Efficient Study of Domain-Based Transactions Ranges Program Mechanisam

Dr. V.Anandam
Dept. of CSE, Vardhaman College of Engg., Kacharam, Shamshabad, Hyderabad, Telangana, India

Abstract
The scope of this Transactions ranges from the mechanisms through the development of principles to the application of those principles to specific environments. Specific topic areas include: (a) development and maintenance methods and models, e.g., techniques and principles for the specification, design, and implementation of software systems, including notations and process models; (b) assessment methods, e.g., software tests and validation, reliability models, test and diagnosis procedures, software redundancy and design for error control, and the measurements and evaluation of various aspects of the process and product; (c) software project management, e.g., productivity factors, cost models, schedule and organizational issues, standards; (d) tools and environments, e.g., specific tools, integrated tool environments including the associated architectures, databases, and parallel and distributed processing issues; (e) system issues, e.g., hardware-software trade-off; and (f) state-of-the-art surveys that provide a synthesis and comprehensive review of the historical development of one particular area of interest.

Keywords
MODIS, IR, CP, CN, FP, FN, GHSOM, Semantic Retrieval Algorithm, Conceptual Graph

I. Introduction

From its beginning in the compiler community, source code analysis has spread into a variety of software engineering (SE) tasks. However, these roots have left a bias toward the kinds of analyses useful to a compiler. Recently, a growing number of researchers have extracted information of no interest to a compiler. A common example is the semantic information found in the natural language of a program's source code (e.g., within the program's identifiers). Information retrieval (IR) focuses on the analysis of natural language in an effort to classify text documents as relevant or irrelevant to a specific query. Recently, IR has expanded to include techniques for determining answers to questions and for organizing text based on topics. The goal of this entry is to survey the application of IR to the challenges encountered during the first "half" of the SE development process—that is, problems encountered up through a product's initial release. Examples include requirements formation and the need for software repositories. To limit the scope, the survey favors techniques presented with sufficient technical detail to allow reproduction. Furthermore, the entry is biased toward techniques that report results from empirical study. The remainder of this entry first introduces necessary IR terminology and then describes common IR techniques applied to multiple SE problems. These two sections are included for completeness and may be skipped by readers familiar with IR. The bulk of the entry considers the application of IR to the SE activities encountered during initial software development. This is presented roughly in the order that these activities are found in the software development life cycle. Finally, the entry considers some forward-looking thoughts on the future of IR in SE. A companion entry considers the application of IR techniques to problems encountered during software maintenance and evolution (see "Information Retrieval Applications Software Maintenance and Evolution" entry).

II. Terminology

This section introduces the key terminology used in IR techniques and two key metrics used to evaluate them. For consistency, terminology has been normalized across the techniques considered. Common terminology is introduced here; terminology specific to a single technique is defined when used. The section ends with a glossary of the major terms used. To begin with, the term artifact denotes the atomic "entity" traditionally returned in response to a query. In SE, artifacts include requirements documents, design documents, source code, test cases. When only source code artifacts are considered, the term module is used to refer to the basic unit of source code to which a technique is applied. This may, for example, be a method, a class, a function, or a file. There are occasional exceptions when a technique specifically applies to a particular syntactic entity. There is a need to differentiate two kinds of semantics used by many of the techniques. First, programming language semantics refers to the meaning of a program as a state transformer from inputs to outputs. Second, natural language semantics refers to the meaning inherent in the natural language appearing in a program (most often in its identifiers and internal comments). The remaining discussion defines two key metrics used to evaluate IR techniques: precision and recall (informally "the whole truth and nothing but the truth"). Precision measures the proportion of retrieved artifacts that are relevant, which indicates how well a tool distinguishes between relevant and non-relevant artifacts. It can also be interpreted as the probability that a retrieved artifact is relevant. Recall is the proportion of relevant artifacts that are retrieved, which indicates how well a tool retrieves relevant artifacts. Recall can also be interpreted as the probability that a relevant artifact is retrieved. To illustrate these two metrics, consider fault prediction where a module can be either faulty (F) or not faulty (NF). For each possibility, a fault prediction technique may correctly or incorrectly label the module. This gives rise to four partitions. The definitions of precision and recall are based on the number of modules in each partition. In the following, the variables CP, CN, FP, and FN are used to represent these four counts:

