*, *audience*, *target*, *representation*, and *medium*. While *task* (feature location), *audience* (software developers), and medium (standard color screen) are the same for all tools, the remaining two dimensions can be used to systematically discern the visual user interfaces of the tools. Additionally, we also discuss the *evaluation* of the approaches. Table III gives an overview of these dimensions and lists for each of the tools some meta-data and characteristics with respect to the dimensions. The subentries for the dimensions were retrieved specialized to feature location as a superset of the characteristics of all tools.

*1) Information Sources (Targets):* With respect to information sources—or *targets* as called by Maletic et al. [67]—I3 incorporates co-change information and execution traces additional to the standard data source of code and comments (analogous to ImpactMiner). Sando, in contrast, focuses on the vocabulary used in the code and comments; thesauri imported from external source and build by finding co-occurrences of terms are used to help with query (re-)formulation; similarly, CONQUER uses co-occurrences. By taking the parsed code structure into account, MFIE allows to filter parts of the system based on containment and usage. Hence, the tools focus on rather different sources, each of them providing potentially valuable extra information. Comparing the impact of each is still an open research question.

*2) Representation of Information:* Available in all tools, the standard means to represent the retrieved information is a list of code entities. Highlighting of terms is another standard feature, either done in the editor itself (CONQUER,

TABLE III
QUALITATIVE COMPARISON OF INTERACTIVE FEATURE LOCATION TOOLS AND THEIR EVALUATION.

| | | | target | | | | | representation | | | | | | | | evaluation | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tool | year | environment | code and comments | code structure | thesaurus | co-changes | execution traces | list of entities | highlighting of terms | term summaries | query suggestions | query history | related entities | change history | in situ visualization | field study | experiment | # participants | duration (min) | # issues | # projects | baseline comparison | analyze sources | analyze representat. | analyze interaction |
| FLAT$^3$ [32], ImpactMiner [10] | 2010/14 | Eclipse | × | | | × | × | × | × | | | | × | | | | | | | | | | | | |
| Sando [7], [5], [34] | 2012/14 | Visual Studio | × | | × | | | × | × | × | × | × | | | | × | | 274 | | | | | | × | × |
| CONQUER [6], [4] | 2013/14 | Eclipse | × | | × | | | × | × | × | × | | | | | | × | 18 | 45 | 8/28 | 5 | × | | × | |
| MFIE [8] | 2013 | Web | × | × | | | | × | × | × | | × | × | | | | × | 20 | 120 | 4 | 1 | × | | | × |
| I3 | 2015 | Eclipse | × | | | × | × | × | × | × | | | × | × | × | | × | 18 | 90 | 3 | 1 | | × | × | × |

I3), in code snippets giving a preview of the code (Sando, MIEF), or in abstracted visual thumbnail representations of the code (FLAT$^3$/ImpactMiner). Term summaries such as lists, trees, or cloud representations are used for representing co-occurring terms (Sando), organizing the findings in categories (CONQUER), or summarizing the vocabulary of a code entity (MFIE, I3). For formulating or rewording a query, a list of suggested changes or extensions of the query can be useful (Sando, CONQUER). A query history helps keeping an overview and jump back to previous queries (Sando, MIEF). Dependencies or similarities between entities enable finding related entities or understand interaction between the entities, based on structural dependencies (MFIE), textual similarity (I3), or co-changes (ImpactMiner, I3). Finally, I3 is the only tool that leverages in situ visualizations augmenting the code and list of search results and shows a commit history of the code entities. Although I3 is among the most versatile tools regarding data representation, only few additional views are required: the representations are tightly integrated in the existing user interface of the IDE. Adding further representations like query suggestions and history will not be difficult.

*3) Tool Evaluation:* While the user interfaces of FLAT$^3$/ImpactMiner were only evaluated in small case studies, the other tools were tested in user studies. Sando was evaluated in a field study analyzing the log files of 274 participants with respect to representations and interactions (in another study Sando was used, but its user interface was not evaluated [66]). CONQUER, MFIE, and I3 were tested in user experiments under more controlled conditions. The scales of these studies are similar with respect to participants (18–20). The studies of MFIE and I3 let the users investigate in-depth (ca. 90–120 min) a small number of feature location tasks (3–4) for a single project; the study of CONQUER covers 8 tasks from a larger selection of 28 tasks from 5 projects in shorter time (ca. 45 min). CONQUER (compared to simplified version and Eclipse; no statistical significance difference) and MFIE (compared to Eclipse; statistical significant differences) are evaluated in comparison to baseline approaches. Our study evaluated only I3, but focuses on a detailed qualitative and quantitative analysis of the data sources, data representations, and interactions. The comparison CONQUER and MFIE to standard search already suggested that advanced feature location approaches could improve the standard. A comparative user study among the approaches discussed in this section, however, is still lacking.

