

Information Retrieval Applications in Software Development

David Binkley* **Dawn Lawrie**

Loyola College

Baltimore, MD

21210-2699, USA

{binkley, lawrie}@cs.loyola.edu

Keywords Natural Language Processing, Software Engineering, Requirements,
Software Repository, Traceability Links, Reuse, Metrics

Abstract

Information retrieval (IR) extracts and organizes natural language information found in unstructured text. Many of the challenges faced by software engineers can be addressed using IR techniques on the unstructured text provided by source code and its associated documents. A survey of IR-based techniques applied to software engineering challenges during the initial development process is presented. In particular, the following problems are considered: requirements discovery, maintaining software repositories, establishing traceability links, efficient software reuse, and effective software metrics. These techniques highlight the bright future that IR brings to addressing SE problems.

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 towards the kinds of analyses useful to a compiler. Recently, a growing number of researchers have has extracting 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 article 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 article is biased towards techniques that report results from empirical study.

The remainder of this article 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 article 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 article considers some forward looking thoughts on the future of IR in SE. A companion article considers the application of IR techniques to problems encountered during software maintenance and evolution [1].

2 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 software engineering, artifacts include requirements documents, design documents, source-code, test cases, etc. 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 sequel, the variables CP , CN , FP , and FN are used to represent these four counts:

CP = count of correct positives (predicted as F and actually in F)

CN = count of correct negatives (predicted as NF and actually in NF)

FP = count of false positives (predicted as F but actually in NF)

FN = count of false negatives (predicted as NF but actually in F)

Based on these four counts, precision is $CP/(CP + FP)$ (*i.e.*, the ratio of correctly predicted as faulty to all predicted as faulty) and recall is $CP/(CP + FN)$ (*i.e.*, the ratio of correctly predicted as faulty to number of faulty modules).

For example, consider a program with ten modules $M_1 \cdots M_{10}$ where $M_1 \cdots M_3$ are faulty but $M_2 \cdots M_6$ are predicated as faulty. The values of CP , CN , FP , and FN are as follows:

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

Here the precision is $CP/(CP + FP) = 2/5$ and recall is $CP/(CP + FN) = 2/3$.

As precision and recall form the most important measures of an IR technique's performance, a better understanding of the tradeoffs between the two is helpful. Antoniol et al. consider three example SE applications that illustrate these tradeoffs [2]. The first, *aspect mining*, employs IR to identify potential modules

from which to ‘grow’ aspects. The second, *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.

2.1 Glossary

Important terms that are used in multiple places in this article are defined briefly in this section where *Italics* in a definition denotes other terms defined in glossary.

Artifact an atomic ‘entity’ returned in response to a query

Cluster a group whose members are more similar to each other than members of any other group

Corpus a large (semi-structured) set of *artifacts*

Cosine similarity similarity measured as the cosine of the between two *artifacts* represented by vectors, often in a *VSM*

Hard word part of an identifier separated by *Word markers*

Latent Dirichlet Allocation replacement of *LSI* that allows a mixture of topic models rather than a single topic

Latent Semantic Indexing (LSI) a description of a corpus based on weighted vectors for each artifact

Metrics quantification of some aspects of a software system

Module basic unit of source code to which a technique is applied

Natural language semantics meaning inherent in a program's natural language (its identifiers and comments)

Precision the probability that a retrieved artifact is relevant

Programming language semantics meaning of a program as a state transformer

Recall the probability that a relevant artifact is retrieved

Requirements a description of what a system must do

Singular Value Decomposition (SVD) a factorization of a matrix that allows dimensionality reduction

Soft Word dictionary words and common abbreviations found in a *hard word*

Software repository a collection of reusable modules

Software Reuse the use of existing software, or software knowledge, to build new software

Stemming elimination of word suffixes

Stopping removal of terms unlikely to be helpful

tf-idf (term frequency - inverse document frequency) a method of calculating term weights often used in a *VSM*

Traceability links links between software artifacts and the documentation

Vector Space Model (VSM) artifacts represented by weighted vectors (see *tf-idf*)

Word marker underscores or camelCasing separating words in an identifier

3 IR TECHNIQUES

This section provides background on the various IR notions used in SE tools. It first defines the term-vectors used to represent artifacts and their *tf-idf* weighting. These are used in the Vector Space Model (VSM), cosine similarity, and Latent Semantic Indexing (LSI). Finally, stemming, stopping, and hard and soft words, are introduced.