- CP ¼ count of correct positives (predicted as F and actually in F)
- CN ¼ count of correct negatives (predicted as NF and actually in NF)
- FP ¼ count of false positives (predicted as F but actually in NF)
- FN ¼ count of false negatives (predicted as NF but actually in F)

Based on these four counts, precision is CP/(CP þ FP) (i.e., the ratio of correctly predicted as faulty to all predicted as faulty) and recall is CP/(CP þ FN) (i.e., the ratio of correctly predicted as faulty to number of faulty modules). For example, consider a program with ten modules M1 ... M10 where M1 ... M3 are faulty but M2 ... M6 are predicated as faulty. The values of CP, CN, FP, and FN are as follows:

		Predicted	
		NF	F
Actual	NF	CN = 4 ($M_7 \ldots M_{10}$)	FP = 3 ($M_4 \ldots M_6$)
	F	FN = 1 (M_1)	CP = 2 ($M_2 \ldots M_3$)

Here the precision is CP/(CP þ FP) ¼ 2/5 and recall is CP/ (CP þ FN) ¼ 2/3. As precision and recall form the most important measures of an IR technique's performance, a better understanding of the trade-offs between the two is helpful. Antoniol, Gue´he`neuc, and Tonella consider three example SE applications that illustrate these trade-offs.[1] The first example, aspect mining, employs IR to identify potential modules from which to "grow" aspects. The second example, feature location, aims at identifying the parts of a program activated when exercising a given functionality (e.g., the methods to be modified to fix a bug). The final example, design pattern identification, seeks to identify groups of classes whose structure and organization match the structure and organization advocated by a given design pattern. These three tasks illustrate elements of a continuum in which the balance between precision and recall is highly dependent on the task at hand. In migration-oriented tasks, such as aspect mining, IR is used to provide a starting point for developers. Thus, precision of the candidate modules is of high importance; however, recall (completeness) is less important since only a developer can completely identify the modules that belong to an aspect. Impact analysis tasks, such as feature location, require a balance between precision and recall to provide engineers with as few methods as possible while ensuring that important methods are not overlooked. Finally, comprehension-oriented tasks such as design pattern identification, which begins by identifying candidate micro architectures, favor high recall at the cost of precision. This is because identifying candidate micro architectures is time-consuming and error prone for developers while discriminating among identified micro architectures is quite quick and easy