## IX. CONCLUSION AND FUTURE WORK

We presented a novel user interface for feature location and evaluated it in a user study. Feature location was modeled as a cyclic sense-making process. Our approach specifically supports the cognitive tasks of this process: while a standard IR-based approach is used for search, execution traces allow for filtering the results with respect to a specific program run. Supporting reading the results and extracting information, search results are highlighted in the code editor and augmented with in situ visualizations. These visualizations also provide a starting point for following relations to textually similar or co-changed methods. The user study shows that the IR-based approach is the central source of information for I3, and that extra information available on demand is relevant for retrieving specific features in the code. The presented user interface is largely intuitive and the majority of functionalities were clearly useful for solving the provided feature locations tasks.

In comparison to other tools, I3 has a specific focus on the exploration of co-change patterns to evaluate the impact of changes. Its use of in situ visualization for explaining and relating search results is a unique characteristic among those tools. Nevertheless, future work should also include integrating additional representations of other feature location tools such as query suggestions and filtering by other criteria. This would finally allow to evaluate the approaches systematically against each other in controlled experiments. Another avenue of future research is to explore whether parts of the introduced user interface are useful in other development scenarios.

## ACKNOWLEDGMENT

Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers. The authors from W&M are supported in part by the NSF CCF-1218129 and NSF-1253837 grants.

## REFERENCES


[1] T. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the 15th IEEE/ACM International Conference on Software Engineering (ICSE'93)*, 1993, pp. 482–498.

[2] S. Bohner and R. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[4] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet, "NL-based query refinement and contextualized code search results: A user study," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering 2014 Software Evolution Week (CSMR-WCRE'14)*, 2014, pp. 34–43.

[5] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill, "How the Sando search tool recommends queries," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering 2014 Software Evolution Week (CSMR-WCRE'14)*, 2014, pp. 425–428.

[6] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "CONQUER: A tool for nl-based query refinement and contextualizing code search results," in *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13)*. IEEE, 2013, pp. 512–515.

[7] 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 (FSE'12)*. ACM, 2012, pp. 15:1–15:2.

[8] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*, 2013, pp. 762–771.

[9] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, 2012, pp. 430–440.

[10] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vsquez, D. Poshyvanyk, and H. Kagdi, "ImpactMiner: A tool for change impact analysis," in *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14), Formal tool demonstration track*, 2014, pp. 540–543.

[11] S. L. Abebe, A. Alicante, A. Corazza, and P. Tonella, "Supporting concept location through identifier parsing and ontology extraction," *Journal of Systems and Software*, vol. 86, no. 11, pp. 2919–2938, 2013.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, 2009, pp. 232–242.

[13] ——, "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 (ASE'11)*, 2011, pp. 524–527.

[14] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.

[15] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*, 2011, pp. 92–96.

[16] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information-retrieval-based bug localization based on bug reports," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[17] D. Poshyvanyk and D. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007, pp. 37–48.

[18] M. Revelle and D. Poshyvanyk, "An exploratory study on assessing feature location techniques," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC'09)*, May 2009, pp. 218–222.

[19] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007, pp. 234–243.

[20] D. Poshyvanyk, Y. Guhneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[21] A. Chen, E. Chou, J. Wong, A. Yao, Q. Zhang, S. Zhang, and A. Michail, "CVSSearch: searching through source code using CVS comments," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, 2001, pp. 364–373.

[22] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Learning from project history: a case study for software development," in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW'04)*. ACM, 2004, pp. 82–91.

[23] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10)*, 2010, pp. 156–159.

[24] 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 (ASE'07)*, 2007, pp. 14–23.

[25] S. Reiss, "Semantics-based code search," in *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*, 2009, pp. 243–253.

