Empir Software Eng
DOI 10.1007/s10664-008-9095-3

An empirical analysis of information retrieval based concept location techniques in software comprehension

Brendan Cleary · Chris Exton · Jim Buckley · Michael English

Editors: Tim Menzies and Letha Etzkorn

Abstract Concept location, the problem of associating human oriented concepts with their counterpart solution domain concepts, is a fundamental problem that lies at the heart of software comprehension. Recent research has attempted to alleviate the impact of the concept location problem through the application of methods drawn from the information retrieval (IR) community. Here we present a new approach based on a complimentary IR method which also has a sound basis in cognitive theory. We compare our approach to related work through an experiment and present our conclusions. This research adapts and expands upon existing language modelling frameworks in IR for use in concept location, in software systems. In doing so it is novel in that it leverages implicit information available in system documentation. Surprisingly, empirical evaluation of this approach showed little performance benefit overall and several possible explanations are forwarded for this finding.

Keywords Information retrieval · Software comprehension · Empirical analysis

1 Introduction

Software comprehension is widely recognised as one of the most pervasive problems of software engineering (Rajlich and Wilde 2002; Littman et al. 1986; Marcus et al. 2003; Knight and Munro 2002). In the maintenance phase alone, often cited as the most costly

B. Cleary (✉) · C. Exton · J. Buckley · M. English
University of Limerick, Limerick, Ireland
e-mail: brendan.cleary@ul.ie

C. Exton
e-mail: chris.exton@ul.ie

J. Buckley
e-mail: jim.buckley@ul.ie

M. English
e-mail: michael.english@ul.ie

software engineering activity, it is estimated that software engineers spend more than half of their time occupied with comprehension tasks (Wilde et al. 2001; Zayour and Lethbridge 2001). If we take software maintenance not as a distinct phase but rather as an activity engaged in throughout the entire software lifecycle (Schneidewind et al. 1999) then the software comprehension problem can be seen as permeating all aspects of a software system's existence from development through to maintenance and evolution.

While software engineers' understanding or comprehension of the systems they work with can be described in terms of the comprehension theories previously cited, it can also be described in terms of a person's ability to communicate intelligently in human oriented terms about a system's implementation. For example Biggerstaff et al. (1994) describes a person as understanding a program when "able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program". This categorization of how an engineer understands a software system rests on two different descriptions of "computational intent" (Simonyi 2005) and the ability of the engineer to associate concepts appearing in one description of intent with the concepts in another.

In software engineering we are usually concerned with two descriptions of intent: one described using a human language and another using a programming language. For clarity and to compare related work, we refer to the former as problem domain descriptions of intent and the latter as solution domain descriptions of intent. These different descriptions or domains of intent are separated by constraints on the sets of concepts expressible using the language in which they are described which constitute a "conceptual gap" between domains (Rajlich and Wilde 2002). Biggerstaff et al. (1993) describes the difference between the domains as follows; "the first case [problem domain] expresses computational intent in human orientated terms, terms that live in a rich context of knowledge about the world. In the second case [solution domain], the vocabulary and grammar are narrowly restricted, formally controlled and do not inherently reference the human orientated context of knowledge about the world". The problem of creating a mapping between the problem and solution domains, which according to Biggerstaff et al. is at the heart of software comprehension, is termed the concept location problem.

Recent research (Marcus and Maletic 2003; Marcus et al. 2004; Zhao et al. 2004; Poshyvanyk et al. 2006a, b) has seen several attempts to bridge the conceptual gap that lies at the core of the concept assignment problem based on the application of methods drawn from the information retrieval (IR) community. The concept location problem put simply is the problem of identifying the subset of elements comprising a software system related to a set of problem domain concepts. These information retrieval based concept location approaches typically index the source code "documents" of a system under study constructing a vocabulary of terms and models of term–document relationships. When a user specifies a query the vocabulary and term–document models are consulted in order to determine a set of documents (usually methods or functions) which are ranked in descending order and returned to the user as being related to the terms defined in their original (problem domain) query.

Approaches which follow this general schema tend to focus on information derivable solely from the source code of the system in constructing their models and ranking documents as being related to a query. While source code is one artefact used by software engineers to express their intent, engineers have traditionally had recourse to use other artefacts to communicate and record concerns which either were not easily expressed directly in code (crosscutting concerns for example; Kiczales et al. 1997; IEEE 2000; Greenfield and Short 2004) or which they intended to express in code at some other time.

Artefacts such as design and requirements documentation have traditionally been used to express such concerns while more recently bug tracking databases, online forums and even email have become in some development environments the primary mechanisms for expressing and communicating such concerns (Cubranic et al. 2005).

These non-source code artefacts serve as a "human oriented" repository of information related to a system under study that parallels the source code "machine orientated" artefacts of the system. By concentrating on source code artefacts alone the concepts and concerns described in these artefacts cannot be taken into account when evaluating a user's query. In this paper we present a new approach to the concept assignment problem based on a cognitively motivated information retrieval technique which we term cognitive assignment. This technique while similar to the general information retrieval based concept location schema outline above differs in that it allows us to transparently incorporate information derived from non-source code artefacts in implementing a ranked search over the source code artefacts of a system. In this paper we will also present one of the largest concept location experiments conducted to date to compare the performance of some of the most popular information retrieval based concept location techniques in terms of average precision. An analysis of the findings of this experiment will show that the cognitive assignment technique is capable of outperforming existing concept location techniques based on similar information retrieval formalisms. However, in this paper we also present results from the same experiment which identifies a simpler and less computationally expensive information retrieval based concept location technique which is able to outperform other concept location techniques based on more complex and expensive formalisms.

In the next section, Section 2, we present a critical literature review of existing software comprehension and concept location approaches. In Section 3 we present the formal definition of the novel cognitive assignment technique. In Sections 4 and 5 we present an analysis of the performance of cognitive assignment technique in comparison with other competing techniques through an experiment. Finally in Section 6 we present our conclusions and discuss future work.

2 Related Work

The concept location problem put simply is the problem of identifying the subset of elements comprising a software system related to a set of problem domain concepts. As such the concept location problem is very similar in intent to the concern localization and feature location problems in that each is concerned with mapping from the problem domain to the solution domain. However, while the three terms are frequently used interchangeably we can envisage a distinction in terms of the constraints placed (usually informally) on the definition of the set of problem domain concepts in each. While concept location places very little constraint on its definition of a problem domain concept (except that it be defined using human-orientated terms), concern localization is motivated by problem domain concepts that are also system stakeholder interests. Finally feature location goes one step further in constraining problem domain concepts to be those stakeholder concerns which are also observable functional requirements and executable using test cases.

2.1 Feature Location and Concern Localization

Feature location approaches, due to the more stringent constraints they place on the definition of the problem domain concept set, are able to capitalize on the formal

relationships expressed in a function call trace that results from the execution of one or more test cases related to a feature or features. Dynamic software analysis or feature location techniques such as software reconnaissance (Wilde and Scully 1995) and formal concept analysis (Eisenbarth et al. 2003), focus on localizing concepts that are expressible either through test cases or through navigation of control and data flow. Unfortunately while a system's implementation may imply the intent that led to its development, the intent is not expressed explicitly in that implementation (Greenfield and Short 2004). As such these techniques are only able to localize concepts which are expressible as test cases. While this is a limitation, in cases where there exists no system expert or documentation, they can be of great benefit in assisting software engineer comprehension. These approaches are also able, given enough test cases, to identify sets of elements specific to particular features.

Concern localization approaches, due to their definition of problem domain concepts as being that which a system stakeholder is concerned with, tend to involve those stakeholders (usually software engineers) and their existing knowledge or actions in defining the mapping between concerns and software elements. Semi-automated or manual approaches such as IBM's concern manipulation environment (CME; Chung et al. 2005) and FEAT (Robillard 2003) provide an environment which allows engineers to explicitly describe and record associations between software elements and user defined concerns. Similarly artefact recommender systems or agents such as Hipikat (Cubranic et al. 2005) suggest pertinent artefacts (both source code and documentation) to engineers as they engage in an understanding task by attempting to automatically model the concerns the engineer is currently working on and making inferences from that model. Mylar (Murphy et al. 2006) is another tool that attempts to model the concerns or tasks that an engineer is concerned with as a set of evolving software elements; this information is then used to perform filtering of the elements presented to the engineer in an IDE on a task by task basis in an attempt to reduce information overload.

Dora (Hill et al. 2007) takes a natural language query and a user specified starting point or seed within the source code to generate a "relevant neighbourhood" or subsection of the systems call graph which is most likely to contain the code of interest. To do this it uses structural information embedded within the source code (call graph) to rank and prune irrelevant code from further consideration by postulating that source code that is structurally distant from the current area of interest is less likely to be of relevance. Find-Concept (Shepherd et al. 2007) uses a semi-automated technique to expand the initial query into a more effective query, by requiring the user to interact and refine the search according to a number of different possible search paths that are presented by the tool. This is initially achieved by restricting the user to action oriented queries, which allows the system to take advantage of simple English grammatical concepts such as objects and verbs to formulate a number of related possible terms, which are presented to the user, who then is required to select those which are most seminal to their enquiry. The possible terms are derived from a set of commonly occurring concepts which are identified from related documents such as bug reports.

Canfora and Cerulo (2005, 2006), in a related piece of research, explore the possibility of using IR techniques on CVS and Bugzilla repositories for impact analysis. For each new change request encountered (CR), they compared it textually to other, already complete, CRs and their associated 'work files', based on the comparison algorithm in (Jones et al. 2000). Their work has similarities to this work in that the authors use system documentation as a basis for concept location, and use existing CVS records as the basis for measuring precision and recall. However, unlike this work, they do not expand the terms in the original query.

2.2 IR Based Approaches

The concept location problem being very much more liberal in its definition of a problem domain concept requires approaches which are able to construct relationships between arbitrary problem domain concepts described in human orientated terms and the elements of a software system from only that evidence which already exists in the corpus of software system artefacts. When described like this the concept location problem would seem well fitted by techniques from the information retrieval community where the goal is to find material (usually documents) of an unstructured nature (usually text) that satisfy an information need from within large collections (Manning et al. 2007). Zhao et al. (2004) describe their attempts at using an IR technique, the vector space model (VSM), in identifying specific and relevant sets of functions for a set of given feature descriptions. Their approach sees them construct feature descriptions from natural language texts such as requirements and design documentation. These feature documents are then matched, using the VSM, against query documents derived from identifiers found in functions in the source code of the system under study. The resulting ranked set of feature–function pairs is then used to describe an initial specific function set. In order to improve the accuracy of this initial specific function set and to derive a relevant function set the authors use the initial specific function set to direct a traversal of an augmented call graph they term a branch-reversing call graph (BRCG). This traversal yields the set of relevant functions for the feature set from which the final specific set of functions is then identified.