Many techniques used in IR represent artifacts as n -dimensional weighted vectors $(x_1, x_2, \dots, x_n)$, where x_i represents the weight of the i^{th} term extracted from an artifact. In source code, *terms* are most often extracted from identifiers and comments. For example, in the following code, the extracted terms (in order) are int, distance, point, p1, p2, return, sqrt, x, and y.

```
int distance (point p1, point p2)
{
```

```

    return sqrt((p1.x - p2.x)2 + (p1.y - p2.y)2)
}

```

Using frequency counts for the weights, the term-weight vector for this function would be $\langle 1, 1, 2, 3, 3, 1, 1, 2, 2 \rangle$. Note that when one wants to represent multiple modules, the vectors often include many zeros corresponding to those terms that occur only in other modules.

In practice, simple counts do not make effective weights. Instead, weights are most often assigned using *tf-idf* (term frequency - inverse document frequency) [3]. In short, term frequency (within an artifact) shows how important the term is to that artifact. Document frequency (of the term) shows how generally important the term is. A high weight is achieved by a term that occurs much more than the average in an artifact, but is rare in the entire collection. This provides a method for weighting the importance of a term to an artifact relative to the frequency of the term in the entire collection. It also makes the vectors insensitive to common language terms (*e.g.*, programming language keywords) because of the *idf* factor.

For example, consider a collection of three source modules that include (only) the following terms:

Module M_1 — *score*, *goals*, *Chelsea* (five times)

Module M_2 — *score*, *winner*, *Chelsea*

Module M_3 — *score*, *goals*, *winner*

In this case, the artifact frequency of *score*, which appears in all three modules, means that it provides very little information to a query. In contrast, having *Chelsea* appear multiple times in M_1 , while being rare in the overall collection, means that it should receive considerable weight; in other words, M_1 has a

high probability of being of interest to any search that includes the search term *Chelsea*.

In the *Vector Space Model* (VSM), artifacts are represented by weighted vectors. The collection of artifacts, referred to as a *corpus*, forms a term-by-artifact matrix that captures the distribution of terms in the artifacts. Queries are also expressed as vectors. *Similarity* is measured using *cosine similarity*: the cosine of the angle between two vectors, where a vector represents either an artifact or a query. Thus, when searching, the cosine similarity of a query and each artifact is computed to determine the likelihood that an artifact is relevant to the query. A smaller angle between the two (and thus a larger cosine value) indicates a better match and thus a higher rank in the retrieved set. VSM works much like popular search engines (*e.g.*, Google) by creating a signature (in this case a weighted vector) for each element of interest.

Like VSM, the retrieval technique *Latent Semantic Indexing* (LSI) forms a corpus based on weighted vectors for each artifact. However, it applies techniques to estimate latent structure and thus reduce the dimensionality of the vectors. In more detail, any rectangle matrix, for example a $t \times d$ matrix relating terms to documents, can be decomposed into the product of three matrices: $T_0 S_0 D_0'$ such that T_0 and D_0 have orthonormal columns and S_0 is a diagonal matrix, referred to as the *Singular Value Decomposition* (SVD). In the SVD element $e_{i,i}$ is greater than or equal to element $e_{j,j}$ when $i < j$. This is used to construct a subspace, called the LSI subspace (or semantic space). Each vector (representing an artifact) from the corpus is translated into a vector in the LSI subspace. Setting less important entries of the SVD matrix to zero reduces the dimensionality of the vectors and thus helps with polysemy and synonymy, since such terms are grouped together in the subspace. The cosine between two vectors is again used to measure the semantic similarity between two artifacts.

Finally, three preprocessing steps are often employed in IR. First, *stopping* removes terms unlikely to be helpful in determining the relevancy of an artifact to a query. For English, example stopwords include ‘the’, ‘about’, and ‘can’. In software, they also include programming-language keywords and common library function names.

The second preprocessing step, *stemming*, eliminates suffixes so that the frequency of a word disregards its particular forms. There are several stemming techniques in existence including the Porter Stemmer [4] and the Krovetz Stemmer (k-stem) [5]. One of the main differences between these two is that the Porter Stemmer truncates words whereas the Krovetz Stemmer replaces a word with the unsuffixed version of the word. These two techniques stem English words; however, stemming techniques for many of the world’s languages have been developed.