III. Software Repositories

For years, libraries have allowed programmers to reuse common functions (e.g., qsort). Scaling this up, a software repository is a collection of reusable modules. Access to such a repository makes an engineer more efficient and increases software quality when previously "burned-in" modules are reused. However, to be useful, a repository must provide a sufficient number of modules covering a sufficiently wide spectrum of domains, and it must provide a satisfactory retrieval system by which an engineer can locate an appropriate module.[8] Techniques for building module repositories are divided roughly into two groups: IR-based approaches that use natural language and AI-based approaches that use extracted knowledge. In the former, no semantic knowledge is used and no interpretation of the modules is given; thus, a tool attempts to characterize the modules rather than understand them. For IR, the natural-language documentation (e.g., manual pages and comments) forms a rich source of information from which to organize repositories. Once a repository of reusable modules is assembled, effective search capability is essential. In the ideal case, a search provides an exact match for an engineer's needs. However, it is more common for no such match to exist. In this case, the engineer needs to be able to browse the repository to find the module that best matches the desired functionality. A wide range of component categorization and searching methods has been proposed, from the simple string search to faceted classification and behavioral matching [9]. These different methods involve different trade-offs between performance and implementation cost. This section considers two techniques that automatically cluster modules for placement into a repository. While cluster has no commonly agreed-upon definition, herein it is considered to be a group of objects whose members are more similar to each other than to the members of any other group. The first approach to automatically building a repository uses a growing hierarchical self-organizing map (GHSOM) [8]. Such maps are based on an artificial neural network referred to as a Self-Organizing Map (SOM). In essence, a SOM determines a winning neuron for each input vector using a similarity measure (e.g., Euclidean distance) to compare the weights of the input vector to those of each neuron. This process continues until learning converges to a stable set of weight vectors for each neuron. After training, the topology of the data becomes geographically explicit in that similar input data are mapped onto nearby regions of the map. Traditional SOMs are not practical when the number of software modules is large, as intensive iterative training is required. A recent improvement, the GHSOM, is built from a hierarchy of multiple layers of SOMs. A GHSOM grows from a single neuron in two dimensions: horizontally (by increasing the size of a SOM) and hierarchically (by increasing the number of layers). The upper layers of a GHSOM provide a coarse organization of the major clusters in the data, whereas the lower layers offer a more detailed view. In more detail, horizontal growth starts by initializing the weight vector of each neuron with random values. It then performs traditional SOM learning for a fixed number of iterations. Finally, two neurons are identified: 1) the neuron with the largest deviation between its weight vector and the input vectors it represents and 2) its most dissimilar neighbor. The approach then inserts a new row or a new column between these two neurons (with weight vectors initialized as the average of their neighbors), and the traditional SOM learning is repeated. This process continues until the mean quantization error of the map drops below a user-defined threshold. Hierarchical growth checks each neuron to find out if its quantization error is above a userspecified threshold. If so, a new SOM is assigned at a subsequent layer of the hierarchy. This SOM is trained with the input vectors mapped to the high-quantization neuron. When building software repositories, the weight vectors are initially composed of tf-idf values for the key concepts (non-stop words) extracted from the source code and the documentation using a single-term free-text indexing scheme. During an empirical study of 273 samples from three different domains, it was determined that key concepts occurring in fewer than five modules or in more than 218 modules should be omitted. After removal, both techniques successfully created a topology-preserving representation of the three domains. However, when dealing with a large number of software modules, GHSOM behaved better than SOM in the sense that architecture was determined automatically during its learning process. Moreover, GHSOM was able to reveal the inherent hierarchical structure of the data in its layers and provided the ability to select the granularity of the representation at different levels of the GHSOM. The second repository construction approach is unusual. Most IR techniques ignore the location of terms within an artifact. This

second approach incorporates term proximity into the GURU tool for automatically building software repositories [8]. It assembles a conceptually structured software repository based on natural-language documentation (e.g., manual pages and comments). Repository construction is done in two steps: first, attributes are extracted from the documentation by identifying lexical affinities (LAs); second, a hierarchy for browsing is generated. In general, an LA is the correlation between two units of language. For repository building, these units are words and are restricted to those separated by, at most, five intervening words within a single sentence. The LAs are further filtered based on resolving power: a combination of the quantity of information associated with each word and the frequency of occurrence of the LA within the considered artifact. The quality of information from the LA < w1, w2 > is defined as -log(P(w1) * P(w2)), where P(w) is the observed probability of an occurrence of w in the corpus (therefore more frequent words carry less information). The power of the LA < w1, w2 > occurring f times is then f -log(P(w1) * P(w2)). For example, the LA appears as often as in the mv manual page; however, it has a lower resolving power because the word system has a lower quantity of information (appears more often) than overwrite in the documentation. Finally, to compare different artifacts, resolving power is normalized to a standard z score, denoted hereafter using r. Based on the normalized power of each artifact's LAs, artifact clusters are constructed. In doing so, only the LAs whose value is at least one standard deviation above the mean are retained. The process starts with each artifact as its own cluster and repeatedly merges the two most-similar clusters, until a single cluster remains. Here similarity is defined so that it takes resolving power into account: similarity(x, y) ¼ i rx(i) ry(i) where rx(i) is the standardized value of LA having index i in artifact x. Post construction, the repository can be queried using natural language. The result is a ranked list of modules; however, using the structure obtained during clustering makes it possible for a user to interactively inspect nearby modules. For example, this allows a user trying to "identify a process" to quickly go from the top-ranked kill man page to the ps man page that is clustered with it. In an empirical study using the AIX man pages, GURU performed better (higher precision and comparable recall) than the IBM-supplied Info Explorer on a collection of representative search tasks.