One of the fundamental problems associated with the VSM, as used by Zhao et al. (2004), is that correlation between term sets is used to compute the similarity measure between documents or between documents and queries, That is, two documents are considered similar if they share the same terms. While logical, this scheme requires that if documents are discussing a particular concept, then to be considered similar those documents need to use the same terms when describing the concept. Where a document or a user generated query uses different terms when referring to the same concept then documents that should be considered similar will likely not be classified as such by VSM. These problems are termed synonymy and polysemy, respectively, where synonymy describes the problem of different terms being used to describe the same concept and polysemy describes the problem of a single term (depending on context) having more than one distinct meaning (Deerwester et al. 1990). This is of potentially great significance to novice software engineers or engineers encountering an unfamiliar system for the first time that may not possess a large vocabulary of terms with which to describe their concerns and so may be unsuccessful in constructing queries which would elicit the desired results.

Latent semantic indexing (LSI) is an extension to the VSM model that attempts to increase classification accuracy by reducing the impact of the synonymy and polysemy problems. Whereas VSM computes similarity based on a raw term-document occurrence matrix (with possibly some term weighting) LSI attempts to alleviate the problems we have just described by performing an extra analysis step in which the overall distribution of a term over its usage context, independent of its correlations with other terms, is first taken into account prior to computing a similarity measure between documents (Landauer et al. 1998). This allows otherwise unrecognized or "latent" relationships between terms to be exposed and recognized when performing a search. Marcus et al. (2004) expand on the work in Marcus and Maletic (2003) to apply the LSI method directly to the concept location problem. The authors first index the source code of the system under study identifying and extracting index terms from identifiers and comments. The authors then partition the system under study into a set of documents (defined in terms of the index term set) corresponding

to the set of functions defined in the system under study. Finally each document is mapped, using a technique called singular value decomposition, into the LSI space. Query documents are similarly mapped into the LSI space and the similarity between documents and the query is computed in a similar fashion as in the VSM. In an attempt to further resolve the lack of vocabulary problem described previously the authors investigate two mechanisms for defining the query documents, the first is a simple user generated natural query. The second sees the authors implement a limited form of query expansion where given a single query term other potentially related query terms are derived from the LSI space based on their relationship to the "seed" query term.

While using the LSI space to identify latent relationships between terms in the source code of a system under study will identify some previously unrecognized relations, it is only making use of one part of the available corpus of information about the system under study. The work presented in this paper similarly to Marcus et al. (2004) employs a sophisticated IR method that seeks to address the problems of synonymy, polysemy and lack of vocabulary mentioned previously. However, unlike Marcus et al. (2004) the method we describe provides a structured and cognitively motivated mechanism for incorporating information derived from the potentially large (depending on the system under study) corpus of non-source code artefacts in generating sets of ranked software elements in response to a user's query. The next section describes our approach, its cognitive motivation and implementation.

3 Assisting Concept Location Through Language Modelling

Language modelling (LM) is an approach used in many recent studies in IR that not only produces promising experimental results comparable with the best IR systems but also provides a sound theoretical setting (Bai et al. 2005). The LM approach to information retrieval calculates the conditional probability $P(Q|D)$ of generating a query Q given an observed document D, where $P(Q|D)$ is calculated based on a probabilistic language model derived from document D.

3.1 Classical LM Framework

The classical LM framework can be described as follows; given a query Q consisting of a sequence of query terms $Q=\{q_1,q_2,\ldots q_n\}$ the probability of generating Q from document D is equal to the probability of observing the sequence of n query terms from Q in D.

$$P(Q|D) = P(q_1, q_2, ..., q_n|D).$$

For reasons of computational tractability the independence assumption which states that terms are statistically independent from each other (Gao et al. 2004) is often invoked. This assumption results in a unigram model being calculated from D whereby terms are considered to be conditionally independent. This means that the order or sequence of occurrence of query terms in the document does not need to be considered when calculating the correspondence of the query and document.

$$P(Q|D) = \prod_{q_i \in Q} P(q_i|D).$$

Models such as this have already been used by for recovering traceability links between source code and high-level documentation. Antoniol et al. (2002) used unigram estimation based on term frequency to create links that describe the similarity between elements of the code base (object-orientated classes) and high level system documentation. Antoniol et al. use a stochastic language model based on identifiers found in the source code elements to calculate the set of conditional probabilities between a given source code element and the set of system documents.

An alternative formulation of the classic language modelling framework is Kullback Leibler-divergence (KLD) which estimates two models: one for the document M_{D} but also one for the query M_{Q}, the similarity of the query Q and document d_i then being determined by the divergence between the two models:

$$-KL\left(M_Q \middle| M_D\right) = -\sum_{t \in V} M_t^Q \log \frac{M_t^Q}{M_t^D} \tag{1}$$

Equation 1—Kullback Leibler-Divergence

which is $\overset{RANK}{\propto}$ (rank proportional) to the similarity function Similarity (Q, d_i)

$$\mathrm{Similarity}(\mathrm{Q}, \mathrm{d_i}) = \sum_{\mathrm{t_i} \in \mathrm{V}} \mathrm{P}(\mathrm{t_i}|\mathrm{Q}) \times \log \mathrm{P}(\mathrm{t_i}|\mathrm{d_i}). \tag{2}$$

Equation 2—LM Based Similarity Function

Given that $P(t_i|Q)$ is usually determined without smoothing using maximum likelihood estimation based on term frequency tf, where $P(t_i|Q) = tf(t_i, Q) \Big/ \sum_{t_i \in V} tf(t_1, Q)$. We can ignore the non query terms from the vocabulary in evaluating the similarity function.

$$\mathrm{Similarity}\,(\mathrm{Q}, \mathrm{d_i}) = \sum_{\mathrm{q_i} \in \mathrm{Q}} \mathrm{P}(\mathrm{q_i}|\mathrm{Q}) \times \log \mathrm{P}(\mathrm{q_i}|\mathrm{d_i}).$$

To demonstrate how this similarity function can be applied to concept location in source code, we present the following example. The source code document presented in Fig. 1 is the "queryDocumentNames" method taken from the Index class in the org.eclipse.jdt.internal.core.index package. This method, which is part of the search component of the Eclipse java development toolkit, is used to implement a search feature which returns the names of documents from the index which contains the specified substring or query, basically a simple filename search.

If a human were asked to describe this functionality they might reasonably consider a query such as "document, name, search" as being representative of the concept implemented by the source code document presented in Fig. 1. If we were to attempt to calculate the similarity of the document to this query, a classic language modelling approach such as KLD (without smoothing) would calculate the similarity between the document and the query based on the direct occurrence of terms from the query in the document.

In Fig. 2 we see the terms "document" and "name" highlighted in the document, the probabilities of these two terms occurring in the document then being used to calculate the similarity of the query and document. Note that the term "search" while part of the original query did not occur in the document and so cannot be included in the similarity calculation. Like the LM framework used by Antoniol et al. described earlier, this kind of similarity

```
/**
 * Returns the document names that contain the given substring, if null
then returns all of them.
 */
public String[] queryDocumentNames(String substring) throws IOException {
      SimpleSet results;
      if (this.memoryIndex.hasChanged()) {
            results = this.diskIndex.addDocumentNames(substring,
this.memoryIndex);
            this.memoryIndex.addDocumentNames(substring, results);
      } else {
            results = this.diskIndex.addDocumentNames(substring, null);
      }
      if (results.elementSize == 0) return null;

      String[] documentNames= new String[results.elementSize];
      int count = 0;
      Object[] paths = results.values;
      for (int i = 0, l = paths.length; i < l; i++)
            if (paths[i] != null)
                  documentNames[count++] = (String) paths[i];
      return documentNames;
}
```

Fig. 1 Example source code document

function also uses a unigram model based on the term independence assumption to estimate term probabilities in the document. While the independence assumption makes the development of retrieval models easier and the retrieval operation tractable, it does not hold in textual data (Gao et al. 2004). In reality a word may be related to other words (Bai et al. 2005). As such the unigram model has the potential to miss potentially significant dependencies between terms; for instance the synonymy relationship. This deficiency has prompted research into extending the classical LM framework.

```
/**
 * Returns the document names that contain the given substring, if null
then returns all of them.
 */
public String[] queryDocumentNames(String substring) throws IOException {
      SimpleSet results;
      if (this.memoryIndex.hasChanged()) {
            results = this.diskIndex.addDocumentNames(substring,
this.memoryIndex);
            this.memoryIndex.addDocumentNames(substring, results);
      } else {
            results = this.diskIndex.addDocumentNames(substring, null);
      }
      if (results.elementSize == 0) return null;

      String[] documentNames= new String[results.elementSize];
      int count = 0;
      Object[] paths = results.values;
      for (int i = 0, l = paths.length; i < l; i++)
            if (paths[i] != null)
                  documentNames[count++] = (String) paths[i];
      return documentNames;
}
```

Fig. 2 Classical LM approach

3.2 Extending the Classical LM framework

One avenue researchers have pursued to extending the LM framework is incorporating term dependencies back into the language model, thus relaxing the independence assumption of the unigram model while simultaneously attempting to preserve its desirable computational characteristics. In these approaches for a document to be matched with a query it not only would have to contain the query terms but also demonstrate the term dependencies required by the query. These approaches range from simply considering bi-gram or more generally *n*-gram dependencies to more complex dependency models. Gao et al. (2004) present a dependency language model for information retrieval in which documents are not considered simply as bags of words but rather as sequences of terms over which they define "linkages" using dependency parsing and learning techniques. Their goal is to identify an optimal set of linkages over the sequence of terms rather than consider every possible dependency which would result in degraded performance. We have investigated a similar approach to the concept location problem previously (Cleary and Exton 2006a, b).

Another approach to extending the classic LM framework, and the one which we investigate in this paper, is to consider indirect correspondences between query terms and document terms so that documents can be retrieved even if they do not contain the original query terms. In these approaches a model of relationships between terms is first defined, then when two terms are compared if there is no direct correspondence this relationship model is consulted to see if an indirect match can be made. How the model of term relationships is implemented allows us to classify these extensions to the LM framework as practicing either document expansion or query expansion.

Document expansion approaches (Gao et al. 2004) use the relationship model to enrich the document model so that terms in the relationship model which have related terms in the document are artificially inserted into the document model. In these approaches then matching of a query to a document proceeds as normal except that terms not originally in the document will now also be considered in the evaluation. Query expansion approaches (Bai et al. 2005; investigated in this paper) do not manipulate the document model but instead modify the query model, identifying terms in the relationship model that are related to the query terms and then considering these additional terms when evaluating the query against document models.

3.3 Query Expansion with Term Relationships

More formally we can define query expansion in the LM framework as an extension to classical smoothing techniques. In the classical LM framework for a document to be retrieved given a query that document would have to contain all the query terms. If only one term from the query were not present in the document then the document would not be determined as being similar to the query, this is termed the zero probability problem. In order to allow documents which contain only some of the query terms to be retrieved the document model or query model can be smoothed in terms of the collection model so that the probabilities of terms not actually in the document or query are increased to some small non-zero value.

