Improving Feature Location Practice with Multi-faceted Interactive Exploration

Jinshui Wang*, Xin Peng*, Zhenchang Xing†, and Wenyun Zhao*
*School of Computer Science, Fudan University, Shanghai, China
09110240018@fudan.edu.cn, pengxin@fudan.edu.cn, wyzhao@fudan.edu.cn
†School of Computer Engineering, Nanyang Technological University, Singapore
zcxing@ntu.edu.sg

***Abstract*—Feature location is a human-oriented and information-intensive process. When performing feature location tasks with existing tools, developers often feel it difficult to formulate an accurate feature query (e.g., keywords) and determine the relevance of returned results. In this paper, we propose a feature location approach that supports multi-faceted interactive program exploration. Our approach automatically extracts and mines multiple syntactic and semantic facets from candidate program elements. Furthermore, it allows developers to interactively group, sort, and filter feature location results in a centralized, multi-faceted, and intelligent search User Interface (UI). We have implemented our approach as a web-based tool *MFIE* and conducted an experimental study. The results show that the developers using *MFIE* can accomplish their feature location tasks 32% faster and the quality of their feature location results (in terms of F-measure) is 51% higher than that of the developers using regular Eclipse IDE.**

I. Introduction

When performing software maintenance tasks, developers often need to investigate the system's code base to find and understand program elements (e.g., classes, methods) that are pertinent to a specific feature. Such program comprehension activity, often termed as *feature location* (or *concept location*) [1] in the literature, is a human-oriented and information-intensive process [2].

To help developers accomplish feature location tasks, various feature location approaches have been proposed, including ones based on lexical information (e.g., identifiers and comments) in the code [3], on structural dependencies (e.g., call graph) of the program [4], [5], on dynamic traces as the program executes [6], [7], and on the hybrid of several information sources [8], [9], [10]. The output of these approaches is a ranked list of program elements that are pertinent to the task at hand. Researchers have also applied techniques, such as Formal Concept Analysis (FCA) [11], PageRank [12], to better filter or rank the relevance of program elements in the result list.

The recent developer-oriented study by Wang et al. [2] suggests that feature location is a *human-oriented information seeking* process during which developers must interactively search, browse and navigate a *multi-dimensional information space*. Each dimension describes program elements from a specific syntactic or semantic facet. However, existing feature location approaches simply consider feature location as an one-shot activity of $search \rightarrow results$. Thus, these approaches focus on technology-oriented issues such as searching algorithms and ranking strategies.

When performing feature location tasks with these approaches, developers often feel it difficult to formulate an accurate feature query (e.g., feature descriptions or keywords). Furthermore, it is hard for them to determine the relevance of returned results, especially when a large number of results are returned. Therefore, an intuitive interactive exploration approach is required to better support the human-oriented and information-intensive process of feature location. The approach should meet the information needs of developers for efficiently exploring and understanding the multi-faceted information space of the feature location results. Moreover, the approach should allow developers to begin with a rough query and gradually refine and adjust it based on previous feedbacks (i.e., returned results).

In this paper, we present an approach that supports *multi-faceted interactive exploration* for feature location. The goal of our approach is to ease the exploration and understanding of potentially relevant program elements by allowing developers to interactively group, sort, and filter feature location results in a centralized, multi-faceted, and intelligent search User Interface (UI). Our approach begins with an initial feature query provided by a developer and proceeds with an iterative process. In each step, it automatically extracts or mines multiple syntactic and semantic facets from returned program elements by leveraging static program analysis and data mining techniques. These syntactic and semantic facets include package structure, inheritance hierarchy, usage dependencies, and intent. Then, it presents the extracted and mined facets in a multi-faceted search UI [13]. Based on dynamically generated facets and grouping and sorting of search results, a developer can make quick and accurate decisions about the relevance of candidate program elements to a feature, and determine whether to further refine the feature query or return to a previous search step.

We have implemented our approach as a web-based tool *MFIE* (short for Multi-Faceted Interactive Explorer), which can be provided as a web-based service or integrated with Integrated Development Environments (IDEs) like Eclipse. To evaluate if developers can use *MFIE* to accomplish their feature location tasks more efficiently and effectively, we conducted an experimental study in which 20 developers performed four feature location tasks on an open-source system (JEdit) using *MFIE*. We compared the performance of 10 developers (i.e., the experimental group) against the performance of the other 10 developers using the Eclipse IDE (i.e., the control group) on the same feature location tasks.

We found that the developers were able to effectively explore mined multi-facets in multi-faceted search UI to locate program elements pertinent to the given feature location tasks. We also found that using mined multi-facets and multi-faceted search UI the developers found it easier to perform the assigned tasks and were more confident in their success in completing the tasks. In fact, the developers with *MFIE* support completed the tasks 32% faster than those without the support. Furthermore, the quality of feature location results (in terms of F-measure) of the developers with *MFIE* support is 51% higher than that of the developers without the support.

The remainder of the paper is organized as follows. Section II reviews related work. Section III describes the proposed approach. Section IV introduces the implementation of the proposed approach in *MFIE*. Section V presents the results of an experimental study and discusses some threats to our study. Section VI discusses the advantages and possible improvement of our approach. Section VII concludes the paper with a summary of our findings.

II. Related Work

Related work of our research spans three aspects, i.e., feature location techniques, information seeking and program exploration, data summarization and abstraction.

A. Feature Location

Many (semi-)automatic techniques have been proposed in the area of feature location (or concept location). These techniques use information retrieval (IR) [3], static analysis [4], [5], dynamic analysis [6], [7], or a hybrid of several analysis techniques [8], [9], [10] to improve developers' efficiency and performance in feature location tasks. A systematic literature survey of feature location techniques has been presented in [1]. Moreover, some researchers also use IR, static or dynamic analysis techniques to recover traceability links between different kinds of artifacts (e.g., between documents and source code [14] or duplicated bug reports [15]).