IV. Traceability Links Between Software Artifacts

Traceability links tie together software artifacts from stakeholders' initial requests to requirements specifications, design artifacts, models, reports, source code, and test cases [10-11] Maintaining these links is an arduous task. Information Retrieval Applications in Software However, inadequate links is one of the main factors contributing to project cost overruns and failures; thus, there is a need for tool support to (re)establish traceability links. Given that link maintenance is a costly manual process, several IR-based automatic and semiautomatic techniques have been proposed. This section outlines three such techniques. The main focus of the work to date has been on how to report candidate links to a user with maximum recall without sacrificing precision. LSI is the most popular retrieval technique used, although several other methods including VSM and probabilistic IR have been experimented with. No method has yet emerged as a clear favorite. The first technique focuses on discovering links between high-level requirements and low-level requirements [12]. Three different IR approaches are compared: VSM, VSM with manual identification of key phrases, and thesaurus retrieval. In the second approach, key phrases are sequences of k technical terms extracted from the definitions or acronyms sections of the requirements specification and manually added to each requirement. They increase the relevance of matches related to technical terminology. For the third approach, each thesaurus entry is formally a tuple (ti, tj, aij) where ti and tj are terms (either words or key-phrases) and aij 2 [0,1] is an expert-assigned similarity coefficient. The thesaurus's coefficients are created for the set of words present in the data dictionary and the acronym lists found in appendices of requirements documents. In the thesaurus approach, the cosine similarity equation is augmented by adding d(ti) q(tj) þ d(tj) q(ti) to the numerator for terms ti and tj, when terms ti and tj are related according to the hand-built thesaurus. Here, d(t) is the weight of term t in the design and q(t) is the weight of term t in the query. This gives added "credit" to artifacts that include words related to query words, but not actually found in the query. The results were compared with the commercial tool Super Trace Place and an analyst's judgments using two different data sets from NASA's publicly available Moderate Resolution Imaging Spectroradiometer (MODIS) project. The first set contained 10 high-level requirements and 10 lower-level requirements. The second contained 19 high-level requirements and 10 lower-level requirements. Comparing the three approaches, the VSM approach achieved a recall of 23% and precision of 18%. The VSM with key phrases approach achieved a recall of 27% and precision of 5%, and the thesaurus version achieved a recall of 85% and precision of 40%; thus, of the three, thesaurus retrieval performed the best. when compared to Super Trace Plus and the human analysts, it achieved higher recall but lower precision. The lower precision can be counted against the considerable human time differential. Constructing the thesaurus required only half an hour, while applying Super Trace Plus took four hours and the analyst took nine hours. The second link traceability technique extends the ADAMS tool to establish traceability links using an LSI based technique.[10] The extension requires three enhancements: an indexer, an SVD generator, and a query executor. The indexer updates the term artifact matrix, while the SVD generator recomputed the LSI's single-value decomposition. The third enhancement identifies traceability links. The tool is designed to be used iteratively beginning with the user identifying an initial set of links for each artifact a, links(a). The system then retrieves the set of links whose similarity to a is greater than or equal to a user-determined similarity threshold e, retrieved(a,e). Comparing these two sets yields four possibilities:

Inclusion Match(a, €)= links(a)∩ retrieved(a, €)
Exclusion Match(a, €)= links(a) ∩ retrieved(a,€)
Missing Links(a, €) =links(a) ∩ retrieved(a, €)
Warning Links(a, €)=links(a) ∩ retrieved(a, €)

Here the latter two sets contain links in need of attention. For example, Warning Links includes those links that the user may want to remove from links(a). The generation of these four sets is iterative with the engineer first updating links(a) and potentially lowering the threshold. The user stops iterating when links(a) and retrieved(a,e) come into agreement. Given the importance of maximizing recall while not sacrificing precision, the following methods of limiting the retrieved artifacts were experimented with:

1. Constant Threshold

All artifacts that have a cosine similarity score greater than a constant threshold are retrieved. A widely used threshold is e ¼

0.707, which corresponds to a 45angle between the corresponding vectors.

2. Variable Threshold
The top k percent of the returned artifacts are reported.

3. Cut Point
This is a traditional limit where n artifacts are selected.

In a case study, 150 artifacts were gathered from EasyClinic (a product developed by final year students at the University of Salerno, Italy). The program manages the operations required by a medical ambulatory. Artifacts include 30 use cases, 20 interaction diagrams, 63 test cases, and 37 code classes. To achieve 100% recall, a constant threshold of e ¼ 0.11, a variable threshold of k ¼ 10%, or a cut point of n ¼ 132 artifacts were required. Each method resulted in about 12% precision. The best results (recall ¼ 80% and precision ¼ 24%) were achieved by a e ¼ 0.28, k ¼ 31%, or n ¼ 46 artifacts. The final technique extends the above by modifying the threshold strategy and applying filtering to both dimensions of the similarity matrix [13].