[26] M. P. Robillard and G. C. Murphy, "FEAT: a tool for locating, describing, and analyzing concerns in source code," in *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, 2003, pp. 822–823.

[27] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the 6th International Conference on Aspect Oriented Software Development (AOSD'07)*, 2007, pp. 212–224.

[28] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*, 2011, pp. 119–123.

[29] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *Proceedings of the 14th International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 137–148.

[30] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *Proceedings of the 18th International Conference on Program Comprehension (ICPC'10)*, 2010, pp. 14–23.

[31] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," vol. 18, no. 2. Springer US, 2013, pp. 277–309.

[32] 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 – Volume 2 (ICSE'10)*, 2010, pp. 255–258.

[33] B. de Alwis, G. C. Murphy, and M. P. Robillard, "A comparative study of three program exploration tools," in *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007, pp. 103–112.

[34] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill, "How developers use multi-recommendation system in local code search," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*, 2014, p. (to appear).

[35] 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*, vol. 32, no. 12, pp. 971–987, 2006.

[36] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the the 6th Joint Meting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE'07)*, 2007, pp. 361–370.

[37] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*, 2013, pp. 142–151.

[38] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[39] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, 2007.

[40] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "TopicXP: Exploring topics in source code using Latent Dirichlet Allocation," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–6.

[41] J. Wang, X. Peng, Z. Xing, and W. Zhao, "How developers perform feature location tasks: a human-centric and process-oriented exploratory study," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[42] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, 2011, pp. 125–134.

[43] S. L. Abebe and P. Tonella, "Towards the extraction of domain concepts from the identifiers," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE'11)*, 2011, pp. 77–86.

[44] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*, 2013, pp. 842–851.

[45] K. T. Stolee, "Finding suitable programs: Semantic search with incomplete and lightweight specifications," in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*. IEEE Press, 2012, pp. 1571–1574.

[46] K. T. Stolee and S. Elbaum, "Toward semantic search via SMT solver," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM, 2012, pp. 1–4.

[47] ——, "On the use of input/output queries for code search," in *Proceedings of the 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, 2013, pp. 251–254.

[48] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 3, pp. 26:1–26:45, 2014.

[49] L. Martie and A. Van der Hoek, "Toward social-technical code search," in *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'13)*, 2013, pp. 101–104.

[50] E. R. Tufte, *Beautiful evidence*, 1st ed. Graphics Press, 2006.

[51] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation," in *Proceedings of the 21st Australian Software Engineering Conference (ASWEC'10)*, 2010, pp. 171–180.

[52] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret, "Augmenting static source views in ides with dynamic metrics," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 253–262.

[53] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC'13)*, 2013, pp. 63–72.

[54] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, "Visual monitoring of numeric variables embedded in source code," *Proceedings of the 1st IEEE Working Conference on Software Visualization (VISSOFT'13)*, pp. 1–4, 2013.

[55] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214–223.

[56] (2008) The Apache Software Foundation - Lucene. [Online]. Available: http://lucene.apache.org

[57] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proceedings of the 19th International Conference on Program Comprehension (ICPC'11)*, 2011, pp. 11–20.

[58] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "A dataset from change history to support evaluation of software maintenance tasks," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13), Data Track*, 2013, pp. 131–134.

[59] P. Pirolli and S. Card, "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis," in *Proceedings of International Conference on Intelligence Analysis*, vol. 5, 2005, pp. 2–4.

[60] Z. Liu, N. J. Nersessian, and J. T. Stasko, "Distributed cognition as a theoretical framework for information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1173–1180, 2008.

[61] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[62] P. Pirolli and S. Card, "Information foraging," *Psychological Review*, vol. 106, no. 4, pp. 643–675, 1999.

[63] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[64] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 351–360.

[65] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock, "An empirical study of the concept assignment problem," School of Computer Science, McGill University, Tech. Rep., 2007.

[66] K. Damevski, D. Shepherd, and L. Pollock, "A case study of paired interleaving for evaluating code search techniques," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering 2014 Software Evolution Week (CSMR-WCRE'14)*, 2014, pp. 54–63.

[67] J. I. Maletic, A. Marcus, and M. L. Collard, "A task oriented view of software visualization," in *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*,