These technology-oriented approaches perform matching between features and program elements in a non-interactive way. In contrast, some research on feature location has been focused on better supporting interactive exploration by an iterative/incremental process or improved organization of search results. Lucia et al. [16] proposed an incremental process for IR-based traceability recovery, which allows software engineer to control the number of validated correct links and the number of discarded false positives by incrementally decreasing the similarity threshold. In [17], Lucia et al. empirically compared tracing performances achieved by participants using the "one-shot" process and the incremental process. Their study indicated that incremental process can improve tracing accuracy and reduce effort to analyze the inferred links. Poshyvanyk et al. [11] proposed an approach that can provide additional structure among the results of feature location by generating a concept lattice using FCA (Formal Concept Analysis). Developers can determine whether a category in the concept lattice is relevant by examining its label, and their search effort thus can be reduced by exploring only relevant categories in the lattice.

Compared with existing interactive feature location techniques, our approach supports a more systematic exploration process by providing multi-faceted categories in each search step and a complete record of exploration history for navigating exploration paths back and forth.

B. Information Seeking and Program Exploration

JQuery [18] is a source code browser that combines the advantages of a hierarchical browser with the flexibility of a query tool. Its hierarchical browser supports navigation among program elements based on particular kinds of relationships such as supertype/subtype and caller/callee. Ferret [19] supports program exploration by answering conceptual queries such as "What calls this method?" and "What transactions changed this element?". The answers are based on the integration of different sources of information such as static relationships, dynamic calls and revision history. Suade [20] recommends additional program elements for investigation based on their estimated structural relevance to a previously-identified set of program elements. Alwis et al. [21] conducted a comparative study with three exploration tools (JQuery, Ferret, and Suade) and reported that these tools had little apparent effect.

These tools support program exploration by a pre-defined question & answer process or an incremental navigation process from previously-identified program elements. In contrast, our approach allows developers to provide a rough query in the beginning and incrementally refine it with automatically generated multi-faceted categories.

Faceted search as an information exploration technique has been widely applied in a number of domains such as e-commerce and digital library. Faceted search allows a user to explore desired information from search results organized according to a set of category hierarchies, each of which corresponds to a different facet (dimension or feature type) relevant to the targeted domain [13]. Thus, the facets provided can be seen as a set of filters for the user. A faceted search interface allows the user to progressively narrow down the choices by choosing from a list of suggested query refinements [22]. In existing applications of faceted search, multi-faceted categories are usually created manually, although assignment of documents to categories can be automated to a certain degree of accuracy [13]. In contrast, in our approach the facets in each step are automatically extracted or mined from source code. A main challenge here is how to generate meaningful labels for users to understand and choose relevant facets. Our approach addresses this challenge by grouping

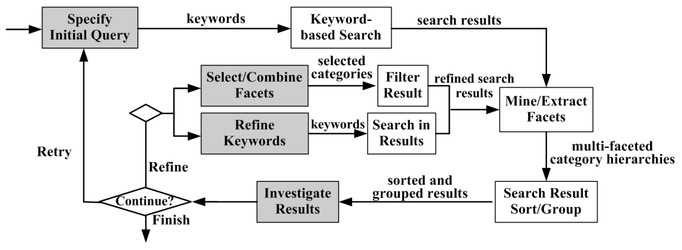


Fig. 1. Process Overview

candidate program elements using semantic clustering and abstracting descriptive labels for each generated cluster.

C. Data Summarization and Abstraction

Rastkar et al. [23] proposed an automated approach that produces a natural language summary for crosscutting source code concerns, which describes both what a concern is and how the concern is implemented. Kim et al. [24] proposed a rule-based program differencing approach that automatically discovers and summarizes systematic code change as logic rule. Sridhara et al. [25] presented an automatic technique for identifying code fragments that implement high level abstractions of actions and generating natural language description for them. The objectives of these approaches are to ease developers' program comprehension. In contrast, our approach generates multi-faceted categories that can be used to classify program elements for developers to explore feature search results and refine their feature queries.

III. The Approach

In this section, we first give an overview of the exploration process supported by our approach. We then detail the underlying techniques.

A. Overview

To implement multi-faceted and interactive feature location, our approach employs an incremental and iterative exploration process starting with an initial search-based query. Figure 1 presents an overview of our approach, showing the main steps and input/output in the process. The four gray rectangles in the figure represent steps involving human interaction.

To locate program elements (e.g., methods, classes) relevant to a feature, a developer first specifies a query consisting of a set of keywords. As the initial search results will be iteratively refined in the following steps, it is not necessary for the developer to come up with a perfect query initially. In other words, for this initial step, recall is much more important than precision. Based on the keywords specified by the developer, an initial set of candidate program elements can be obtained by using techniques such as regular expression matching and IR. Then our approach automatically extracts or mines multiple syntactic and semantic facets from candidate program elements. Each facet provides a category hierarchy for exploring candidate program elements. Based on multi-faceted categories, the search results can be sorted and grouped accordingly. The mined/extracted facets and the sorted/grouped search results help the developer better understand what choices he has so that he can determine what to do next after investigating the results. The developer can finish feature location process when he successfully identifies relevant program elements. Or he may choose to retry (i.e., start a new exploration process).

The developer can also continue the exploration process by asking smarter questions. This will increase the likelihood of success on feature location tasks. There are two alternatives for the developer to ask smarter questions. He can choose to refine the initial search-based query by issuing queries with more precise keywords based on the current result set and the multiple facets generated. Our approach searches in the current result set with the refined keywords and narrow down search results. Alternatively, he can choose to select/combine categories in one or more facets. Although the developer does not explicitly issue new queries in the second way, he essentially refines his query by using additional information of multi-faceted categories to filter the current search results. Based on the filtered set of candidate program elements, a new iteration of exploration can be started.

To better support the interactive exploration, our approach allows a developer to roll back to a previous search step in the exploration process. A developer may find that the exploration process has deviated from his expectation when he realizes that the desired results may have been filtered out from the result set. Therefore, it is helpful to allow a developer to roll back to a previous step and try other exploration pathes from there. In our current implementation (see Section IV), the exploration history of a feature location process is recorded in a tree-like structure with descriptive labels for each step. With history explorer, a developer can roll back to a previous step by selecting corresponding node and then click "Navigate" button to roll back to the step.

B. Multi-Faceted Filters

The multi-faceted categories in our approach are automatically mined or extracted from candidate program elements and dynamically updated during the iterative exploration process.