Unfortunately this form of smoothing while solving the zero probability problem increases the probabilities of all the non-occurring terms in the document uniformly or proportionally to the term distribution in the whole collection (Bai et al. 2005). This means that terms which are actually not related to those already in the document are artificially incorporated into the document model (for example), while those terms that deserve to be incorporated in the model (by their relatedness to terms in the document) receive no special treatment. For

example if the term "engine" appears in a document the probability that a query with the term "car" should be considered similar to that document is higher than if the query contained the term "desk" instead. In this case it is intuitively more reasonable to assign a higher smoothing value to the term "car" due to the relationship between "engine" and "car".

An alternative solution to smoothing the query model is to incorporate terms derived from some explicit model of term relationships. Bai et al. (2005) smooth the original query model $P_{ML}(t_i|Q) = P(t_i|Q)$ by another probability function defined over an explicit model of term relationships $P_{\mathrm{R}}(t_i|Q)$ so that the query model from (2), $P(t_i|Q)$ now becomes:

$$P(t_i|Q) = \lambda P_{ML}(t_i|Q) + (1-\lambda)P_R(t_i|Q)$$

where λ is a mixture parameter which allows us to regulate the influence of the different probability estimations on the calculation of $P(t_i|Q)$. Given this definition for $P(t_i|Q)$ we have the following:

$$\begin{aligned}\mathrm{Similarity}(Q,d_i) &= \sum_{t_i \in V} P(t_i|Q) \times \log P(t_i|d_i) \\ &= \sum_{t_i \in V} [\lambda P_{\mathrm{ML}}(t_i|Q) + (1-\lambda)P_{\mathrm{R}}(t_i|Q)] \times \log P(t_i|d_i) \\ &= \lambda \sum_{q_i \in Q} P_{\mathrm{ML}}(q_i|Q) \times \log P(q_i|d_i) + (1-\lambda)\sum_{t_i \in V} P_{\mathrm{R}}(t_i|Q) \times \log P(t_i|d_i)\end{aligned} \tag{3}$$

Equation 3—Query Expansion Similarity Function

As terms not in the query will have a probability of zero for $P_{\mathrm{ML}}(t_i|Q)$ we can ignore them and redefine the first sum in the equation over the query terms rather than the whole vocabulary (Bai et al. 2005). However, the second sum in the equation that depends on the model of term relationships $P_{\mathrm{R}}(t_i|Q)$ is still defined over the entire vocabulary.

3.4 Hyperspace Analogue to Language

A human encountering a new concept derives its meaning via an accumulation of experience of the contexts in which the concept appears (Bruza and Song 2002). Hyperspace analogue to language (HAL) is a cognitively motivated and validated semantic space model for deriving term co-occurrence relationships from a corpus of text (Lund and Burgess 1997). HAL is significant because the term associations computed by the HAL model correlate with human judgments in word association tasks (Bruza and Song 2002).

HAL represents words/terms/concepts as vectors in a high dimensional space based on lexical co-occurrences. A simple windowing based co-occurrence analysis can be used to construct a HAL space, whereby a window of size *l*-words is passed in one word increments over the corpus of text. Where two words occur within the window a co-occurrence relationship is defined between them. For an *n*-word vocabulary this co-occurrence analysis results in an $n*n$ matrix of co-occurrence relationships. For example; given a very simple "document" consisting of a sequence of characters (a b b c a b e d d a a c a b a a b e a c b a c). We can derive the co-occurrence matrix (HAL space) presented in Table 1 by passing a window of 4 characters in length over the "document" from left to right one term at a time and counting the number of times each term from the vocabulary (a, b,c,d,e) occurs with another term from the vocabulary within the window.

This example shows us that the term pair (b,c) co-occurred three times in the "document" and the term pair (e,d) co-occurred two times. The example co-occurrence

Table 1 Simple co-occurrence matrix (HAL Space)

	a	b	c	e	d
a	8	8	4	3	1
b	5	3	3	2	2
c	4	3	1	1	0
e	2	1	1	0	2
d	4	0	1	0	1

matrix presented in Table 1 is a unidirectional co-occurrence matrix meaning that the order in which terms co-occur is preserved, that is, the term pair (a,b) is not equivalent to the term pair (b,a).

To illustrate how a HAL space can be computed from a more complex document we will construct a simple co-occurrence matrix from the first paragraph of this section. Constructing a co-occurrence matrix from a document first requires the definition of a suitable window size and for this example we have selected a window size of 9. Next we apply a stop list which removes unwanted (typically uninformative) terms from the document. This normally includes numbers and non-alphanumeric characters. If we define the following stop list; (a, its, via, an, of, the, in, which, to, is, and, for, from, because, by, with) and apply it to the document we get the document presented in (Fig. 3).

Next we proceed to pass the window over the document one term at a time constructing a unidirectional co-occurrence matrix by counting the number of times terms co-occur within the window. Unfortunately the reality is that most co-occurrence matrices even for a very simple example like this are large sparse matrices which can be very difficult to reproduce in print. However to demonstrate the construction of a co-occurrence matrix we show *an excerpt* of the matrix constructed from the document described in Fig. 3 for a reduced vocabulary consisting of four terms (human, concept, HAL, term) in Table 2.

You can see from this unidirectional co-occurrence matrix that the term pair (human, concept) co-occurs once within the example document while the term pair (HAL, human) co-occur twice given the widow size of 9 defined earlier. For more information building co-occurrence matrices and HAL spaces, we refer the reader to (Lund and Burgess 1996).

Given a co-occurrence matrix such as that in Table 2, a concept c_i then can be represented as a vector drawn from this matrix $c_i = \langle w_{c_i p_1}, w_{c_i p_2}, \ldots, w_{c_i p_n} \rangle$ where p_1,p_2,…, p_n are called dimensions of c_i and correspond to the other concepts/words from the vocabulary which c_i participates in a co-occurrence relationship with. $w_{c_i p_1}$ is then the weight of p_i in the c_i concept vector (Bruza and Song 2002). An example HAL vector for the term "HAL" derived from the full co-occurrence matrix we computed for the first paragraph of this section is given in Fig. 4.

Given a HAL vector representation for a concept c_i, a set of quality properties $QP(c_i)$ for that concept can be derived. Quality properties are those properties of the concept which frequently co-occur in the same context as the concept. A property p_i of a concept c_i is declared a quality property if $w_{c_i p_1} > \partial$, where the threshold ∂ is the mean weight for the concept vector c_i (Bruza and Song 2002).

Fig. 3 Document after stop word removal

human encountering new concept derives meaning accumulation experience contexts concept appears Bruza Song HAL Hyperspace Analogue Language cognitively motivated validated semantic space model deriving term co-occurrence relationships corpus text Lund Burgess HAL significant term associations computed HAL model correlate human judgments word association tasks Bruza Song

Table 2 Example co-occurrence matrix

	Human	Concept	HAL	Term
Human	0	1	0	0
Concept	0	1	1	0
HAL	2	0	1	1
Term	1	0	2	0

3.5 Information Flows

The co-occurrence matrix at the heart of the HAL and cognitive map representations can be used directly to make inferences about term relationships. However, Song and Bruza (2001) propose a more complex model of term relationships based on HAL vectors. The goal of the HAL-based information flow (IF) model is to produce information-based inferences which correlate with inferences made by humans (Song and Bruza 2003).

Given a source term or set of source concepts $c_1,\ldots,c_k$ and a target concept c_j there is an information flow from the set of source concepts to the target concept $c_1 \ldots, c_k| - c_j$ if the former suggest or entails the latter to some degree (Bai et al. 2005).

The degree of information flow degree $\left(\underset{1\le i\le k}{\oplus} c_i \triangleleft c_j\right)$ from $c_1,\ldots,c_k$ to c_j is given by:

$$\text{degree}\left(\underset{1\le i\le k}{\oplus} c_i \triangleleft c_j\right) = \frac{\sum_{p_l \in \left(QP\left(\underset{1\le i\le k}{\oplus} c_i\right) \wedge QP(c_j)\right)} w_{c_i p_l}}{\sum_{p_l \in QP\left(\underset{1\le i\le k}{\oplus} c_i\right)} w_{c_i p_l}} \tag{4}$$

Equation 4—Information Flow

where c_i denotes the HAL vector representation of concept c_i, $\underset{1\le i\le k}{\oplus} c_i$ denotes the combination of the individual concept vectors $c_1,\ldots,c_k$ and $w_{c_i p_l}$ represents the weight of property p_l in the vector of concept c_i.

Essentially information flow measures how many of the quality properties of the source vector are also properties of the target vector (Bai et al. 2005), that is, the ratio of the intersection of the set of quality properties of $\underset{1\le i\le k}{\oplus} c_i$ and c_j to the number of quality properties in $\underset{1\le i\le k}{\oplus} c_i$.

3.6 Query Expansion Using Information Flows

Using (4) as a measure of the degree of information flow between concepts, and given a concept or set of concepts in the form of a query we can compute information flow values for each term in the vocabulary and by imposing a threshold or by selecting a set of the top

HAL =<human:2, encountering:0, new:0, concept:0, derives:0, meaning:0, accumulation:0, experience:0, contexts:0, appears:0, Bruza:1, Song:0, HAL:1, Hyperspace:1, Analogue:1, Language:1, cognitively:1, motivated:1, validated:1, semantic:1, space:1, model:2, deriving:0, term:1, co-occurrence:0, relationships:0, corpus:0, text:0, Lund:0, Burgess:0, significant:1, associations:1, computed:1, correlate:2, judgments:1, word:1, association:1, tasks:1>

Fig. 4 Example HAL vector

ranked terms define a set of terms related by information flow to the terms in the query. That is we can use HAL-derived information flow to define $P_{\mathrm{R}}(t_i|Q)$.

More formally if we define information flow between terms as a probability:

$$P_{IF}(t_2|t_1) = \frac{\text{degree}(t_1 \triangleleft t_2)}{\sum_{t_k \in Vocabulary} \text{degree}(t_1 \triangleleft t_k)}$$

then we can define $P_{\mathrm{R}}(t_i|Q)$ as follows:

$$P_R(t_i|Q) = P_{IF}(t_i|Q) = \sum_{Q_j \subseteq Q} P_{IF}\left(t_i|Q_j\right) \times P\left(Q_j|Q\right)$$

where Q_j can be a single query term or a group of query terms but usually corresponding to the query itself and where $P\left(Q_j|Q\right) = \frac{1}{|Q|}$.