Finally, software offers its own challenge when applying IR techniques. In particular, identifiers, one of the most abundant sources of natural-language text, do not necessarily include indicators that separate words and make prolific use of abbreviations. To address these issues, identifiers must be decomposed into their constituent words, hereafter referred to as *hard words*, as delineated by word markers (*e.g.*, the underscore character or camelCasing). Techniques for further dividing identifiers into individual words, known as *soft words* have also been presented [6, 7]. For example, the soft words **sponge** and **bob** are found in the hard word **spongebob**. One such technique employs a greedy algorithm that recursively identifies the longest remaining dictionary-word prefix or suffix extractable from a hard word.

4 REQUIREMENTS

The following sections consider the application of IR to problems faced by software developers roughly in the order that they are encountered in the software life cycle. Each section first briefly defines the problem considered, then describes its importance, and the current state of the art. This is followed by presenting the relevant IR techniques.

The first problem is the need for *complete* requirement specifications. This can be addressed by searching for *missing* requirements and thus makes a good introduction to the application of IR to SE because the use of natural language in most requirements specifications precludes more formal analysis. At present, the discovery of missing requirements is a highly labor intensive and error-prone task [8]. This section considers an IR-based approach to finding missing requirements.

One solution to this problem employs a repository of requirements from past projects. The approach first extracts key-concepts (*e.g.*, those having high *tf-idf* values) from each requirement in a new specification. These are then used to query both the new specification and a repository of past specifications. If the search turns up requirements not found in the new specification, then a potentially missing requirement has been identified and is presented to the user for consideration.

5 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 [9].

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 [10]. 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) [9]. 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. The closest neuron is the winner.

During training, its weights and those of its immediate neighbors are adjusted by moving them ‘towards’ the input vector. 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 user-specified 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 an 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 to repository construction approach is unusual. Most IR techniques ignore the location of terms within an artifact. The second technique incorporates term proximity into the GURU tool for automatically building software repositories [9]. 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 $\langle w_1, w_2 \rangle$ is defined as $-\log(P(w_1) * P(w_2))$, 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 $\langle w_1, w_2 \rangle$ occurring f times is then $f \times -\log(P(w_1) * P(w_2))$. For example, the LA $\langle \text{file, system} \rangle$ appears as often as $\langle \text{file, overwrite} \rangle$ 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 ρ .

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 kept. 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: $\text{similarity}(x, y) = \sum_i \rho_x(i) \times \rho_y(i)$ where $\rho_x(i)$ 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 which, 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 InfoExplorer on a collection of representative search tasks.

6 TRACEABILITY LINKS BETWEEN SOFTWARE ARTIFACTS

Traceability links tie together software artifacts from stake-holders' initial requests to requirements specifications, design artifacts, models, reports, source code, and test cases [11, 12]. Maintaining these links is an arduous task. However, inadequate links is one of the main factors contributing to project cost over-runs 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 semi-automatic 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 [13]. 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 (t_i, t_j, a_{ij}) where t_i and t_j are terms (either words or key-phrases) and $a_{ij} \in [0, 1]$ 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 $d(t_i) \cdot q(t_j) + d(t_j) \cdot q(t_i)$ to the numerator for terms t_i and t_j , when terms t_i and t_j 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 **SuperTracePlace** 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 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. Comparing it with **SuperTracePlus** 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 **SuperTracePlus** 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 [11]. The extension requires three enhancements: an indexer, an SVD generator, and a query executor. The indexer updates the $term \times artifact$ matrix, while the SVD generator recomputes 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 ε , $retrieved(a, \varepsilon)$. Comparing these two sets yields four possibilities:

- $InclusionMatch(a, \varepsilon) = links(a) \cap retrieved(a, \varepsilon)$
- $ExclusionMatch(a, \varepsilon) = \overline{links(a)} \cap \overline{retrieved(a, \varepsilon)}$
- $MissingLinks(a, \varepsilon) = \overline{links(a)} \cap retrieved(a, \varepsilon)$
- $WarningLinks(a, \varepsilon) = links(a) \cap \overline{retrieved(a, \varepsilon)}$

Here the latter two sets contain links in need of attention. For example, *WarningLinks* 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 $retrieved(a, \varepsilon)$ 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 $\varepsilon = 0.707$, which corresponds to a 45 degree angle 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 $\varepsilon = 0.11$, a *variable threshold* of $k = 10\%$, or a *cut*

point of $n = 132$ artifacts was required. Each method resulted in about 12% precision. The best results (recall = 80% and precision = 24%) were achieved by a $\varepsilon = 0.28$, $k = 31\%$, or $n = 46$ artifacts.