Currently, our approach supports the following five facets for result filtering:

1) **Package Structure**. This structure facet reflects package structure extracted from candidate program elements, showing how they are structured in different packages.
2) **Inheritance Hierarchy**. This structure facet reflects type hierarchy extracted from candidate program elements.
3) **Intent**. This intent facet categorizes candidate program elements by semantic clusters mined from their source code. It reflects concerns or topics of program elements.
4) **Use**. This dependency facet categorizes candidate program elements by the modules on which they depend for implementation.
5) **UsedBy**. This dependency facet categorizes candidate program elements by the modules that use them for implementation.

These above five facets can be categorized in three aspects, i.e., hierarchical structure, program intent, and usage dependency. Hierarchical structure facets reflect how program elements are intentionally structured in packages or type hierarchy, similar to Package Explorer or Type Hierarchy views of modern IDEs (e.g., Eclipse). Program intent facet categorizes program elements by relevant topics and concerns involved in program elements. Usage dependency facets differentiate program elements by the modules using them or used by them. It should be noted that these five facets are not completely orthogonal. For example, topic and concern information may also be reflected by package structure and type hierarchy in structure facets.

We now introduce how each facet is extracted or mined from candidate program elements and how it is used for result filtering.

1) Structure Facets: The extraction of the two structure facets is straightforward. The package structure is extracted from the package structure of candidate program elements in search results. Similarly, the inheritance hierarchy is extracted from the inheritance relations of classes/interfaces declared in candidate program elements.

When a package in **Package Structure** facet is selected, all the program elements that are contained in the package are selected from the current result set. When a class or interface in **Inheritance Hierarchy** facet is selected, all the program elements that are declared in the class and its subclasses are selected from the current result set.

2) Intent Facet: Categories in **Intent** facet are mined from the source code of candidate program elements in the current result set. To reflect the intent of program elements at a higher level of abstraction, we group candidate program elements using semantic clustering technique [26] and abstract descriptive labels for each generated cluster (see Section III-C). The generated clusters thus construct the categories in **Intent** facet and are displayed by their descriptive labels.

When a category in **Intent** facet is selected, all the program elements that are contained in corresponding semantic cluster are selected from the current result set.

3) Dependency Facets: Categories in the two dependency facets are mined from the dependency context of candidate program elements in the current result set (see Figure 2). For **Use** facet, we group all the program elements that candidate program elements in the current result set depend on by semantic clustering and use the generated clusters as categories. For **UsedBy** facet, we group all the program elements that depend on candidate program elements in the current result set by semantic clustering and use the generated clusters as categories. For example, in the example shown in Figure 2, program elements that candidate elements in the current result set depend on are grouped into semantic clusters u_1 to u_j that constitute categories in **Use** facet; program elements that depend on candidate elements in the current result are grouped into clusters g_1 to g_k that constitute categories in **UsedBy** facet. Similar to **Intent** facet, descriptive labels are generated for each cluster and displayed in corresponding facets for developers to investigate.

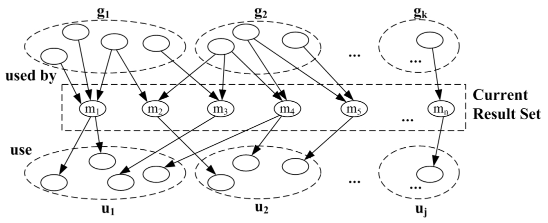


Fig. 2. Categories in Use/UsedBy Facets

When a category in **Use** facet is selected, all the program elements that depend on elements in the corresponding cluster are selected from the current result set. When a category in **UsedBy** facet is selected, all the program elements that are used by elements in the corresponding cluster are selected. For example, in the example shown in Figure 2, if u_1 is selected in **Use** facet, m_1, m_3 and m_4 will be selected from the current result set; if g_1 is selected in **UsedBy** facet, m_1, m_2 and m_3 will be selected.

C. Semantic Clustering and Labeling

Semantic clustering reveals concerns and topics of source code by grouping source artifacts that use similar vocabulary [26]. In our approach, semantic clustering is used to mine categories for **Intent** facet and two dependency facets. To that end, clustering algorithm need to first group relevant program elements into meaningful clusters and then generate descriptive labels for developers to investigate the relevance of program elements.

Given candidate program elements or their dependent elements, our approach first transforms a set of program elements into documents for clustering. The identifier names and comments of each program element (method or class) are transformed into a document by standard IR pre-processing steps, i.e., tokenization, filtering out stop words and word stemming. Then the produced documents are grouped into clusters with labels indicating their intent. In our current implementation (see Section IV), we apply the Lingo algorithm [27] provided by `Carrot2`[1] (an open-source search results clustering engine) for the purpose of semantic clustering. Lingo reverses the traditional order of cluster discovery by first finding good, conceptually varied cluster labels and then assigning documents to the labels to form clusters [27]. It can generate longer, often more descriptive labels than other clustering algorithms. This characteristic greatly helps developers to understand and select categories in **Intent**, **Use**, and **UsedBy** facets.

D. Search Result Grouping and Sorting

In addition to being used as result filters, multi-faceted categories generated in each step can also be used to group candidate program elements. Each facet groups program elements from a different dimension. For example, **Intent** facet groups program elements by concerns or topics implemented by them, and **Use** facet and **UsedBy** facet group program

[1]http://project.carrot2.org

elements by their usage dependencies. With these groups from different dimensions and their descriptive labels, developers can have an overview about the distribution of search results in a short time. They can thus examine a group of search results from a specific dimension. For example, from **Intent** facet, a developer can find that current candidate program elements are distributed in a set of modules implementing different concerns. He can then select a group with the most relevant concerns and investigate candidate elements in this group.

In addition to grouping, search results can also be sorted by their relevance to the feature query to facilitate the navigation and examination of search results. The relevance of a program element (usually a method) is estimated by the weighted occurrence count of the keywords given in the query. Keywords in class name have the highest weight, method name the second, method body the third. Categories in each facet are also sorted. For the three facets generated by semantic clustering, i.e., **Intent**, **Use**, **UsedBy**, categories are sorted by the number of results involved in each category. The categories in the other two facets, i.e., **Package Structure** and **Inheritance Hierarchy**, are sorted by package or class name. With sorted categories in each facet, developers can quickly find categories with desired number of results.