To limit the number of term relationships considered we can then define a set of the top ranked IF relationships E using some threshold and only consider those terms which are part of a relation in E.

$$P_R(t_i|Q) = P_{IF}(t_i|Q) = \sum_{Q_j \subseteq Q \wedge R\left(t_i, Q_j\right) \in E} P_{IF}\left(t_i|Q_j\right) \times P\left(Q_j|Q\right)$$

This definition of $P_R(t_i|Q)$ is then used to smooth our query model in Equation 3 (taken from (Bai et al. 2005)):

$$\begin{aligned} Similarity\,(Q, d_i) = {} & \lambda_{IF} \sum_{q_i \in Q} P_{ML}(q_i|Q) \times \log P(q_i|d_i) \\ & + (1 - \lambda_{IF}) \sum_{Q_j \subseteq Q \wedge R\left(t_i, Q_j\right) \in E} P_{IF}\left(t_i|Q_j\right) \times P\left(Q_j|Q\right) \times \log P(t_i|d_i) \end{aligned} \tag{5}$$

Equation 5—Information Flow Based Query Expansion

3.7 Cognitive Assignment

While source code is the primary artefact used by software engineers to express their intent, those same engineers have traditionally had recourse to use other artefacts to communicate and record concerns which were not easily expressed directly in code. While recognized for their potential importance in assisting software engineer comprehension of unfamiliar systems, it has not been immediately obvious how to make use of intent rich non-source code artefacts in tools or techniques designed to assist software comprehension.

Direct and explicit use of non-source code artefacts is only one way in which these artefacts can be used to assist software comprehension. These non-compliable development artefacts can also serve as a repository of term relationships specific to the particular system. The query expansion LM approach, based on the cognitively motivated HAL and IF representation of term relationships discussed in this section then serves as a principled foundation which allows us to incorporate these previously underutilized development artefacts in alleviating the concept location problem in software comprehension. The technique has also been experimentally shown to result in between 12% and 27%

improvement over the standard Kullback Leibler-divergence language modelling approach on which it is based (Bai et al. 2005).

Our cognitive assignment technique modifies the basic HAL and IF query expansion LM framework so that instead of calculating a term relationship model from the source code of a system under study, we generate a HAL space from the non-source code artefacts related to the system. This HAL space is then used in combination with IF analysis to generate sets of terms that are potentially related to query terms specified by the user which are then used to smooth the classic LM query model. That is we calculate the query model smoothing function $P_R(t_i|Q)$ not from the source code of the system but from other non-source code documentation artefacts related to the system. The hypothesis for doing this is that by using natural language non-source code artefacts, useful relationships between terms will be included in the term relationship model that may not be expressed in source code due to the unrestricted nature of natural language documents. Further, these extended relationships will result in an improved KLD technique which will perform better than existing information retrieval techniques, including the unmodified KLD technique.

To demonstrate how the cognitive assignment technique can be used to perform concept localisation in source code, we expand our example from Section 3.1 to show how the cognitive assignment technique assesses the similarity of the query and document by expanding the query to include related terms from a HAL space derived from a set of non source code documents related to d_i. HAL spaces used for information retrieval are typically large sparse matrices and so difficult to depict in print. For this example we present a contrived simple HAL space (Fig. 5) to help us illustrate our point (please note the quality properties of the individual concepts vectors are highlighted).

From this contrived HAL space we compute information flow values, using (4), between the query terms and each term in the vocabulary, see Fig. 6.

Based on this set of information flow values we then select the most relevant terms to expand the original query, in this example we select the term “query” as being related. This then gives us the expanded query; “document, name, search, query”.

The impact of the expansion step on the terms used to determine the similarity of the document and the query is depicted visually in Fig. 7. Here we see that the term “query” has been highlighted in the document along with the original query terms. This indicates that this term will be included in the calculation of the similarity function (5), producing a higher similarity score for this document compared to that which would be generated if query expansion had not been applied.

	document	name	search	query	index	string	contain	mean
document	2	**5**	2	**3**	2	1	**3**	2.5714
name	2	1	**4**	**3**	**3**	**4**	0	2.4286
search	**4**	2	**5**	**9**	**6**	**6**	2	4.8571
query	**4**	**6**	**8**	**6**	**6**	**6**	**5**	5.8571
index	**5**	2	**6**	**7**	2	2	**4**	4
string	**6**	0	**4**	**6**	**3**	**4**	**4**	3.8571
contain	2	0	2	1	**3**	2	2	1.7143

Fig. 5 Example HAL space

Fig. 6 Information flow values for vocabulary terms

Term	IF
document	0.483871
name	0.7096774
search	0.7096774
query	**0.9032258**
index	0.3870968
string	0.6451613
contain	0.6129032

4 Experiment Procedure

The following prerequisites are required to evaluate the performance of the cognitive assignment technique against that of a set of other IR based concept location techniques;

- A software system over which to perform the experiment (the system under study).
- A corpus of natural language documents related to the system under study from which to create a HAL space.
- A set of concepts related to the system under study.
- A set of query terms that accurately describe each concept.
- A set of relevant software elements for each concept.

The next sections discuss the definition of each of these requirements. The exact versions of the prerequisites used in this experiment such as document corpus and source code repository are available from the author on request. The implementation of the cognitive assignment technique is available online (Cleary 2007).

Fig. 7 Cognitive assignment approach

```
/**
 * Returns the document names that contain the given
substring, if null then returns all of them.
 */
public String[] queryDocumentNames(String substring)
throws IOException {
      SimpleSet results;
      if (this.memoryIndex.hasChanged()) {
            results =
this.diskIndex.addDocumentNames(substring,
this.memoryIndex);
            this.memoryIndex.addDocumentNames(substring,
results);
      } else {
            results =
this.diskIndex.addDocumentNames(substring, null);
      }
      if (results.elementSize == 0) return null;

      String[] documentNames= new
String[results.elementSize];
      int count = 0;
      Object[] paths = results.values;
      for (int i = 0, l = paths.length; i < l; i++)
            if (paths[i] != null)
                  documentNames[count++] = (String)
paths[i];
      return documentNames;
}
```

4.1 Experiment System

In choosing a system over which the experiment would be performed there were several concerns that influenced our decision. To ensure external validity of the experiment we were restricted to choosing a system which the experimenters had no involvement in developing. Secondly one of the objectives of this experiment was to analyse the generality of the cognitive assignment technique both in terms of the size of system to which we applied it and the number of concepts we attempted to localise. As such we looked for a system that was both non trivial in terms of code size (measured in KLOC) and also non trivial in terms of the number of concepts which we could attempt to localise. Finally we require a corpus of natural language documentation from which to initialise a HAL space related to the system. This corpus of documentation needs to fulfil many of the same criteria that the source code does, for instance like the source code it needs to be publicly available so that the experiment can be replicated by other researchers.

The Eclipse project is one of the most popular open source projects with the Eclipse java IDE being one of the most popular java IDEs in common use in both commercial and open source software development worldwide. As such it comfortably fulfils the criteria for a large and popular open source project and one which is independent from the experimenters. Being an open source project it is largely the result of the collaboration between a globally distributed set of software engineers who use email, newsgroups and bug tracking databases to communicate satisfying the need for a large natural language documentation corpus. As such the Eclipse project would seem an ideal candidate for information retrieval concept location experimentation.

The Java development tooling (JDT) component of the Eclipse project is responsible for providing the set of plug-ins required to implement the Eclipse Java IDE. It provides users with the ability to edit, compile, and debug programs written in the Java programming language. As such it is a fundamental component of the Eclipse project and is used by many thousands of software engineers every day. The JDT is split into several sub components. The core subcomponent (org.eclipse.jdt.core) of the JDT is responsible for providing the classes used by the JDT to construct the Java model representation of java files written or edited using the JDT. This Java model is then used to implement some of the advanced features of the JDT which among others include incremental compile, outline views of the source code and queries of the method call hierarchy. The Core sub component of the JDT comprising 17,923 methods, 1,130 class files and over 400,000 LOC satisfies our requirements for this experiment, being independent of the experimenters, of non trivial size, open source, available as a part of an offline repository copy and having a large collection of associated natural language documentation while also being of a practical size for experimentation.

4.2 Concept Set Definition

Definition of the concept set is the most important and difficult part of designing a concept location experiment such as the one described here. Problem domain concepts are by their definition human orientated (Biggerstaff et al. 1993) and as such they require human involvement in their identification and definition. In a previous experiment we defined a set of concepts based on feature descriptions authored by a human system expert (Cleary and Exton 2007). In this experiment, however, we wished to placate the validity issues raised by this strategy while at the same time increasing the number of concepts

which the experiment could be performed over, so as to increase the generality of the results.

The authors in Poshyvanyk et al. (2006a, b) and Poshyvanyk and Marcus (2007) describe their use of bug tracking databases to define a small number of concepts which are then used to evaluate the performance of a concept location technique. The authors manually search the bug tracking database for bugs which are in a fixed or completed state, that is, bugs for which patches have been developed. These bugs then are the concepts or features which the authors use in their experiments. This concept set identification strategy is made possible because bug tracking databases act as repositories where human experts (over time) define and describe their knowledge about concepts (bugs) related to the system under study. This is of course a by product of a more immediate priority that software engineers have to record and coordinate the resolution of bugs or defects in their systems. However, it is a valid strategy in that bugs can be considered as unwanted features or concepts (Poshyvanyk et al. 2006a, b; LeGear et al. 2005).

Our experiment in comparison with that described in Poshyvanyk and Marcus (2007) requires a much larger number of concepts. As such rather than a manual process for identifying suitable bugs we define a semi-automated approach. We start defining the concept set by querying the Eclipse bug tracking database (Bugzilla) to elicit bug reports which match the criteria for the experiment. Those criteria were firstly that; bugs should have been defined reasonably recently or have been updated reasonably recently so that the current state of the code base would not diverge significantly from that to which the bug relates. Secondly bugs should be considered complete within the bug tracking database, that is, we are interested only in those bugs which we expect would require no further commit operations.

Bugzilla defines a bug life cycle in terms of the stages which a bug can be in at any one time. A bug should follow a path from NEW—(being defined by a user) to ASSIGNED—(allocated to a specific developer) to RESOLVED (changes committed to source code control and a resolution type assigned) to VERIFIED—(checked by another developer) to CLOSED (bug considered complete). While this is the hypothesised ideal path for a bug to undergo, from our observations of the Eclipse bug tracking database we have seen that frequently many bugs remain at the RESOLVED and VERIFIED stages without moving to the CLOSED stage. A second complication is that bugs when moved to the RESOLVED state should be assigned a resolution type of either FIXED, DUPLICATE, WONTFIX, WORKSFORME, INVALID, REMIND or LATER. Table 3 describes the numbers of JDT core bugs at the RESOLVED, VERIFIED or CLOSED stages, first for the entire history of the JDT and second for the period of the experiment 01 January 2006 to 31 December 2006.

Table 4 describes the number of bug reports at the RESOLVED, VERIFIED and CLOSED stages which were assigned a resolution of FIXED. As these tables demonstrate

Table 3 RESOLVED, VERIFIED and CLOSED JDT core bug reports

Stage	Resolution	Total	01/01/06–31/12/06
RESOLVED	*	5,225	1,101
VERIFIED	*	3,637	922
CLOSED	*	372	75

Table 4 RESOLVED, VERIFIED, CLOSED and FIXED JDT

Stage	Resolution	Total	01/01/06–31/12/06
RESOLVED	FIXED	846	82
VERIFIED	FIXED	3,436	821
CLOSED	FIXED	136	9

in both the total and recent past of the JDT core component there is a significant disparity between the numbers of bugs marked as being at the RESOLVED stage with some resolution type and those marked as being at the RESOLVED stage with a resolution type of FIXED.