V. Software Reuse
Software reuse is the use of existing software knowledge or artifacts to build new software. It has the potential to improve software quality, productivity, reliability, and maintainability [14]. Existing reuse algorithms can be classified as free text, faceted index, or semantic net based [15]. The free-text approach, to which IR's indexing technology is most applicable, extracts key concepts from each module to be used as search keys by engineers. In faceted index approaches, experts extract keywords from program descriptions and documentation; they then arrange the keywords by characteristics. Finally, the semantic-net approach uses a large knowledge base, a natural-language processor, and a semantic retrieval algorithm to retrieve software components. An age-old debate, first in the IR literature and later in the context of software reuse, considers the pros and cons of free-text retrieval versus controlled vocabulary, multifaceted retrieval. In short, some claim that free-text retrieval produces too many false positives and false negatives. However, controlled vocabulary involves the (significant) cost of building and maintaining vocabularies and of classifying/indexing components [9]. The four techniques considered in this section are considered in roughly chronological order. They all use the "building block" approach to reuse, where developers must find reusable components, assess their worth, and then potentially tailor them to the problem at hand. The first technique includes some history and perspective by considering an older faceted approach. The remaining three techniques are free-text techniques reflecting more recent trends. The final approach further exploits the fact that free text does not need structure by applying IR to the requirements phase (where typically only natural-language descriptions exist). Early work on reuse, done when faceted approaches were popular, includes that of Wood and Summerville who observe that a balance must be struck between the need for meaningful representation and ease of use [16]. For example, keywords provide ease of use and general applicability while lacking meaningful representation. On the other hand, with a faceted approach, significant time is required to construct the conceptual classification. Wood and Summerville describe an IR-based reuse system designed to store and retrieve software components based on frames (expert designed component descriptors). The approach uses a hierarchic (enumerative) classification scheme that demonstrates how individual keywords fail to provide an accurate description of software component purpose. The frames approach originated with the conceptual dependency technique used in natural language understanding to represent the semantics of an "understood" text [16]. Atomic frames capture one of three types of concepts: actions, nominal's, or modifiers. Actions correspond to the basic functions that software components perform; nominal's correspond to the objects that perform the functions; and modifiers refine actions and nominals. Each frame has a variable number of slots (which can be left unfilled). For example, the print frame has three slots:

Frame: print < Actor, Printee, Destination >
Example: print < more, t.c, terminal >
Example: print < lpr, t.ps, lj4 >

Atomic frames are designed to apply at the function level. For larger syntactic entities, aggregate frames are used. Each primitive in an aggregate includes a slot that refers to the aggregate; thus, a search that uncovers a part can lead to the whole. To perform a search, a user (partially) fills in a frame for the component to search for. In addition, more precise descriptions than afforded by keywords are supported by the meaningful relationships between concepts using descriptors. Recent trends favor free-text techniques. Example evidence of the turning point can be found in the work of Mili et al.[9] who note "this result [the superiority of free-text reuse] was surprising as earlier work had often shown that controlled vocabulary performed better than free text." Their experiment compared all-manual controlled-vocabulary retrieval with free-text retrieval. Instead of the typical IR based computation of recall and precision based on some abstract measure of "relevance," they used a measure that took into account "the true utility of the retrieved components." Further, they used a more realistic experimental protocol (closer to the way that such tools are used in practice), where a developer's decision to use a tool or not includes the estimated effort to build a component from scratch, the cost of using a tool, and the perceived track record of the tool. In contrast to past experiments, the study controlled only the search method used. Subjects could perform an unlimited number of searches and had no time limitation. For precision, Mile et al. used the ratio of the retrieved components that had non-zero pertinence to the total number of retrieved components. Pertinence is used because it relates to a developer's ability to solve the problem at hand. The pertinence of artifact A is taken in the context of a solution set S (i.e., A is useful "only if" the other components required to build a solution are retrieved with it). The pertinence of a set of artifacts is the sum of their individual pertinence. One implication of this definition is that total user satisfaction can be achieved with a subset of the relevant components, which is not the case for recall. Data from five subjects (all experienced Cþþ programmers) performing 11 queries were collected using a data set of about 200 classes and 2000 methods taken from the OSE (OTC (Overseas Telecommunications corporation) Software Environment) library. For each subject, each query was randomly assigned either the keyword-based or plain-text search method. Informally, plain-text retrieval yields better recall and somewhat better precision. Statistically, plain-text retrieval yields significantly better recall than controlled vocabulary-based retrieval (p ¼ 0.0500), while there is no statistical difference in their precision (p ¼ 0.3404). As stated earlier, these results run counter to previous experimental evidence where artifact retrieval experiments have consistently shown that controlled vocabulary-based retrieval yielded better