The final technique extends the above by modifying the threshold strategy and applying filtering to *both* dimensions of the similarity matrix [14]. The new *bounded threshold* approach combines both a constant threshold and variable threshold: the similarity of an artifact must exceed a constant lower bound ε and also be in the top k percent of the rank list of artifacts. This approach also extends existing techniques that filter artifacts by ranking only one dimension of the similarity matrix, since it ranks both. For example, when linking requirements and test cases, ranking might be used to select ‘good’ (similar) requirements. The proposed approach ranks *both* dimensions of the similarity matrix; empirically, this has the effect of removing many false negatives.

Three case studies were undertaken. All three used the following artifact categories: requirements (called use cases), the design, and the test cases. The first two studies, *Pacman* and *Calisto* (both student projects), used $\varepsilon = 0.707$ and the variable threshold of $k = 20\%$ and 50% respectively. Compared to the one-dimensional strategy, the two-dimensional strategy had a positive impact on precision, but a negative impact on recall. Furthermore, recall and precision were higher for the *use case* to *test* link recovery than for the *use case* to *design* link recovery.

The third case study was carried out at Philips Applied Technology using artifacts for a Philips DVD recorder. Unfortunately, the correct links are unknown so recall and precision cannot be computed. Informally LSI provided better results than the current tools being used by Philips. In fact, compared to the other two case studies, more links were recovered between the *use cases* and the *design*. It was hypothesized that this was because the level of granularity of the

use cases was more similar to that of the design than the tests. In the first two case studies, the granularity of use cases was more similar to tests.

7 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 [15]. Existing reuse algorithms can be classified as free-text, faceted index, or semantic-net based [16]. 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, multi-faceted 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 [10].

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 Sommerville who observe that a balance must be struck between the need for meaningful representation and ease of use [17]. For example, keywords provide ease of use and general applicability whilst lacking meaningful representation. On the other hand, with a faceted approach, significant time is required to construct the conceptual classification.

Wood and Sommerville 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 [17].

Atomic frames capture one of three types of concepts: actions, nominals, or modifiers. Actions correspond to the basic functions that software components perform; nominals 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. A search for matching frames returns a single best-match if possible. Otherwise, assuming multiple object slots are filled, single object matches are sought and finally, just matches of the action or object alone.

No empirical evaluation of the technique is performed as the authors deem such to be too subjective to be of value. Instead they provide subjective arguments for their representation. For example, high recall is supported by the conceptual classification of all specified terminology. 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. [10] 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 only controlled the search method used. Subjects could perform an unlimited number of searches and had no time limitation.

For precision, Mili 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 C++ programmers) performing 11 queries was collected using a data set of about 200 classes and 2000 methods taken from the OSE 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 ($p = 0.0500$), while there is no statistical difference in their precision ($p = 0.3404$).

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. Multi-faceted 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 multi-faceted classification and retrieval are the wrong level of formality in two ways. First, when used in the early stages of a project, it coincides with analysis, which is fairly exploratory. A multi-faceted 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 multi-faceted 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 [16]. 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 express 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 subgraphs 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).

QueryPATRicia 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 QueryPATRicia and human experts found that QueryPATRicia performs satisfactorily when compared to the manual approach, which was more accurate, tedious, time consuming, and error prone [16].

Finally, the most recent study illustrates how free-text can be applied during the requirements phase where there is a lack of structured information [18]. Stierna et al. compare 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 pairwise comparisons. Only requirements sharing a keyword were manually compared. Requirements were also compared using cosine similarity based on a variant of *tf-idf* where *idf* is replaced by the 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 (*e.g.*, “acft” for “aircraft” or “capable of sharing” for “shall be exportable”). In some of these cases (*e.g.*, the first example), abbreviation expansion would help facilitate the

matching [7].

8 METRICS

Software metrics attempt to quantify aspects of a software system. Such measures facilitate decision making (consider Tom DeMarco’s famous quote “You can’t control what you can’t measure”). For example, quality metrics allow test engineers to focus their efforts on those modules most likely to contain faults [19, 20]. Existing metrics are primarily based on structural aspects of software, such as the number of attributes referenced in a method, the number of lines of code, etc. Recently IR-based metrics, which focus on natural-language semantics, have been proposed. This section considers three: first, IR-based coupling and cohesion metrics, and then two IR-based metrics aimed at measuring software quality.