As such for the experiment we choose to only include bugs which were in the VERFIED stage and had a resolution type of FIXED in the concept set. We feel that the numbers of bugs that matched these criteria were sufficient to draw conclusions from and that the state and resolution types of these bugs were more appropriate to performing experiments over.

4.3 Query Term Sets Definition

Accurate definition of the query term sets is important because the performance of the IR techniques under examination will depend on the quality of the queries they are presented with. It is difficult in that given a bug report one has to define, in an unbiased manner, a set of terms which are appropriate as a short form description of the intent of the bug report which itself might be an aggregation of several different concepts defined at different granularities and abstractions.

There are several possible methods available for generating a set of query terms for a given bug report;

- Create a panel of human experts and novices to read bug reports and generate queries that they consider representative of the intent report.
- Automatically generate a query term set based on the content of the bug report.
- Use metadata already present in the bug report to define the query term set.

One of the guiding objectives of this experiment, in order to increase external validity, repeatability and to remove any bias, is to automate as much of the process as possible. As such we reject the first option due to the inability to identify a large enough expert panel with the requisite knowledge and also to ensure that the query term sets generated are without bias. Then, the question remained, given that we want to perform a completely automated experiment, how do we automatically define queries that would match the queries generated by a human. We investigated the second option by building a tool that would read bug reports and then generate a set of the most representative terms in the report (based on the popular tf-idf measure; Salton and Buckley 1987) which could then be used as the query term set for that bug report. While promising, this approach proved to generate query term sets that were quite unlike what one would expect from a human and so we chose not to use this generative approach. This is most likely due to the tool being unable to incorporate terms not in the bug report in order to describe that bug report where as the human was able to read it, interpret it and summarize it using terms that were more representative but not necessarily prominent or for that matter in the bug report at all.

What we required was an automatically derivable short form description of the bug report which would be equivalent to that which a human might produce. While Bugzilla

does provide the user the option of providing various metadata about a bug when creating a bug report including a set of key words, this option is rarely used. Users do, however, generally attempt to describe the problem the bug report is related to using the summary field into which they input roughly three or four terms they consider descriptive of the problem. This source then provides an adequate human defined short form description of the associated bug report and as such is a source of query terms which would be equivalent to that generated by a human expert reading the bug report. To implement this strategy for extracting the query term set from a bug report we constructed a simple tool to read the bug reports and parse the summary field. For each, we limited the number of terms in the query to a maximum of three terms but did not apply any other pre-processing to the query term set such as stemming or stop word removal. We choose the three term limit based on our observations of users searching behaviour from previous experiments (Cleary and Exton 2007).

4.4 Relevant Element Set Definition

In order to evaluate the performance of the cognitive assignment technique we need to be able to compare the results generated by it and other IR concept location approaches against known sets of relevant elements generated by some expert for a set of test concepts. One of the primary objectives of this experiment is to test the cognitive assignment techniques concept location performance against a large system and large number of concepts. Given these objectives, manual definition of the relevant element set using one or more expert software engineers was deemed to be intractable.

Defining the relevant element set for a given concept requires us to be able to identify a set of elements in the code base of the system under study which are relevant to the concept. To distinguish a subset of the source code elements that comprise a system under study as being relevant to a concept, a concept location technique requires some evidence on which to base its hypothesis as to the relatedness of a given element. Different automatic concept location techniques can be distinguished based on the source of the evidence they use to generate their hypothesises. A form of evidence which has recently been the focus of increased interested (Hassan and Holt 2004; Zimmermann 2006; Kagdi et al. 2007) in the software comprehension and reengineering communities comes from source code control and versioning systems, sometimes called source code repositories. These systems such as CVS and subversion allow software engineers to maintain a database of changes applied to the files that comprise a software system.

These databases, among other features, allow multiple engineers to collaborate on the same set of files, to see which engineer worked on a particular version of a file, and to compare the current version of a file with its predecessors. In addition when an engineer “commits” or “checks in” a revision to a file the engineer is asked to attach a natural language commit comment describing the changes that were applied to the file which is saved in the database along with the revision. In some organisations and on some open source projects it is customary to insert into the commit comment a “bug” or “task id” that relates the file revision committed to the source code control system to a database of bugs related to the project, allowing engineers to associate the changes made to a file to a specific bug report or feature request.

This largely informal practice of inserting a bug id into a commit comment, in effect labels that commit operation and its associated file revisions as being relevant to the associated bug id. If we consider bugs or feature requests as concepts which we want to localize to a set of source code elements then the commit comment log maintained by

source code control systems could be used as the basis of a coarse grained static mechanism for localising concepts to sets of source code files relevant to those concepts.

We can envisage a naïve static concept location method based on the commit comment log which would accept a bug id and return the set of files relevant to that bug, the relationship between the bug id and file set being determined from the set of commit operations whose associated commit comments contain references to the given bug id. Essentially we derive a coarse grained concept to file mapping based on a search of the commit comment log for occurrences of a bug id. A static concept location technique defined at this level of granularity would, however, be only of limited usefulness in that it would be only capable of indicating which source files were relevant to the given bug id. While file level localisation might be potentially useful for assisting understanding in very large systems, ideally we would like to be able to accomplish concept location at a finer sub file level of granularity, specifically at the method or function level.

Given a file deemed relevant to a concept as determined by the naïve concept localisation procedure described above, how do we identify the set of software elements contained in that file that are relevant to the concept. A possible solution would be to identify the specific lines in the file that were modified from the previous revision in the commit operation. Given this information, and a parser capable of parsing the file to identify programming language elements contained within, we can resolve from line numbers altered in the commit operation to software elements altered in the commit operation. This static concept localisation method then is capable of generating for a given concept a relevant element set defined at a granularity which we use to evaluate the cognitive assignment technique against other IR based concept location techniques and which also allows comparison with related work.

However, at this point it is beholden on us to state that this method, while a tractable solution to the relevant element set definition problem, is not perfect and falls down in two key respects. First it is reliant on the software engineer tagging commit operations with the correct bug identifier. Second is the issue of the semantic relevancy of the automatically derived relevant element set, in that, while a software engineer may explore many elements in attempting to fix a bug, only a few may need to be modified and so only this subset will be recognised as relevant by the procedure described above. This means that while the procedure will generate relevant element sets with high precision the recall may be lacking. While this is a limitation of our research, it is unfortunately unavoidable in the absence of professionally pre-categorised collections such as those used by the information retrieval communities (TREC 2007).

4.5 Corpus Definition

Before computing the system under study specific HAL space for use by the cognitive assignment technique in the experiment, we needed to define a corpus of natural language documentation associated with the system under study. We decided to use three non-source code documentation sources all related to the JDT core component for this experiment; help documentation, newsgroup postings and bug reports. In total the corpus was comprised of 3,770 documents, 142,325 lines of text or 787,500 words. The documentation types, their source and the quantity of each type of document are listed in Table 5.

We collected artefacts in each category for the period from 01 January 2006 to 31 December 2006. Help documentation was sourced from the most current version of the eclipse project related to the JDT component. No distinction was made between help documentation as it applied to the different JDT components as it was not automatically

Table 5 Corpus document source and quantity

Document type	Source	Quantity	Size (kB)
Help documentation	http://help.eclipse.org/help31/index.jsp, http://www.eclipse.org/articles/	8	146
Newsgroup	http://www.eclipse.org/newsportal/thread.php?group=eclipse.tools.jdt	2941	3,270
Bugzillia	https://bugs.eclipse.org/bugs/query.cgi	821	2,740

possible to do so and was not deemed necessary. We collected news group postings relating to the JDT from the eclipse.tools.JDT newsgroup for the period of 1 year from 01January 2006 to 31 December 2006. We downloaded the message files from the eclipse news server and applied a script to remove the header information from each message file, leaving only the content of the message. Again, like the help documentation, the newsgroup postings were sourced from a JDT newsgroup and no attempt was made to identify only those postings which related to the Core component. Finally bug reports for the component and the period were collected from the Eclipse Bugzilla bug tracking database; these, unlike the help docs and news group messages were specific to the Core component of the JDT product because the bug tracking database allowed us to make that distinction. We used an in-house tool to parse the set of concepts defined using the concept set definition procedure outlined in Section 4.2 which generated for each concept a file containing the related bug reports description, comments and attachments.

4.6 Computing the HAL space

Having defined the natural language documentation corpus for the experiment we then used the freely available AutoMap tool (Diesner and Carley 2004) version 2.6.50 to compute a HAL space based on the patterns of term co-occurrence within the documentation corpus. The procedure used is as follows;

- We first imported the documentation corpus files into AutoMap.
- We then applied four pre-processing techniques to reduce the number of unique and total concepts in the corpus (the results of each of these steps on the total and unique sets of terms in the corpus are detailed in Table 6).
  - We applied an analysis which removed all numbers and symbols from the corpus.
  - We then applied the porter stemming algorithm using English as the language to reduce inflected words or terms to their root or stem.
  - Next we applied a stop delete list to remove common words, further reducing the total number of terms in the corpus.

Table 6 Total and unique terms after pre-processing

Pre-processing step	Total	Unique
No pre-processing	934,242	69,509
Remove symbols	833,432	42,600
Stemming	833,432	38,871
Stop list	653,710	38,836
Frequency limit	504,050	9,429

  - Finally, we removed from the term set those terms which occurred with a frequency of less than three times. The stop list used for the experiment and the set of terms removed as a result of applying the frequency limit can be accessed online at (https://forager.svn.sourceforge.net/svnroot/forager/Experiments/JDT Experiment/stoplist.txt; Cleary 2007).

- Next we configured the AutoMap map analysis settings that control how the HAL space is constructed.
  - We chose to create a unidirectional map where associations are created between terms occurring in a window in one direction only.
  - A window size of 9 was chosen for constructing relations between terms, this choice being motivated and consistent with the relevant literature.
  - We then instructed Automap to generate a semantic network (HAL space or co-occurrence matrix) file for each file in the corpus.
- Finally we combined these file specific HAL spaces into a single HAL space for the corpus using the CompareMap utility in AutoMap. This resulted in a single CSV file which represented the HAL space for the document corpus defined in Section 4.5.

The final HAL space for the JDT core documentation corpus contained 1.3M unique statements and 7M statements in total.

4.7 Source Code Document Index Generation

Generally the first step in applying any information retrieval technique to searching a document corpus is to build an index of that corpus. The index is a short form representation of the document corpus, that is, a simpler form of the original set of documents: for example, the set of terms in the document corpus and the number of times each occurs. Indexing is performed primarily to improve the runtime performance of the retrieval operations at the cost of a once off indexing operation.

In this experiment we were required to generate an index of our document corpus to satisfy the requirements of the information retrieval techniques we were comparing. However, this index was to be computed not from natural language documents but rather from source code documents (which in this study are methods), as such the details of how this index was constructed deserves some consideration.