recall and precision than plain-text (although the difference was judged by many as being too small to justify the extra costs involved in controlled vocabulary-based retrieval). To explain these results, several hypotheses were investigated, but none were validated by the data. For example, controlled vocabulary might make the search more tedious, causing users to give up too easily, yielding lower recall. Plain-text retrieval might favor queries whose answers involved a mix of methods and classes. Multifaceted retrieval (e.g., based on content and time) might require more information than the user is able to provide in the early stages of problem solving (and then fails to capture a faithful expression of users' needs at later stages). Finally, the quality of indexing might be to blame. There are two potential weaknesses, but neither accounts for the observed difference in performance. These results complement an emerging consensus that while measured performance may favor controlled vocabulary retrieval it hardly justifies the cost. Perhaps more importantly, four subjects out of five preferred plain-text search. This preference is likely to persist with the increased exposure to plain-text search available from web search engines. Finally, Mili et al. suggest that multifaceted classification and retrieval are the wrong level of formality in two ways. First, when used in the early stages of a project, they coincide with analysis, which is fairly exploratory. A multifaceted search is too rigid and constraining because the solution is unformed, so a plain-text search is more appropriate. Second, after contemplating several designs, a developer may then start searching for components that would play a given role within a design, and multifaceted classification may not be expressive enough. The second free-text approach shows how the Patricia (Program Analysis Tool for Reuse) system can be used to search for components using semantic matching [15]. Patricia uses a knowledge base built from an ontology of interrelated domain terms and definitions to describe software components. This information is stored using a conceptual graph (CG): A system of logic-based semantic networks, an AI technique that expresses meaning in a form that is logically precise, humanly readable, and computationally tractable. Although the knowledge base adds information that is generally not available to a retrieval engine, this is considered free-text retrieval because the user inputs a free-text query and the knowledge base operates directly on software artifacts without a human-defined vocabulary. The ontology supports semantic matching between natural-language user queries and component descriptions using a (manually constructed) domain knowledge base. The similarity between a software component and a user query is defined as the maximum semantic intersection between any two sub graphs from the two CGs. This is computed as the maximal akin index of the concept's nodes in the CGs for the component and the query. The akin index is calculated as the number of links in the ontology between the definitions of pairs of concepts. For example, "otter is a mammal is an animal" results in an akin index of 2 for the concept pair (otter, animal).

Query Patricia extends Patricia to output metrics that describe the degree to which the user query matches the conceptual graphs of the observed software. An experiment comparing Query Patricia and human experts found that Query Patricia performs satisfactorily when compared to the manual approach, which is more accurate, tedious, time consuming, and error prone [15]. Finally, the most recent study illustrates how free text can be applied during the requirements phase where there is a lack of structured information [17]. Sterna and Rowe compared requirements from two large military systems (containing 577 and 3538 requirements). A manual comparison first assigned each requirement a subset of 35 keywords. These were used to avoid the 577 3538 pair wise comparisons. Only requirements sharing a keyword were manually compared. Requirements were also compared using cosine similarity based on a variant oftf-idf where idf is replaced by a simple word count. As a result, the rate of occurrence in all requirements is very significant in this application. Matching keywords produced 632 requirement pairs. Of these 453 had at least one word in common. However, precision was 0.26% for 90% recall and 2.98% for 10% recall. This is quite low, perhaps because the requirements used were short, and many differences occur in describing the same concept In some of these cases abbreviation expansion would help facilitate the matching.