Coupling and cohesion metrics can be used to assess design quality, predict software quality, identify fault prone modules, and identify reusable components [21, 22]. Coupling metrics measure the degree to which modules require other modules. Modules with high cohesion provide a crisp abstraction of a concept from the problem domain. Thus, cohesion metrics attempt to measure the degree to which elements of a module belong together.

Three metrics based on LSI are considered, although other IR methods such as the VSM or a Bayes classifier could be used. Each is based on the conceptual similarity between two modules, which is computed from a corpus constructed out of identifiers and comments.

The conceptual similarity between classes, *CoCC*, is defined in terms of the conceptual similarity between a method and a class (*CSMC*) and the similarity between methods (*CSMM*). These three similarities are defined as follows:

$$\begin{aligned}
\text{let } x &= \frac{vm_k^T \times vm_j}{(||vm_k||_2 \times ||vm_j||_2)} \quad (\text{the cosine similarity of } vm_k \text{ and } vm_j) \\
CSMM(m_k, m_j) &= \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \\
CSMC(m_k, c_j) &= \frac{\sum_{i=1}^t CSMM(m_k, m_{j_i})}{t} \\
CSCC(c_l, c_j) &= \frac{\sum_{i=1}^t CSMC(m_{j_i}, c_l)}{t}
\end{aligned}$$

where c denotes a class, m a method, vm the vector for method m in the LSI space, $c_k \neq c_j$, and c_j has the methods $\{c_{m_1}, \dots, c_{m_t}\}$.

Based on $CSCC$, the first of the three metrics, the *Conceptual Coupling of Class* c is defined as $CoCC(c) = \sum_{i=1}^n CSCC(c, c_i) / (n - 1)$ where $c \neq c_i$.

If class c is strongly coupled to the rest of the classes in the system, then $CoCC(c)$ will tend closer to 1. The second metric, $C^3(c)$, measures the *Conceptual Cohesion of a Class*. It is defined as the average $CSMM$ value for all the methods in the class unless this average is less than zero, in which case $C^3(c) = 0$. The third metric, $LCSM$, measures the lack of conceptual similarity between the methods of a class; thus, it is an inverse measure of cohesion where higher values indicate lower cohesion. $LCSM$ is the number of pairs of methods from a class that share a common similar method minus the number of pairs that do not share a common method. $LCSM$ is defined to have a minimum of zero. Here methods m_i and m_j are *similar* if $CSMM(m_i, m_j)$ is higher than the average for the class.

To determine whether conceptual coupling captures different information from existing structural coupling metrics, a collection of open-source software systems were analyzed. For $CoCC$, the underlying orthogonality was measured by performing *Principal Component Analysis* on the 979 classes studied. As $CoCC$

appears alone in its own principal component, it captures different information from the structural metrics. C^3 and *LCSM* were evaluated using pairwise comparison with each of the structural metrics and each other (statistically, the use of pairwise comparison greatly increases the probability of Type I error). This evaluation used 34 classes with 522 methods. The analysis reveals significant correlations between C^3 and two of the structural metrics (ICH and LCOM5). The first is not very surprising as ICH uses the number of invocations of other methods weighted by the number of parameters in its formula. On the other hand, no correlation was found between LCSM and any of the structural metrics nor between LCSM and C^3 .

To obtain further insight, several modules were examined by hand. Of these, the lowest C^3 value came from a large class having 1397 LOC and 804 lines of comments in 76 methods. It was clear from the source code that the class implements multiple unrelated abstractions and could be restructured into several more cohesive classes.

The final two metrics consider software quality. In the first, Stein et al. extend quality metrics to the design phase. Experimentation demonstrates that the metric values for a human-generated design specification predict the same quality as the code-based metric values for the source code written from those design documents [23]. The approach is based on *semantic metrics*, calculated by first extracting class names and the relevant paragraphs from an IEEE formatted design document. Then, using the CLIPS expert system, a knowledge-based hierarchical semantic net is created. The net has an interface layer of keywords tagged with part-of-speech information and internal nodes that capture higher level concepts with greater distance from the interface layer. Empirical validation of this approach is presently underway.

The final IR-based metric, the QALP score, is aimed at supporting human-

leveraged software assessment; that is, it uses IR’s language processing techniques to facilitate the assessment of *software in the large* (*i.e.*, in situations where a brute force approach is infeasible and for which automated techniques have failed to capture human intuition [24]).