IV. Tool Implementation

We have implemented our approach in a web-based tool (*MFIE*) with a multi-faceted search UI. *MFIE* provides mechanisms to facilitate developers to investigate returned search results of a feature query and select/combine facets. Furthermore, *MFIE* keeps track of a complete exploration history and allows developers to navigate back and forth among exploration steps. In the current implementation, *MFIE* supports feature location of Java programs at method level.

A. Layout

MFIE has a typical multi-faceted search UI as shown in Figure 3. The four areas in the UI are search panel, result list, facet panel and history explorer.

A developer can enter multiple keywords in search panel as a feature query. He can start a new feature location process (click "Search" button) or click "Search in Results" button to refine current search results.

The candidate program elements returned in each step are sorted and displayed in the result list. In this area, each result (Java method) is sorted and displayed with its package name, method name, and a snippet of statements involving one or more keywords given in the query. *MFIE* also generates tags for each result method for quick understanding of the result method (see Section IV-B for tag generation). A developer can examine detailed descriptions (e.g., source code) of a result method by clicking the link on the method name.

The five facets generated from the current search results are organized in facet panel. Categories of a facet are displayed as a tree with descriptive labels and the number of candidate program elements involved in a category.

MFIE keeps track of a complete exploration history of a feature location process. This exploration history is displayed in history explorer using a tree-like structure describing exploration paths (see Figure 4 for an example and Section IV-C for explanation).

B. Result Investigation and Facet Selection

The search results returned and facets mined in each step are organized and displayed in a way that developers can easily investigate the relevance and distribution of returned results and make informed decisions about query refinement or restarting a new search.

Candidate program elements are sorted in the result list by their estimated relevance to the query. As the method body of a result method may be too long to be fully displayed in the result list, *MFIE* generates a small snippet for each result. The snippet is generated by locating the first occurrence of the searched keywords in the method body and excerpting a fragment of about 250 words around that location. When shown in the result list, searched keywords in the snippet are highlighted, and special code elements such as Java keywords, numbers and comments are displayed in different colors in a similar style to modern IDEs such as Eclipse.

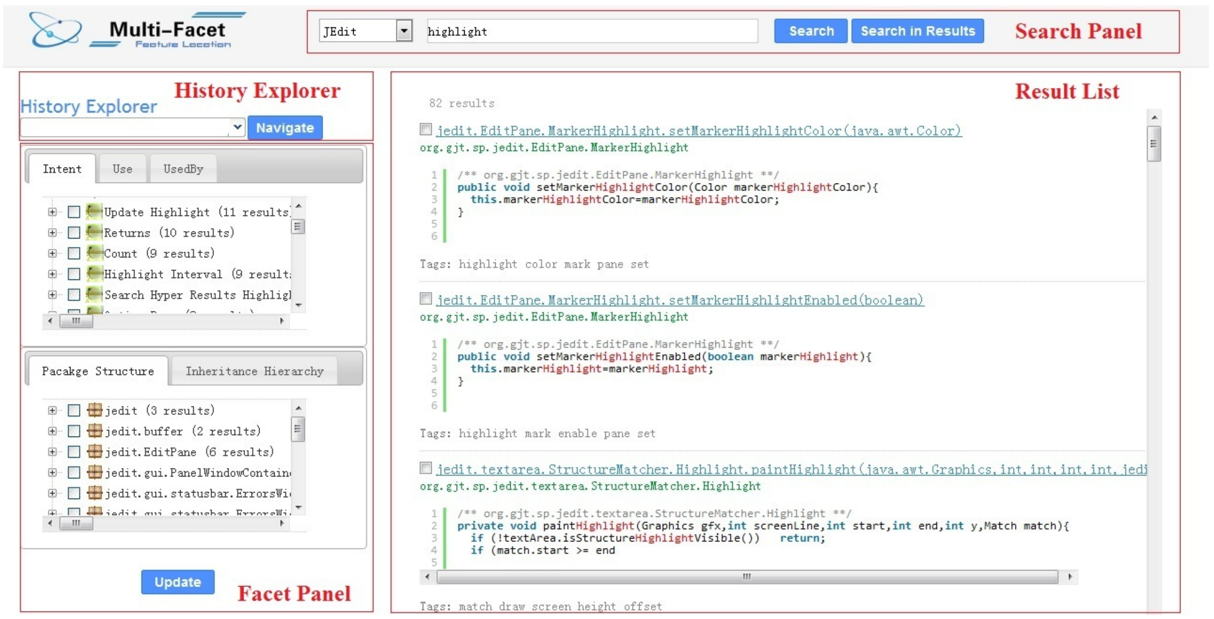
To allow a developer to have a quick understanding of the search results, *MFIE* generates a set of tags for each result method. The tags are generated by selecting the five highest weighted terms for each search result (Java method) using term-frequency/inverse-document-frequency (TF/IDF) metric. TF/IDF is commonly used in information retrieval techniques such as Vector Space Model (VSM). The weight of a term in a method is calculated as $f = \frac{n}{\sum_k n_k}$, where n is the number of occurrences of the term in a method and $\sum_k n_k$ is the sum of the number of occurrences of the term in all methods.

MFIE also provides a detailed information window (not shown in this paper) for developers to examine complete information about a result method, including source code, caller and callee methods, and fields accessed by the method.

To facilitate developers examining search results by facets, *MFIE* provides an instant search-preview feature. This feature allows a developer to preview filtered results by choosing a category in a facet. Filtered search results that are involved in the chosen category are immediately displayed in the result list for the developer to examine. The developer can quickly preview different categories in the same or different facets. Note that categories chosen in instant search-preview will not be used to refine the feature query immediately. If the developer decides to refine his feature query with the chosen categories, he can click the "Update" button. Then the chosen category will be incorporated into the feature query and the search results will be further filtered with the refined query. At the same time, all the categories in each facet are updated for the new search results.