The source code document indexer component of the cognitive assignment plug-in (Cleary 2007) is responsible for computing a short form representation of a set of source code documents. The type of short form representation generated by the source code document indexer is dependent upon the types of document scoring functions in use and on the programming language of the source code documents. Typically a document scoring function will require some form of a frequency table that describes the frequency of occurrence of terms in the document, however, some of the more complex document scoring functions require co-occurrence matrices to be calculated for each document. The basic operation of the source code document indexer is described next.

First the indexer splits the source code document into a sequence of non unique terms. In indexing natural language based documents this step is relatively straight forward as typically white space can be used to delimit one term from another. However, as the

documents to be indexed are source code documents and not natural language documents we have to look to other methods for splitting a source code document into sets of terms.

Existing techniques for term set identification from source code documents typically rely on the importance of meaningful identifiers to an engineer's ability to understand source code. These techniques, using heuristics that define where and how software engineers tend to use meaningful identifiers, perform a selective parsing of the source code of the system under study to generate a set of terms (Anquetil and Lethbridge 1997; Merlo et al. 2003). While simplistic, these bottom-up approaches to term identification can be very effective; however, their success is dependent on the heuristics or assumptions that the technique makes. As such these techniques can fail where meaningful identifiers are not used and are faced with the added problems of abbreviation generation and term splitting.

Other authors, recognising the limitations of deriving a term set from identifiers alone but while wanting to maintain close relations to the source code, have attempted to use comments in the source code as a basis (or in conjunction with identifiers) from which to build a set of terms (Sayyad-Shirabad et al. 1997). One reason for looking to comments for a term set is that they provide a much richer vocabulary of potential concepts while maintaining a relatively strong relationship to the code. Also comments are more akin to natural language and so can be easily parsed. However comments are not executable and as such cannot be completely trusted as being up to date; also, while of greater richness as compared with identifiers, comments are still often written in a constrained vocabulary.

Our approach to identifying a term set representation for a source code document is a mixture of these two approaches in that we make use of both identifiers and comments to form the term set representation of a source code document. We define an expanded delimiter set more appropriate to parsing Java source code. Through trial and error we identified the following set of delimiters as optimal: [" ", ";", "(", ")", "=", "<", ">", ".", "[", "]", ",", "*", "/", "\\", "'", "\t", "\n", "\r", "@", "+", "\""]. Using this delimiter set we segment the source code document in to a sequence of terms much like how simple white space based segmentation works in natural language document indexing.

However, there are further segmentation heuristics (identified from a review of related literature and the author's own programming knowledge) that can be employed to improve the quality of the term sets identified:

1. We remove Java keywords from consideration as terms.
2. We break apart terms identified from the delimiter analysis using a capitalisation heuristic which breaks apart terms based on the appearance of capitalised letters within the term signifying that the term is a compound term.
3. We break apart terms which include underscore characters "_" into separate terms.
4. To improve the quality of the term set we apply a stop list (available online) which removes very common words such as: ["and", "for", "is", "if", "i", "the"] and impose some term length constrains on the terms that can be included in the term set. Specifically we restrict terms that are less than or equal to two characters in length.
5. Lastly we implement stemming on the term set to remove redundant terms.

Having split incoming source code documents into sequences of terms we then compute the representations required by the different scoring functions supported by the cognitive assignment plug-in; specifically we compute three representations:

1. A simple frequency table describing the number of times a term from the term set of the document occurs in the document.

2. A vector representation where each term in the corpus vocabulary is allocated a dimension and if a term occurs in a particular document that term's dimension in the document is set to the term's frequency of occurrence.
3. A co-occurrence matrix for the document is calculated using a windowing based approach over the term sequence.

Finally we also calculate a global term frequency table which serves as a corpus vocabulary and a count of all the terms in the corpus. When we have finished computing the index representation of the source code document we insert it into the source code document index.

4.8 Experiment Procedure

This section describes the detailed experiment procedure;

1. First we manually selected a set of concepts using the concept set definition procedure outlined in Section 4.2 using the web based interface to the Eclipse Bugzilla bug tracking database. We specifically selected bugs which were in the verified and fixed states and which were expressly classified as being relevant to the JDT Core component (821 concepts).
2. Next we used the concept set to automatically extract a set of query terms from the Eclipse bug tracking database using a query term set extractor application that implements the procedure defined in Section 4.3. The set of concepts and corresponding query term sets were recorded in a simple comma separated value file.
3. A relevant element set generator application implemented in accordance with the procedure specified in Section 4.4 was used to generate, for each concept, a set of relevant elements using an offline copy of the Eclipse source code repository created on 31 December 2006.
4. Next we manually and automatically collected the documentation corpus for the experiment from the documentation sources and using the procedure described in Section 4.5.
5. The system under study specific HAL space for the experiment was then generated based on the documentation corpus using the procedure described in Section 4.6.
6. Next we initialised the cognitive assignment tool for the experiment. This entailed launching a new instance of the eclipse IDE and opening the cognitive assignment plug-in. We then imported the JDT Core source code (the system under study) as a new java project in the Eclipse IDE from the offline copy of the Eclipse source code repository created on 31 December 2006. This automatically caused the cognitive assignment tool to begin building a source code document index for the newly created project.
7. We then read the system under study specific HAL space file generated in step 5 into memory using the cognitive assignment plug-in. This then populated the non-source code document index used by the cognitive assignment technique.
8. We then set the cognitive assignment plug-in to operate in automatic mode (a mode designed to run experiments without displaying results to the UI or requiring manual intervention). Next we loaded the concept and query term sets file into memory, and instructed the Cognitive Assignment plug-in to compute result element sets for each query term set, storing each result set in an individual file. This generated a file containing set of rankings for each document in the system under study for each concept for which we computed a query.

9. Finally using a simple in house script we compared the result element sets with the relevant element sets calculated earlier using the average precision measure described in Section 5.1.

4.9 Document Scoring Functions

In this section we briefly provide an overview of the different document scoring functions which we later compare to the cognitive assignment technique and the similarity measures employed by each. The interested reader is referred to the cited sources for more details on particular techniques, however, it should be noted that each technique uses an index computed as per the procedure described in Section 4.7.

As described earlier the classic language model (LM) scoring function considers source code documents as a simple "bags of words", that is to say, the order of occurrence of terms in the source code document is not considered:

$$\text{Similarity}_{\text{LM}}(Q, d_i) = \prod_{q_i \in Q} P(q_i|d_i).$$

To avoid the zero probability problem we have to "smooth" the probability function, that is introduce some small non zero value into the function that prevents it from going to zero in cases where a query term does not occur in the document:

$$\text{Similarity}_{\text{LM}}(Q, d_i) = \prod_{q_i \in Q} ((1-\lambda)P(q_i|C) + \lambda P(q_i|d_i)).$$

The interested reader is referred to Zhai and Lafferty (2004) for more details on the classic language model information retrieval technique.

The dependency language model (DLM) is similar to the LM technique except it attempts to relax the independence assumption by reintroducing dependencies back into the unigram scoring function, for this experiment we used the algorithm described in Gao et al. (2004) which uses the following similarity measure:

$$\text{Similarity}_{\text{DLM}}(Q, d_i) = \log P(Q|d_i) = \log P(L|d_i) + \sum_{i=1}^{n} \log P(q_i|d_i) + \sum_{(q_i,q_j) \in L} \text{MI}(q_i, q_j|L, d_i)$$

where L is a linkage or a set of pairs of related terms selected from the document based on the query terms and where MI or the mutual information of the terms in the linkage is given by:

$$MI(q_i, q_j|L, d_i) = \log \frac{P(q_i, q_j|L, d_i)}{P(q_i|D)P(q_j|d_i)}.$$

The vector space model (VSM; Salton et al. 1975) is one of the simplest and most fundamental information retrieval models. Following a pre-processing stage and given a unique set of terms T of size n from a set of documents D, and a term-document matrix M that describes the frequency of occurrence of term t_i in document d_j for all $t_i \in T$ and $d_j \in D$, we can describe document d_j as an n-dimensional vector X_j where for $1<i<n$, $x_{ij} = twc_{ij} \times gwc_i \times nc_j$.Where twc_{ij}, gqc_i and nc_j are weighting components designed to positively effect document discrimination.

If we represent our query Q as an n-dimensional weighted vector Y and our document d_i as an n-dimensional weighted vector X we can calculate the similarity between the two as follows:

$$\text{Similarity}_{\text{VSM}}(Q, d_i) = \text{Similarity}(Y, X)$$

where Similarity(Y, X) is defined using the cosine coefficient (Salton 1989) and where X_i and Y_i are the ith components of vectors X and Y respectively:

$$\text{Similarity}(Y, X) = \frac{\sum_{i=1}^{n} X_i \times Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2 \times \sum_{i=1}^{n} Y_i^2}}.$$

Latent semantic indexing (LSI) is an extension to the VSM that applies an extra analysis step, called singular value decomposition (SVD), to the weighted term–document matrix prior to that matrix being used in computing a similarity measure between documents (Landauer et al. 1998). As such LSI is able to employ the same similarity measure as the VSM document scoring function described previously. We use the SVDLIBC library (Rohde 2007) to implement the SVD step of our LSI scoring function. SVDLIBC is a C library based on the SVDPACKC library (Berry et al. 2007) which provides methods for computing the singular value decomposition of large sparse matrices.

Finally the Kullback Leibler-divergence similarity measure, which we have discussed previously, was implemented according to (1).

The different document scoring functions evaluated during the experiment, their abbreviations, and default parameters where applicable are listed in Table 7. The approach discussed in this paper is the cognitive assignment approach which we shorten with the abbreviation CA. Except for the latent semantic indexing scoring function, each algorithm was implemented by the authors of this paper.

The abbreviations detailed in Table 7 are used later when discussing the different document scoring functions and are presented here for easy reference. The document scoring parameters listed here are the defaults used during the experiment.

5 Results and Analysis

A qualitative analysis of these document scoring functions would require studying the performance of each in locating individual concepts and making qualitative judgments about those performances based on the attributes of the concepts. While such a detailed analysis is

Table 7 Document scoring functions

Name	Abbreviation	Default parameters
Classic language model	LM	NA
Dependency language model	DLM	NA
Vector space model	VSM	NA
Latent semantic indexing	LSI	d=300
KL-divergence	KLD	NA
Cognitive assignment	CA	m=0.5, x=50

desirable, it is intractable in this experiment due to the large number of the concepts over which the experiment was run and due to the lack of access to a sufficiently qualified system expert. As such, a quantitative analysis of the performance of the document scoring functions was the only practical method of evaluation that could be employed considering the size of the system under study and the number of concepts which we wished to localise.