For example, one of the techniques proposed involves module rating. This metric begins by building a corpus where each module is split into two artifacts: one includes the soft words extracted from the source code and the other the header and inline comments. The comments are stemmed to remove suffixes and stopped using a standard English stop-list. The code is then stopped using a programming language-specific stop-list that includes, for example, language keywords. Next the cosine similarity between source code and corresponding comments is computed.

A correlation between QALP score and quality allows it to be used to partition modules such that two modules in the same partition would receive similar assessment if shown to an engineer. An empirical study demonstrated this correlation. Figure 1 shows the results using a ranked list of pairwise differences. The data is sorted based on the magnitude of the difference in tool scores. Then, the average number of agreements in the top k entries yields percent precision through k . As expected, the result shows agreement for “larger” differences and then, as the magnitude of the difference approaches zero, agreement becomes more random. The fourth column includes “yes” iff the tool and the survey agree on the relative ranking of the two programs. If assignment were random, then the probability of agreement would be 50%. The perfect assignment yields 100% agreement. The penultimate column gives the percent precision. The final column provides the p score comparing this percent with a tool that made random quality assignments. As the underlying distribution satisfies the normality test, the student’s t -test was used to test the significance of each row.

Programs Compared	Differences		Agreement		p (Mean > 0.5)
	Tool Score	Survey Rating	Survey - Tool	Precision	
p3 - p5	-0.38	-1.93	yes	100%	-
p5 - p6	0.37	0.33	yes	100%	-
p1 - p5	-0.34	-0.73	yes	100%	-
p2 - p5	-0.26	-0.80	yes	100%	-
p3 - p4	-0.24	-1.13	yes	100%	-
p4 - p6	0.23	0.33	yes	100%	-
p1 - p4	-0.20	0.07	no	86%	0.023
p4 - p5	-0.14	-0.80	yes	88%	0.010
p2 - p3	0.13	1.13	yes	89%	0.004
p2 - p4	-0.12	0.00	no	80%	0.026
p2 - p6	0.11	-0.47	no	73%	0.069
p1 - p2	-0.08	0.07	no	67%	0.133
p1 - p3	0.04	1.20	yes	69%	0.087
p1 - p6	0.03	-0.40	no	64%	0.151
p3 - p6	-0.01	-1.60	yes	67%	0.104

Figure 1: Tool - Human Agreement (sorted by tool score difference).

The expected pattern is present in the data where, for differences of at least 0.23 in tool scores, there is 100% agreement between the tool and human judgment. Statistically, for a difference of at least 0.12 in tool score (first underline) the percent precision is significantly different from random ($p < 0.05$). Even for differences as low as 0.04, there is weak statistical evidence ($p < 0.10$) of non-randomness (second underline).

9 FUTURE DIRECTIONS

The initial success that IR techniques have found in SE is a testament to the wealth of information stored in the natural language of a program and its supporting artifacts. In the future, exploitation of this information should increase the value it brings to software engineering tools.

Such exploration might begin by considering more recent IR techniques. At present there is a great divide between the retrieval methods used in IR and those applied in SE. For instance, while the IR community has been focused on

language models and probabilistic approaches to retrieval over the past decade, the application of IR to SE has remained fixated on the use of LSI. This is unfortunate, as one of LSI's limitations is the often incorrect assumption that an artifact is generated from a single topic [25].

One potential reason for this divergence is the difference in size of the collections. In SE, a collection is typically a single program, and is thus orders of magnitude smaller than the typical IR collection (which can consist of terabytes of data). LSI has been validated on small collections. Nevertheless, the wide application of LSI in SE may be because it does not require stemming [26]. However, this may not be a valid rationale as the IR community has produced stemmers for many of the most common natural languages (ranging from English and Italian to Arabic). Recent work in SE supports this observation. For example, de Lucia et al. showed that there is no pay-off for the added complexity of LSI over VSM as long as one effectively used stemming and stopping [27]. Furthermore, Canfora et al. have found that VSM is particularly well-suited to representing software changes [28].

Looking forward, IR researchers continue to develop new methods for retrieval. SE researchers should periodically experiment with these new methods to see if the advantages observed on large collections hold for the smaller ones encountered in SE. For example, one recent development that may prove useful is *Latent Dirichlet Allocation*, a replacement for LSI that allows artifacts to be generated from a mixture of topic models rather than a single topic [25].