C. Exploration History

To support developers to navigate back and forth through the exploration history of a feature location process, *MFIE*



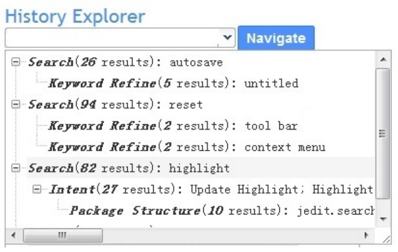
```
JEdit    highlight
```

```
Update Highlight (11 results
Returns (10 results)
Count (9 results)
Highlight Interval (9 result
Search Hyper Results Highligl
```

```
jedit (3 results)
jedit.buffer (2 results)
jedit.EditPane (6 results)
jedit.gui.PanelWindowContain
jedit.gui.statusbar.ErrorsWi
```

```
82 results

jedit.EditPane.MarkerHighlight.setMarkerHighlightColor(java.awt.Color)
org.gjt.sp.jedit.EditPane.MarkerHighlight

/** org.gjt.sp.jedit.EditPane.MarkerHighlight **/
public void setMarkerHighlightColor(Color markerHighlightColor){
  this.markerHighlightColor=markerHighlightColor;
}

Tags: highlight color mark pane set

jedit.EditPane.MarkerHighlight.setMarkerHighlightEnabled(boolean)
org.gjt.sp.jedit.EditPane.MarkerHighlight

/** org.gjt.sp.jedit.EditPane.MarkerHighlight **/
public void setMarkerHighlightEnabled(boolean markerHighlight){
  this.markerHighlight=markerHighlight;
}

Tags: highlight mark enable pane set

jedit.textarea.StructureMatcher.Highlight.paintHighlight(java.awt.Graphics,int,int,int,int,jedi
org.gjt.sp.jedit.textarea.StructureMatcher.Highlight

/** org.gjt.sp.jedit.textarea.StructureMatcher.Highlight **/
private void paintHighlight(Graphics gfx,int screenLine,int start,int end,int y,Match match){
  if (!textArea.isStructureHighlightVisible())   return;
  if (match.start >= end

Tags: match draw screen height offset
```

Fig. 3. MFIE Layout



```
Search(26 results): autosave
    Keyword Refine(5 results): untitled
Search(94 results): reset
    Keyword Refine(2 results): tool bar
    Keyword Refine(2 results): context menu
Search(82 results): highlight
    Intent(27 results): Update Highlight; Highlight
        Package Structure(10 results): jedit.searcl
```

Fig. 4. History Explorer

provides a history explorer as shown in Figure 4. The exploration history of a feature location process is represented as a tree. Each node in the tree represents a step in the process and records the relevant exploration information, including the refined facet or keywords, and the number of results returned in the step. The root of the tree denotes the beginning of the process. When a developer refines his feature query by refining keywords or selecting/combining facets, a new child node is created under the node of current search step. Therefore, each path in the tree represents a continuous refinement from the initial feature query. For example, to locate the feature "Highlight Terms in HyperSearch Results", a developer starts an exploration process with the initial query "highlight" and gets 82 results. And then, he identifies three relevant categories in **Intent** facet (e.g., "Update Highlight"). He chooses these categories for further refinement and gets 27 results. As he still feels that the returned results include some irrelevant methods, he chooses two categories in **Package Structure** facet (e.g., "jedit.search") and finally gets 10 results.

V. Evaluation

Our approach to multi-faceted interactive exploration for feature location is intended to help developers better perform feature location tasks. To evaluate if our *MFIE* tool meets this goal, we conducted a user study to investigate the following three questions:

Q1 How does a developer use mined multi-facets when they are available?

Q2 Can our *MFIE* tool help a developer achieve better performance in feature location tasks?

Q3 Can our *MFIE* tool help a developer perform a feature location task more easily?

A. Design

Our study involved 20 participants (13 graduate students and seven senior undergraduate students) from our school of computer science. These 20 participants were divided into two "equivalent" groups (G_1 and G_2) based on our pre-experiment survey on their capability, including software development experience, familiarity with java, familiarity with Eclipse. This allows us to perform a "fair" comparison of their performance on the same set of tasks. Group G_1 was the experimental group that used *MFIE* to perform their feature location tasks. Group G_2 was the control group that used Eclipse IDE to perform the same set of tasks.

The tasks used in our study were selected from the benchmarks[2] provided in [1]. For each task, the participants were requested to find the program elements (methods in this study) that are specifically relevant to a given feature, i.e., the program elements where the initial changes need to be made in order to fix a bug in a relevant feature [1]. Based on an overall consideration of the complexity of the subject system, participants' familiarity with the system, the number of documented features, we chose JEdit among the five projects provided in the benchmarks as the subject system. We further selected the following four features from the 34 features provided by the benchmarks as the task set of our study. These four features

[2]available at: http://www.cs.wm.edu/semeru/data/benchmarks/

have moderate difficulty and can be well understood by the participants.

1) Autosave Turned off for Untitled Documents
2) Text Block Selection via Gutter Right-Click
3) Highlight Terms in HyperSearch Results Window/Color for Type
4) Customize Tool Bar and Context Menu: Reset Button

The participants were requested to accomplish their tasks with *MFIE* or Eclipse IDE. They were given two hours to accomplish their tasks but can submit their results earlier if they thought that they had finished the tasks. As *MFIE* is a static approach, the control group using Eclipse IDE was allowed to use all the search and exploration features provided by the IDE, but not allowed to run and debug the subject system. Before each experiment we provided a half-hour tutorial and training on using *MFIE* and Eclipse IDE.

Because we need to analyze the actions and processes of each participant in completing the assigned tasks, we required the participants to run a full-screen recorder once they started working on the tasks. Furthermore, all the participants were asked to fill in a post-study questionnaire to provide feedback about the perceived difficulty of their tasks, challenges or difficulties in the process, and their experience on tool usage.

We evaluated participants' performance on all the tasks by precision, recall and F-measure. Given a feature f, let P_f be the percentage of correctly reported links between features and program elements (i.e., precision) and R_f be the percentage of actual links between features and program elements reported (i.e., recall). Given a set of feature location tasks T consisting of a set of features F, we computed the overall precision P_T and recall R_T for T as follows: $P_T = \sum_{f \in F} P_f/|F|, R_T = \sum_{f \in F} R_f/|F|$. F-measure for T is then computed as $F_T = (1+b^2)/(1/P_T + b^2/R_T)$ to reflect a weighted average of the precision and recall. In our study, following the treatment in [28], we set b to 2, i.e., recall is considered four times as important as precision, because we deem that finding missing links is more difficult than removing incorrect links.