5.1 Average Precision Analysis

Average precision is defined as the mean of the precision scores obtained after each relevant document is retrieved (Kishida 2005). In Buckley and Voorhees (2000) the authors further qualify this definition by defining average precision as the mean of the precision scores obtained after each relevant document is retrieved, using zero as the precision for relevant documents that are not retrieved. It can be derived based on p_m:

$$v = \frac{1}{R}\sum_{i=1}^{n} I(x_i)p_i$$

where R is the total number of relevant documents in the collection and n is the number of documents included in the ranked document list. $I(x_i)$ is a function such that:

$$I(x_i) = \begin{cases} 1 \text{ if } x_i = 0 \\ 0 \text{ otherwise} \end{cases}.$$

If x_i is defined as x_k above that is as a binary variable then we can set $I(x_i)=x_i$. Given this we have the following definition for average precision.

$$v = \frac{1}{R}\sum_{i=1}^{n} \frac{x_i}{i}\sum_{k=1}^{i} x_k. \tag{6}$$

Equation 6—Average Precision

The mean average precision (MAP) is the mean value of the average precisions computed for each of the queries separately. MAP allows us to summarise the large amount of data generated by the experiment. It is one of the most commonly used measures in the information retrieval community to summarise the potentially large sets of results that are required by large scale information retrieval experiments. However, the interested reader is referred to the website (https://forager.svn.sourceforge.net/svnroot/forager/Experiments/JDTExperiment; Cleary 2007) where we publish the full average precision results for each of the 698 concepts analysed in this experiment, along with the query term sets and relevant element sets used in this experiment.

For this experiment MAP is used to measure the mean performance of a document scoring function over the entire set of concepts which we are studying. To calculate the MAP of a document scoring function over the set of concepts, we simply sum the average precision scores for the document scoring function for each concept and divide the result by the number of concepts. More formally, let v_h be an average precision score for the hth concept. If we have L concepts in our experiment then we can calculate the mean of the average precision scores $\nu_1,\ldots,\nu_L$ using the following formula taken from (Kishida 2005);

$$\overline{v} = L^{-1}\sum_{h=1}^{L} v_h. \tag{7}$$

Equation 7—Mean Average Precision (MAP)

This then generates a single value representing the average performance of that document scoring function. As there are six document scoring functions under study in this experiment we ran the mean average precision analysis over the raw average precision data six times, once for each set of data corresponding to each document scoring function, generating six values describing the average precision of each of the document scoring functions. The results of this analysis are presented in Table 8.

As can be seen from Table 8 none of the document scoring functions performed particularly well over the entire concept set. The best score was achieved by the KLD function which scored over 6.5% average precision over the entire concept set. The CA technique fared a little worse achieving a 5.6% average precision score, while VSM achieved a little more than 4%. This group is then trailed by the rest of the techniques which are lead by the DLM technique at 1.5% followed by both the LSI and LM techniques who achieve similar scores of 0.67% and 0.6% respectively.

While these results might give the impression of poor performance of information retrieval based concept location, this is not necessarily the case. The mean average precision metric is calculated over the entire concept set (698 concepts, the number of concepts for which we were able to generate relevant element sets). As such, even if a technique achieved perfect average precision on some concepts but performed poorly on the majority, the mean average precision of the technique would be very poor. As such the mean average precision, while useful as a mechanism for summarising a very large data set, could potentially hide important aspects of the performance of a document scoring function, such as the types of concepts over which the function performed well or, potentially more importantly, the types of concepts for which the document scoring function did not perform well.

Table 9 breaks down the mean average precision analysis presented in Table 8 to show, for each 10% range of average precision from 0% to 100%, the number of test concepts which each technique was able to rank within that average precision range. As is to be expected given the overall mean average precision scores, most of the test concepts are ranked with 0% to 10% average precision. However, it is interesting to note that from this analysis both the KLD and CA techniques perform similarly within the 40% to100% range, ranking 27 and 26 test concepts within this range respectively. In this regard, they are quite dissimilar to the other techniques. Unfortunately the number of concepts for which this relatively good performance was achieved only corresponds to slightly more than 0.03% of the total number of test concepts used in the experiment.

In order to analyse further the results for each of the document scoring functions we have performed the Friedman test (Friedman 1937), to determine if the average precision results for the six document scoring functions are different. This is a non-parametric test which compares the average rank of each of the document scoring functions. The need to apply non-parametric statistics to the average precision data is demonstrated through the distributions demonstrated in Table 9 and through the box plots presented in Fig. 8. These box plots demonstrate the highly skewed nature of the data for each document scoring function and hence the appropriateness of non-parametric statistics to this data.

Table 8 MAP for six document scoring functions

	LM	DLM	VSM	LSI	KLD	CA
MAP	0.0060466	0.0153495	0.0403908	0.0067179	0.0650573	0.0561173

The results of the Friedman test are presented in Tables 10 and 11. The null hypothesis for this test states that the six IR document scoring functions have equivalent average precision. However, given the significance value in Table 11 (<0.001), the null hypothesis can be rejected. This means that not all document scoring functions are equivalent and thus a further test is required to determine if the difference between any two document scoring functions is statistically significant.

The Nemenyi test (Nemenyi 1963) is used for this purpose. In this test, the performance of two document scoring functions is statistically significant if their mean ranks differ by at least the critical difference (CD). CD is calculated as follows

$$CD = q_\alpha \sqrt{k(k+1) \div 6N}.$$

If α=0.05 then q_α=2.85 (see Table 5 of critical values for the two-tailed Nemenyi test in Demsar 2006). In this experiment k=6 and N=698 and, on this basis, CD=0.285. Thus, based on the values in Table 10, we cannot say that the document scoring function CA is statistically significantly better than KLD. However these two scoring functions are significantly better than all of the rest. VSM in turn is significantly better than the remaining three techniques and while DLM is significantly better than LM, no conclusion can be drawn as to whether LSI is better or worse than either LM or DLM.

Returning to the contents of Table 9 and the discussion after it, it is very unlikely that all information retrieval based concept location techniques will perform perfectly in all situations. What is possible however is that there will be certain circumstances (combinations of systems and concepts) where different information retrieval based concept location techniques will perform particularly well or particular poorly. The importance of establishing the situations where information retrieval based concept location is applicable cannot be overlooked and is an objective of this paper. As such the following section will attempt to expand on the basic mean average precision analysis of Table 8 and to elicit the differences in the performances of different document scoring functions over different types of concepts.

5.2 Relevant Element Set Size Analysis

In investigating the apparent poor performance of the different techniques we looked at the relevant element sets that were used to compute the average precision scores for each of the techniques. We examined the size of the relevant element sets, to see if size of the relevant

Table 9 Number of test concepts ranked within average precision ranges

	LM	DLM	VSM	LSI	KLD	CA
0.9–1.0	0	3	5	0	11	7
0.8–0.9	0	0	1	0	0	0
0.7–0.8	0	0	0	0	1	0
0.6–0.7	0	0	1	0	3	1
0.5–0.6	2	2	8	0	9	13
0.4–0.5	0	0	1	0	3	5
0.3–0.4	1	1	3	1	20	12
0.2–0.3	2	6	24	1	20	24
0.1–0.2	5	11	28	7	43	35
0.0–0.1	688	675	627	689	588	601

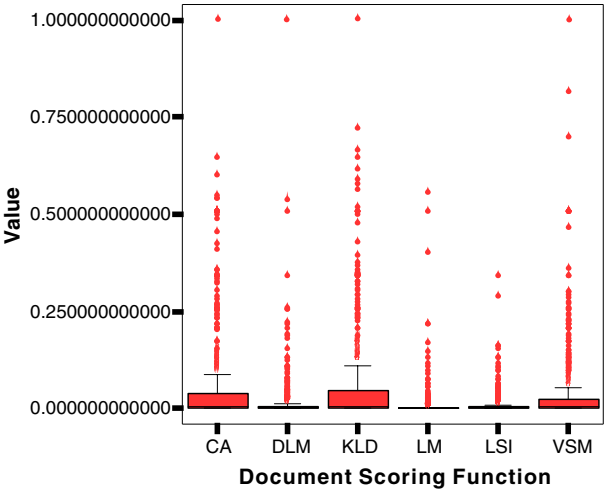

Fig. 8 Distribution of document scoring functions

element set had an impact on performance. To investigate this we looked at the concepts for which each of the techniques performed best to see if there was a correlation between the size of a relevant element set and the performance of the different IR techniques. Table 12 documents the average size of the relevant element sets for the top 10, 20, 50, 100, 200, 300, 400, 500, 600 and 698 (the complete concept set) concepts for which each of the techniques performed best.

From this table we see that, for the VSM and KLD techniques, the top 10 concept average precision scores are generated for concepts whose impact set contained on average 1.5 documents. In contrast, for CA, the average document impact set size for the top 10 ranked concepts is 3.6. This indicates that the CA technique while not performing statistically better than KLD is performing better over different types of documents. Whereas KLD is working well with concepts with a single corresponding relevant element, CA works well with concepts with multiple documents in their relevant element set.

Admittedly, it is difficult to draw conclusions from such a small sample, but the data in general indicates that the best performing techniques (VSM, KLD and CA) perform better for concepts with small numbers (≤5) of documents in their relevant element sets, while the less well performing techniques do best for concepts with large impacts sets (approximately ten documents).

Table 10 Mean ranks

	Mean rank
LM	0.43
DLM	0.87
VSM	0.77
LSI	0.62
KLD	4.52
CA	4.77

Table 11 Friedman test statistics

N	698
χ^2	1,047.014
df	5
Asymp. Sig.	<.001

5.3 Qualitative Analysis

In an effort to identify underlying reasons for the failure of the cognitive assignment technique to significantly improve on the mean average precision score of KLD, we performed a small qualitative analysis comparing the performances of the two techniques. For our analysis we chose two concepts to manually analyse (Table 13), 1 concept for which KLD outperformed CA and a second for which CA outperformed KLD.

The first concept we analysed (83005) had a single element in its relevant element set. We assessed this method in an attempt to identify a set of characteristics which might explain the success of the KLD technique and the relative failure of the CA technique in identifying it as being relevant to the query. The method examined was the complete (ASTNode ASTNode Binding Scope boolean) method in the org.eclipse.jdt.internal.codeassist.CompletionEngine.java class, while the method is too large to print here we can summarise our findings. On examination we found that the method was relatively large, measuring 810 lines of code in total but that less than 2% of the method lines were natural language comments (15 LOC). There were no JavaDoc comments for the method. It was also interesting to note that reviewing the method revealed very few abbreviations with most variables and method invocations using the standard Java naming conventions. In terms of the original query, two of the three query terms appear in the method several times; this is likely why the method was ranked so highly by the KLD technique. When we looked at the terms which were used to expand the original query by the cognitive assignment technique we saw that most of the terms introduced from the HAL IF analysis did not occur in the target method. This is ultimately why the cognitive assignment technique failed to outperform the KLD technique for this concept. When terms are introduced into the query if those terms do not occur in the relevant element then there is a risk that those terms will occur in other documents in the corpus. In such cases the cognitive assignment technique will incorrectly rank other non-relevant documents as being more relevant to the original query which will in turn result in poor average precision.