Another curious perspective of SE researchers is a fixation on the scalability of techniques developed in the IR community. Considering that IR techniques are designed to work with far larger collections than standard SE collections (*i.e.*, the collection of web pages on the internet compared to the artifacts associated with any given application), the scalability of IR techniques into the range of

sizes considered in SE should not be a significant concern. In addition, retrieval index information is designed to be built incrementally and in parallel, which further ameliorates build-time concerns. Although such improvements may not be implemented by researchers in proofs-of-concept tools, it does mean that such tools can easily scale if these algorithms were suitably modified.

Finally, there are many other techniques developed in the IR community that are related to natural language but not strictly retrieval, such as techniques for identifying *topic words* in text. These techniques deserve attention as they may provide fruitful avenues for future SE research.

10 SUMMARY

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 article 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 article 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 traceability 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.

11 ACKNOWLEDGMENTS

The expertise of Craig Allen was invaluable in preparing this survey. Special thanks to all the researchers who responded to the solicitation for current work and ideas for future research. This work is supported by National Science Foundation grant CCR0305330.

References

- [1] D. Binkley and D. Lawrie. Applications of information retrieval to software maintenance and engineering. *Encyclopedia of Software Engineering*, V(N), January 2008.
- [2] G. Antoniol, Y.G Guéhèneuc, and P. Tonella. Finding the balance: Three uses of information retrieval techniques in software engineering. In *Working Session: Information Retrieval Based Approaches in Software Evolution*, Nicosia, Cyprus, September 2006.
- [3] G. Salton and M. McGill. *Introduction to modern information retrieval*. McGraw-Hill Book Company, 1983.
- [4] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3), 1980.

-
- [5] R. Krovetz. Viewing morphology as an inference process. In R. Korfhage et al., editor, *Proceedings of the 16th ACM SIGIR Conference*, June 1993.
 - [6] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*, Dallas, TX, November 2006.
 - [7] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Proceedings of 2007 IEEE Conference on Source Code Analysis and Manipulation (SCAM'07)*, Paris, France, September 2007.
 - [8] S.W. Lee and D. Rine. Missing requirements and relationship discovery through proxy viewpoints model. In *Proceedings of ACM Symposium on Applied Computing*, Nicosia, Cyprus, March 2004.
 - [9] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8), August 1998.
 - [10] H. Mili, E. Ah-ki, R. Godin, and H. Mcheick. An experiment in software component retrieval. *Information and Software Technology*, 45, 2003.
 - [11] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artefact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.
 - [12] E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), October 2002.

-
- [13] J. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proceedings of the 11th International Requirements Engineering Conference*, Monterey, CA, USA, September 2003.
- [14] M. Lormans and A van Deursen. Can lsi help reconstructing requirements traceability in design and test? In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Philadelphia, USA, September 2006.
- [15] S. Tangsripairoj and M.H. Samadzadeh. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In *Proceedings of the 2005 Symposium on Applied Computing*, Santa Fe, NM, March 2005.
- [16] H. Yao, L. Etzkorn, and S. Virani. Automated classification and retrieval of reusable software components. *Journal of the American Society for Information Science and Technology*, to appear, 2007.
- [17] Murray Wood and Ian Sommerville. An information retrieval system for software components. *SIGIR Forum*, 22(3-4), 1988.
- [18] E. Stierna and N. Rowe. Applying information-retrieval methods to software reuse: A case study. *Information Processing and Management*, 39(1), January 2003.
- [19] T. Gyimóthy, R. Ferenc, and I. Siket. Emperical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10), October 2005.
- [20] T. Menzies, J. Greenwald, and A. Fransk. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), January 2007.

-
- [21] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, Philadelphia, USA, September 2005.
- [22] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of 2006 IEEE International Conference on Software Maintenance*, Budapest, Hungary, September 2005.
- [23] C. Stein, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani. Semantic software metrics computed from natural language design specifications. *IET Software*, to appear, 2007.
- [24] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *14th International Conference on Program Comprehension*, 2006.
- [25] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *Proceedings of the Twenty-Ninth International ACM Conference on Research and Development in Information Retrieval*, Seattle, WA, August 2006.
- [26] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Chicago, IL, USA, September 2004.
- [27] A. De Lucia, R. Oliveto, and P. Sgueglia. Can information retrieval techniques effectively support traceability link recovery? In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Philadelphia, USA, September 2006.

-
- [28] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Minneapolis, May 2007.