B. Results: Use of Multi-Facets (Q1)

By analyzing the screen-recorded videos, we obtained the number of times that participants used different facets during their feature location processes (see Table I). The **Intent** facet has the highest usage frequency and every participant used **Intent** facet at least once during the whole experiment. Our analysis of the videos and post-study questionnaires and interviews suggests that participants used **Intent** facet more frequently because:

- High readability. By reading the descriptive labels of categories in **Intent** facet, participants can roughly infer whether a category is relevant to the given task.
- Reducing mental demand of participants. Participants may not only be able to obtain good enough result by selecting categories from **Intent** facet, they may also derive appropriate keywords from descriptive labels of categories listed in **Intent** facet.
- Reducing the high dependance on good keywords. Depending on only keywords may filter out many methods that are relevant but do not contain the given keywords.
- Easy to understand. Although **Use** and **UsedBy** facets also provide descriptive labels, participants still need to distinguish use and used-by dependencies between search results and categories. This incurs a certain amount of overhead.
- Do not need to interpret the location of method. When using facet such as **Package Structure**, participants have to infer the relevance of packages to a given task by interpreting package name. In contrast, **Intent** facet is much simpler to use.

Next to **Intent** facet, **Package Structure** facet has been used 25 times in total, much more than the use of **Inheritance Hierarchy** facet. This result matches our expectation. The granularity of the categories in **Package Structure** facet is more coarse-grained than that of the categories of **Inheritance Hierarchy** facets. Participants thus usually feel it easier to determine the relevance of candidate program elements by package. Furthermore, participants are generally more familiar with the UI of **Package Structure** facet, because it simulates Package Explorer view commonly found in modern IDEs.

TABLE I
USAGE OF MULTI-FACETS

	#Retry	#Keyword Refinement	Facet Refinement					#History
			#Intent	#Use	#UsedBy	#Package	#Inheritance	
P1	29	9	8	0	0	3	0	0
P2	9	1	1	0	0	7	0	0
P3	12	2	7	0	0	2	0	0
P4	8	4	1	0	0	2	0	2
P5	16	5	5	0	0	4	0	2
P6	25	0	3	2	0	1	0	1
P7	31	0	9	1	0	2	2	0
P8	20	0	10	1	3	2	0	3
P9	29	1	3	0	0	0	0	0
P10	32	1	1	0	0	2	0	0
Total	211	23	48	4	3	25	2	8

C. Results: Improvement on Performance (Q2)

We evaluate the performance improvement of *MFIE* by comparing the precision, recall and F-measure of the experimental group (G_1) and the "equivalent" control group (G_2).

1) Hypotheses: We introduce the following null and alternative hypotheses to evaluate how different the performance of the experimental and control groups is.

H0: The primary null hypothesis is that there is no difference between the performance of participants who use *MFIE* and Eclipse.

H1: An alternative hypothesis to H0 is that there is significant difference between the performance of participants who use *MFIE* and Eclipse.

2) Results of Individual Participants: The participants provided 1 to 17 methods for each task and none of them provided empty result list for a task. Table II and Table III present the participants' performance (in terms of precision, recall and F-measure), the number of results submitted for each task, and the time used for completing the four tasks in the experimental group and the control group respectively.

3) Results of Hypotheses Testing: We use paired sample t-tests to evaluate the null hypothesis H0 in terms of precision, recall and F-measure. We evaluate the hypotheses at a 0.05 level of significance. The results of these three tests are shown in Table IV. Based on the results we reject the null hypothesis H0 for all the measures of precision, recall and F-measure. Therefore, we accept the alternative hypothesis H1, i.e., there is significant difference between the performance of participants who use *MFIE* and Eclipse.

Our further analysis of the feature location results indicates that, compared with Eclipse IDE, using *MFIE* can significantly improve the recall of feature location results but may decrease the precision. Our inspection suggests that this decrease was due to the fact that with multi-faceted categories participants tend to be aware of much more methods that could be potentially relevant to a given feature. This often made the participants in G_1 submit more results than those in G_2, which resulted in the decrease on precision.

In terms of the significant improvement on F-measure (i.e., the weighted average of precision and recall), our data shows that *MFIE* did help developers achieve better performance in feature location tasks.

D. Results: Ease of Feature Location Tasks (Q3)

In the post-experiment questionnaires, all participants were requested to rate the difficulty of our feature location tasks (1 being very easy, and 5 being very hard). From the results in Table V, we can see that participants who used Eclipse rated the difficulty at 3.4 ($\pm$ 0.699) on average, and those who used *MFIE* rated the difficulty at 2.5 ($\pm$ 0.527) on average.

TABLE II
PERFORMANCE OF THE EXPERIMENTAL GROUP USING *MFIE* (G_1)

Participant	Recall	Precision	F-Measure	#Results	Time
P1	0.438	0.37	0.42	8.25	42m46s
P2	0.238	0.308	0.247	4.25	40m12s
P3	0.417	0.268	0.37	7.25	18m07s
P4	0.533	0.356	0.474	8.25	22m24s
P5	0.408	0.431	0.404	5.5	49m29s
P6	0.540	0.396	0.458	8.5	71m03s
P7	0.543	0.475	0.515	8.5	68m59s
P8	0.275	0.272	0.251	6.75	61m03s
P9	0.438	0.33	0.39	11.5	45m45s
P10	0.533	0.372	0.48	8	57m01s
Average	0.436	0.358	0.401	7.675	47m41s
Std. Dev.	0.109	0.066	0.092	1.951	17m50s

TABLE III
PERFORMANCE OF THE CONTROL GROUP USING ECLIPSE (G_2)

Participant	Recall	Precision	F-Measure	#Results	Time
P11	0.104	0.333	0.116	1.75	60m51s
P12	0.161	0.5	0.182	2.5	107m07s
P13	0.149	0.275	0.162	3	84m53s
P14	0.219	0.38	0.229	3.5	87m54s
P15	0.13	0.458	0.146	2.5	51m56s
P16	0.233	0.792	0.27	2	55m27s
P17	0.704	0.503	0.626	8.5	86m21s
P18	0.524	0.767	0.55	4.75	39m16s
P19	0.172	0.25	0.174	6.25	64m27s
P20	0.176	0.313	0.190	4.25	62m49s
Average	0.257	0.457	0.265	3.9	70m06s
Std. Dev.	0.196	0.192	0.177	2.125	20m38s