VI. Conclusion

As software becomes ubiquitous, there is a growing need for tool support in its construction. Leaving its roots in the compiler community, modern tools exploit a wide range of information that is of little interest to a compiler. By focusing on IR techniques, this entry has presented a collection of such tools that exploit information contained in the natural language found in a program and its documentation. By organizing the presentation around the stages of the software life cycle, the entry highlights trends found within each stage and across all stages. An example is the growing focus on the text contained in a program's identifiers and its relation to the external documentation. The application of IR to SE has given rise to many useful tools in the areas of requirements discovery, maintaining software repositories, establishing trace-ability links, efficient software reuse, and effective software metrics. In particular, these tools show that useful information can be extracted from the natural-language contained in source code's identifiers and comments as well as other natural-language artifacts associated with a software project. Such artifacts can be manipulated by tools in tasks that previously required extensive human effort or provide an alternative perspective, as in the development of effective software metrics. Given that applying IR to SE is a relatively young endeavor, many new applications are likely to appear. Near term, these can be expected to leverage the diversity of new work from the IR community; however, as the field matures more IR-based techniques designed explicitly to solve SE problems should start to emerge.

References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: A Case Study", In Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'00), Zurich, Switzerland, February 29 - March 3 2000, pp. 227-230.

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.,"Recovering Traceability Links between Code and Documentation", IEEE Transactions on Software Engineering, Vol. 28, No. 10, October 2002, pp. 970 - 983.

[3] Bigger staff, T. J., Mitbander, B. G., Webster, D. E., "Program Understanding and the Concept Assignment Problem", Comm. of the ACM, Vol. 37(5), May 1994, pp. 72-82.

[4] Chen, K., Rajlich, V.,"Case Study of Feature Location Using Dependency Graph", In Proceedings of Intern. Workshop on Program Comprehension (IWPC'00), 2000, pp. 241-249.

[5] Chen, K., Rajlich, V.,"RIPPLES: Tool for Change in Legacy Software", In Proceedings of International Conference on Software Maintenance (ICSM'01), 2001, pp. 230 - 239.

[6] Clayton, R., Rugaber, S., Taylor, L., Wills, L.,"A Case Study of Domain-based Program Understanding", In Proceedings of 5th Workshop on Program Comprehension, Dearborn, MI, May 28-30 1997, pp. 102-110.
[7] Cubranic, D., Murphy, G. C.,"Hipikat: Recommending pertinent software development artifacts", In Proceedings of 25th International Conference on Software Engineering (ICSE'03), Portland, OR, May 2003, pp. 408-418.
[8] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., Harshman, R.,"Indexing by Latent Semantic Analysis", J. of the Amer. Soc. for Info Science, Vol. 41, 1990, pp. 391-407.
[9] Eisenbarth, T., Koschke, R., Simon, D.,"Locating Features in Source Code", IEEE Transactions on Software Engineering, Vol. 29, No. 3, March 2003, pp. 210 - 224.
[10] Etzkorn, L. H., Davis, C. G.,"Automatically Identifying Reusable OO Legacy Code", IEEE Computer, Vol. 30, No. 10, October 1997, pp. 66-72.
[11] Fischer, B.,"Specification-Based Browsing of Software Component Libraries", In Proceedings of ASE, 1998, pp. 74-83.
[12] Fiutem, R., Tonella, P., Antoniol, G., Merlo, E.,"A Cliche'-Based Environment to Support Architectural Reverse Engineering", In Proceedings of Intern Conference on Software Maintenance (ICSM '96), Nov 04 - 08 1996, pp. 319-328.
[13] Frakes, W.,"Software Reuse Through Information Retrieval", In Proc of HICSS, Kona, HI, Jan. 1987, pp. 530-535.
[14] Landauer, T. K., Foltz, P. W., Laham, D.,"An Introduction to Latent Semantic Analysis", Discourse Processes, Vol. 25, No. 2&3, 1998, pp. 259-284.
[15] Landauer, T. K., Laham, D., Foltz, P. W.,"Learning human-like knowledge by Singular Value Decomposition: A progress report", Advances in Neural Information Processing Systems, Vol. 10, 1998, pp. 45-51.