All the participants in G_1 rated the difficulty to be easy to medium and half of them rated the difficulty to be easy. One of the participants (P7) commented that with *MFIE* he was not that afraid of the large number of results returned by the initial queries. In many cases he was able to find proper keywords or categories to refine his queries by investigating the generated categories in different facets (especially **Intent** facet) and the code snippets of several result methods ranked at the top of the result list. Furthermore, although it is sometimes hard for participants to determine relevant categories or keywords for query refinement, with multi-faceted search UI they can often easily exclude obviously irrelevant categories. In contrast to G_1, half of the participants in G_2 rated the difficulty to be hard. They often felt helpless when a large number of results were returned. However, if they used more specific keywords, it was often the case that only few results were returned. In such situation, they usually had no idea about how to refine their queries.

In addition, we can see from Table II and Table III, the average time used to complete all tasks using *MFIE* is about 22m less than that of using Eclipse. Therefore, our data suggests that our *MFIE* tool did help developers perform a feature location task more easily.

TABLE V
RATED DIFFICULTY OF FEATURE LOCATION TASKS

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Avg
2	2	3	3	3	2	2	3	3	2	2.5
P11	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**	**Avg**
4	3	4	4	3	3	3	2	4	4	3.4

E. Threats to Validity

A major threat to the internal validity of the experimental study is the differences in the capabilities of the two groups of participants. This may threaten our assumption of the "equivalence" between the experimental group and the control group for the comparison between *MFIE* and Eclipse IDE. To address this threat, we had tried our best to allocate participants with comparative capabilities into different groups based on the pre-study survey of the participants' capabilities and our evaluation of their capabilities.

The other major threat to the internal validity of the study is the likely insufficient training of the tools (*MFIE* and Eclipse IDE) used in the study. We believe this factor has much greater influence on the experimental group using *MFIE*, since most of the participants use Eclipse IDE regularly while *MFIE* is a completely new tool for the participants. Even in such a disadvantage situation, the participants using *MFIE* still significantly outperformed those using Eclipse. We believe that if the participants were given more systematic training on *MFIE*, they could make even better use of multi-facets and multi-faceted search UI (e.g., history explorer) and then achieve even better performance.

Major threats to the external validity of the study lie in the fact that we only conducted one experimental study on a relatively small Java system. All our findings may not be applicable to systems with millions lines of code that are maintained by teams of professional developers. In the study,

TABLE IV
RESULTS OF T-TESTS OF HYPOTHESES, FOR THE VARIABLE PRECISION, RECALL, AND F-MEASURE. MEASUREMENTS ARE REPORTED IN THE FOLLOWING COLUMNS: MINIMUM VALUE, MAXIMUM VALUE, MEDIAN, MEANS (μ), VARIANCE (σ^2), THE DEGREES OF FREEDOM (DF), THE PEARSON CORRELATION COEFFICIENT (PC), STATISTICAL SIGNIFICANCE (p), T_{crit}, AND THE T STATISTICS.

H	Var	Approach	Samples	Min	Max	Median	μ	σ^2	DF	PC	T	T_{crit}	p	Decision
H0	Recall	Eclipse	10	0.104	0.704	0.174	0.257	0.039	9	0.647	-3.752	2.262	0.005	Reject
		MFIE	10	0.238	0.543	0.438	0.436	0.012	9					
	Precision	Eclipse	10	0.250	0.792	0.419	0.457	0.037	9	0.956	2.426	2.262	0.038	Reject
		MFIE	10	0.268	0.475	0.363	0.358	0.004	9					
	F-measure	Eclipse	10	0.116	0.626	0.186	0.265	0.031	9	0.723	-3.389	2.262	0.008	Reject
		MFIE	10	0.247	0.515	0.412	0.401	0.008	9					

we considered only one programming language (Java) and compared with one development environment (Eclipse). Other development environments with more advanced search and exploration mechanisms such as IntelliJ IDEA may weaken the advantage of *MFIE* over traditional IDEs. Further studies are required to generalize our findings in large-scale industrial projects and with developers who have sufficient domain knowledge and familiarity with the subject systems.

VI. DISCUSSION

Our initial study indicates that multi-faceted interactive exploration can help developers perform their feature location tasks more efficiently because it allows developer to ask smarter questions during feature location. Furthermore, our study also reveals opportunities to further improve our approach, for example by providing more comprehensible facets and supporting guided exploration and navigation.

A. Helping Developers Ask Smarter Questions

Feature location tasks can be accomplished only when a developer knows what he want and how to ask for it. Feature location often fails when a developer does not know which choices he has and he does not know what he does not know.

According to our earlier study [2], when using Eclipse experienced developers usually can get familiar with the program more easily and faster than less experienced developers. Thus, experienced developers may specify proper queries, explore proper program elements, and properly adjust their queries and strategies based on the investigation of returned search results (usually by reading code). In contrast, less experienced developers may feel it difficult to specify proper queries and cannot effectively collect hints from returned search results. Furthermore, they were more likely to get lost in exploration, for example when they explore program by usage dependencies.

By using mined multi-facets and multi-faceted search UI provided by *MFIE*, the gaps between experienced and less experienced developers were significantly reduced. This is indicated by the reduced standard deviation of all the individual measures of the participants in our study, i.e., recall, precision, F-measure, the number of submitted results, time to complete the tasks (see Table II and Table III).

Different from most existing tools that require a developer to formulate feature queries explicitly, the navigable facets generated by *MFIE* help a developer who are not familiar with the program to ask "smarter" questions because these facets expose all the choices available to him. Furthermore, the grouping of search results by different facets provides the developer more abstract and structured feedbacks about the search results. As a result, the developer, even less experienced, can better refine their feature queries based on the hints that he observe from different facets. Finally, multi-faceted query refinement, result grouping, together with history exploration, allow the developer to explore valid paths to reach relevant program elements in a way that he is familiar with and may be more suitable to the task, rather than to be limited by pre-defined paths offered by the tools. This greatly increases the likelihood that a developer may find relevant program elements.

B. Providing More Comprehensible Facets

In our approach, categories in all the facets are automatically extracted or mined from candidate program elements returned by a feature query. These categories aim to group search results for examination and provide means for query refinement. Therefore, the label and description of a category needs to provide an accurate and abstract summary of program elements involved in the category. This summary serves as hints so that developers can easily decide the relevance of program elements to the feature. The readability of these labels and descriptions thus becomes a key for the effectiveness of multi-faceted categories.

Currently, our approach provides package names in **Package Structure** facet, class names in **Inheritance Hierarchy** facet, and descriptive labels generated by semantic clustering algorithm in **Intent**, **Use**, and **UsedBy** facets. In the future, we will apply more sophisticated techniques such as natural language summaries [23] and tag cloud [29] to generate more comprehensible descriptions for multi-faceted categories. It would also be useful to experimentally compare different techniques to evaluate their effectiveness in generating comprehensible descriptions for extracted or mined facets.

C. Supporting Guided Exploration and Navigation

The interactive exploration process provided by our approach allows a developer to explore along and switch among different exploration paths. In this process, the challenge lies in how to provide flexible exploration options for developers without making them lost. The history explorer currently provided by *MFIE* supports the navigation through exploration history with a tree-like structure. This history explorer may be improved with more intuitive and interactive visualization techniques, e.g., 3D visualization used in CodeCity [30]. Such improved history explorer would help to better answer a set of questions during feature location process, such as "how did I reach the current situation?" and "what have been filtered out in the past steps?".

Furthermore, our multi-faceted and interactive approach to feature location can be further improved by providing guided navigation. A system employing guided navigation would assist the developer in achieving their goals by suggesting likely lines of enquiry at critical points. It can update all navigation options at each click, showing only the valid next questions a developer can ask, and eliminating possible dead-end paths. This kind of guided navigation, possibly implemented by intelligent wizards, can be a useful complement to our current strategy of "multi-faceted query plus interactive exploration".

VII. Conclusion and Future Work

In this paper, we have presented a multi-faceted interactive approach for feature location. Our approach begins with an initial feature query and proceeds with a highly interactive exploration process. During the exploration process, our approach automatically extracts or mines multiple syntactic and semantic facets from candidate program elements and allows developers to interactively group, sort, and filter feature location results in a multi-faceted and intelligent search UI.

The approach has been implemented as a web-based tool *MFIE* and evaluated with an experimental study involving 20 gradate and senior undergraduate students. The results indicate that the developers were able to effectively explore mined multi-facets in multi-faceted search UI to accomplish their feature location tasks. Our comparative study shows that the developers with *MFIE* support completed the tasks faster (32% less time) with much better results (51% higher by F-measure).

Our approach and experimental study have provided initial evidence that feature location practice can be improved with a multi-faceted interactive exploration process. Our future work will experimentally investigate more sophisticated techniques for generating more comprehensible facets. We will also provide more intuitive history navigation for better support interactive exploration process during feature location. We are also interested in deploying our *MFIE* tool as a cloud service. This will allows us to collect exploration paths of a community of developers who use *MFIE* to locate feature in a subject system. These exploration paths can be mined to discover "wisdom of crowds" that can then be used to support intelligent guided exploration and navigation.

Acknowledgment

This work is supported by National High Technology Development 863 Program of China under Grant No.2012AA011202.

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *J. Softw.: Evol. and Process*, 25(1):53–95, 2013.

[2] J. Wang, X. Peng, Z. Xing, and W. Zhao. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *ICSM*, pages 213–222, 2011.

[3] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, 2004.

[4] M. Trifu. Using dataflow information for concern identification in object-oriented software systems. In *CSMR*, pages 193–202, 2008.

[5] M.P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):18:1–18:36, 2008.

[6] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi. Locating program features using execution slices. In *ASSET*, pages 194–203, 1999.

[7] A.D. Eisenberg and K. De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *ICSM*, pages 337–346, 2005.

[8] D. Poshyvanyk, Y.G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, 2007.

[9] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

[10] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[11] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC*, pages 37–48, 2007.

[12] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *ICPC*, pages 14–23, 2010.

[13] M.A. Hearst. Clustering versus faceted categories for information exploration. *Commun. ACM*, 49(4):59–61, 2006.

[14] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[15] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, pages 461–470, 2008.

[16] A.De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[17] A. De Lucia, R. Oliveto, and G. Tortora. IR-based traceability recovery processes: An empirical comparison of "one-shot" and incremental processes. In *ASE*, pages 39–48, 2008.

[18] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*, pages 178–187, 2003.

[19] B. de Alwis and G.C. Murphy. Answering conceptual queries with ferret. In *ICSE*, pages 21–30, 2008.

[20] F.W. Warr and M.P. Robillard. Suade: Topology-based searches for software investigation. In *ICSE*, pages 780–783, 2007.

[21] B. de Alwis, G.C. Murphy, and M.P. Robillard. A comparative study of three program exploration tools. In *ICPC*, pages 103–112, 2007.

[22] J. Koren, Y. Zhang, and X. Liu. Personalized interactive faceted search. In *WWW*, pages 477–486, 2008.

[23] S. Rastkar, G.C. Murphy, and A.W.J. Bradley. Generating natural language summaries for crosscutting source code concerns. In *ICSM*, pages 103–112, 2011.

[24] M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Trans. Softw. Eng.*, 39(1):45–62, 2013.

[25] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, pages 101–110, 2011.

[26] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, 2007.

[27] S. Osinski and D. Weiss. A concept-driven algorithm for clustering search results. *IEEE Intell. Syst.*, 20(3):48–54, 2005.

[28] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Trans. Softw. Eng.*, 32(1):4–19, 2006.

[29] B.Y.L. Kuo, T. Hentrich, B.M. Good, and M.D. Wilkinson. Tag clouds for summarizing web search results. In *WWW*, pages 1203–1204, 2007.

[30] R. Wettel and M. Lanza. CodeCity: 3D visualization of large-scale software. In *ICSE*, pages 921–922, 2008.