Table 12 Average relevant element set size for best performing concepts

	LM	DLM	VSM	LSI	KLD	CA
10	10.0909091	4.54545455	1.72727273	13.0909091	1.09090909	3.63636364
20	25.952381	7.76190476	2.23809524	10.6190476	3.42857143	2.52380952
50	33.1764706	13.0588235	4.78431373	20.6666667	5.50980392	4.68627451
100	32.4257426	23.9306931	12.9504951	18.5247525	7.57425743	8.72277228
200	25.079602	21.8208955	13.1144279	19.5174129	12	13.1691542
300	20.0265781	18.2624585	14.0365449	17.6744186	14.6146179	14.4551495
400	16.0448878	15.1446384	13.2294264	14.9875312	13.7955112	14.9077307
500	13.2075848	13.1676647	12.7045908	12.9181637	12.6187625	12.8902196
600	11.2246256	11.359401	11.3294509	11.344426	11.3294509	11.2845258
698	10.0659026	10.0659026	10.0659026	10.0659026	10.0659026	10.0659026

Table 13 Concepts subjected to quantitative analysis

Concept ID	Query	KLD score	CA score
83005	Assist, annotation, override	1	0.25
165069	Compiler, field, hiding	0.36	0.41

The second concept we analysed (165069) was chosen as it was the first concept with more than one element in the relevant element set for which CA outperforms KLD. We assessed each method using the same procedure as before. First we examined the getBinding(char[] int InvocationSite boolean) method in the org.eclipse.jdt.internal.compiler.lookup.Scope.java class. This method consisted of 244 lines of code, including 40 lines of natural language comments (over 16%). The method conformed to Java naming conventions for variables and method invocations with very few abbreviations. Of the original query "compiler" and "field" occurred several times in the method, however, "hiding" did not occur. The expanded query used by the CA technique included several terms which occurred in the method. The second method analysed was the resolve (MethodScope) method in the org.eclipse.jdt.internal.compiler.ast.FieldDeclaration.java class. This method consisted of 139 lines of code of which 24 were natural language comment lines. That is over 17% of the method which consisted of comments. This method, like the rest, conformed to Java naming conventions for variables and method invocations with very few abbreviations. Of the original query "compiler" and "field" occurred several times in the method, however, "hiding" did not occur. The CA technique included several terms in the query which occurred in the method such as "problem" which occurs multiple times.

From this brief analysis we can surmise that the CA technique was able to outperform the KLD technique for this concept due primarily to the query being expanded with terms which did occur in the relevant elements, whereas in the previous case the terms introduced into the query harmed the performance of the technique: here the terms introduced were relevant to the query and so helped, not hindered, the technique. Additionally the CA technique seems to perform better where the relevant elements are smaller in size and where a greater percentage of the relevant elements were natural language comments.

6 Conclusions and Future Work

In this paper we have presented the cognitive assignment technique, a new approach to assisting concept location in software comprehension which uses information flow and co-occurrence information derived from non source code artefacts to implement a query-expansion-based concept location technique. We have also presented an experiment which assesses the performance of the cognitive assignment technique relative to other concept location techniques. Our analysis shows that the cognitive assignment technique is very competitive with the other techniques assessed in the experiment when compared using MAP. When using the Friedman and Nemenyi tests it out performs all but one other technique at a statistically significant level. That other technique is the KLD based concept location approach, from which the cognitive assignment technique is derived. However, the mean rank for CA is larger than the corresponding value for KLD, although not with statistical significance. This result is slightly disappointing because the set of extensions which differentiate the cognitive assignment technique from the KLD technique, are

designed and hypothesised to improve its performance, but seem to only produce a marginal, non-significant performance gain.

While disappointing, this finding is interesting in that the extensions to the underlying KLD technique which have proven successful in other experiments, in other domains have been shown to be less successful when applied to the software concept location problem. The cause of this apparent difference in performance of information retrieval over natural language and source code most likely rests on the factors that differentiate natural language from source code such as the proportion of natural language embedded in the source code, the naming conventions used by the software engineer and the vocabulary of keywords used by the particular programming language. This would imply that techniques which are applicable to natural language information retrieval may not be applicable to the same extent when applied to concept location in source code.

Our analysis of the relatively small gain of the cognitive assignment technique over that of the KLD technique, however, has led to one of the most interesting findings to emerge from the experiment. While the initial analysis presented allowed direct comparison between the performances of the different concept location techniques, the analysis says nothing about the types of concepts for which the techniques performed well. From a purely quantitative analysis it is difficult to perform a detailed analysis of the characteristics of concepts for which the different techniques perform best, but we were able to identify that the cognitive assignment technique and the KLD techniques were performing best for different concepts with different characteristics. Specifically the cognitive assignment technique performed best in localising concepts with a larger relevant element set, while the KLD technique performed best for concepts with a slightly smaller relevant element set. As such, while the quantitative analysis measure does not show any difference between the techniques other than the raw concept location performance, we have shown from our analysis that there can be significant differences in the types of concepts for which different techniques perform best. This implies that, for different types of concepts, not all concept location techniques are equal, and supports the hypothesis that multiple concept location techniques should be used in parallel. Such a system would, instead of providing a single document ranking, provide the user with multiple potential document rankings in response to a query, each generated by a different concept location technique. These rankings could be combined into a single suggested ranking for a document or be presented directly to the user who would be permitted through the different document rankings using a suitable user interface. An example of such a user interface has been developed by the authors of this paper and is presented in Cleary and Exton (2006a, b).

In this study we have not attempted a large scale qualitative evaluation in order to further explain the differences in the performances of the different IR techniques. Instead, we have provided a small qualitative example for two concepts, but we accept that is a very small comparison. An important avenue for future work would be to categorize the concepts and do an analysis for each concept category or alternatively to categorize the documents and do an analysis for each document category.

A final finding from our study was the relatively poor performance of the LSI technique which was unable to outperform the VSM technique and which was outperformed by both cognitive assignment and KLD. While we did experiment with increasing the number of dimensions used in the LSI based technique, the number of dimensions used to generate the results presented here was really the upper limit which could be applied given the computational resources available to us at the time of the experiment. As such, while increasing the number of dimensions might result in better concept location performance there are practical limitations to doing do. Also there is the poor performance of the cognitive

assignment technique to consider; here we see an example of a "semantic" based cognitive assignment technique (like LSI) performing poorly in comparison with the most closely related more basic technique. It might be the case that "semantic" or relation based concept location techniques do not perform well against other simpler and more computationally efficient techniques. However, as we have seen not all concept location techniques analysed here perform well for the same concepts, it may be the case that a more qualitative analysis of these techniques might demonstrate better performance in a real world setting were the techniques are used to support real software engineers engaged in concept location tasks. However, for this hypothesis to be supported more research in this area is required.

References

Anquetil N, Lethbridge T (1997) File clustering using naming conventions for legacy systems. Conference of the centre for advanced studies on collaborative research. IBM, Toronto, Ontario, Canada

Antoniol G, Canfora G et al (2002) Recovering traceability links between code and documentation. IEEE Trans Soft Eng 28(10):970–983

Bai J, Song D et al (2005) Query expansion using term relationships in language models for information retrieval. 14th ACM International Conference on Information and Knowledge Management. ACM, Bremen, Germany

Berry M, Do T et al (2007) SVDPACK. http://www.netlib.org/svdpack/

Biggerstaff TJ, Mitbander BG et al (1993) The concept assignment problem in program understanding. 15th International Conference on Software Engineering. IEEE Computer Society Press, Baltimore, MD, USA

Biggerstaff TJ, Mitbander BG et al (1994) Program understanding and the concept assignment problem. Commun ACM 37(5):72–82

Bruza PD, Song D (2002) Inferring query models by computing information flow. Proceedings of the eleventh international conference on Information and knowledge management. ACM, McLean, VA, USA

Buckley C, Voorhees EM (2000) Evaluating evaluation measure stability. 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, Athens, Greece

Canfora G, Cerulo L (2005) Impact analysis by mining software and change request repositories. 11th IEEE International Symposium on Software Metrics (METRICS'05)

Canfora G, Cerulo L (2006) Fine grained indexing of software repositories to support impact analysis. International Workshop on Mining Software Repositories (MSR'06)

Chung W, Harrison W et al (2005) Working with implicit concerns in the concern manipulation environment. Linking aspect technology and evolution (LATE) co located with aspect orientated software development (ASOD 05). IEEE, Chicago, USA

Cleary B (2007) Cognitive assignment plug-in. https://sourceforge.net/projects/forager

Cleary B, Exton C (2006a) Assisting concept assignment using probabilistic classification and cognitive mapping. 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KSCD2006). IEEE/ACM, Tokyo, Japan

Cleary B, Exton C (2006b) The cognitive assignment eclipse plug-in (ICPC 06). 10th International Conference on Program Comprehension. IEEE Computer Society Press, Athens, Greece

Cleary B, Exton C (2007) Assisting concept location in software comprehension. 19th Annual Psychology of Programming Workshop (PPIG07). Joensu, Finland

Cubranic D, Murphy GC et al (2005) Hipikat: a project memory for software development. IEEE Trans Soft Eng 31(6):446–465

Deerwester S, Dumais ST et al (1990) Indexing by latent semantic analysis. J Am Soc Info Sci 41(6):391–407

Demsar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Diesner J, Carley K (2004) AutoMap1.2-Extract, analyze, represent, and compare mental models from texts. Carnegie Mellon University

Eisenbarth T, Koschke R et al (2003) Locating features in source code. IEEE Trans Softw Eng 29(3):210–224

Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J Am Stat Assoc 32:675–701

Gao J, Nie J-Y et al (2004) Dependence language model for information retrieval. 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, Sheffield, UK

Greenfield J, Short K (2004) Software factories: assembling applications with patterns, frameworks, models & tools. Wiley, New York

Hassan AE, Holt RC (2004) Using development history sticky notes to understand software architecture. Proceedings of the 12th IEEE International Workshop on Program Comprehension

Hill E, Pollock L et al (2007) Exploring the neighborhood with Dora to expedite software maintenance. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)

IEEE (2000) IEEE recommended practice for architectural description of software-intensive systems. Software Engineering Standards Committee

Jones KS, Walker S et al (2000) A probabilistic model of information retrieval: development and comparative experiments. Inf Process Manage 36(6):779–808

Kagdi H, Maletic JI et al (2007) Mining software repositories for traceability links. Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC '07)

Kiczales G, Lamping J et al (1997) Aspect-oriented programming. European conference on object-oriented programming. Springer, Jyväskylä, Finland

Kishida K (2005) Property of average precision and its generalization: an examination of evaluation indicator for information retrieval experiments. National Institute of Informatics, Tokyo, Japan

Knight C, Munro M (2002) Program comprehension experiences with GXL: comprehension for comprehension. 10th International Workshop on Program Comprehension (IWPC 02). IEEE Computer Society Press, Paris, France

Landauer TK, Foltz PW et al (1998) Introduction to latent semantic analysis. Discourse Process 25:259–248

LeGear A, Buckley J et al (2005) Achieving a reuse perspective within a component recovery process: an industrial scale case study. 13th International Workshop on Program Comprehension (IWPC 2005). IEEE Computer Society Press, St. Louis, MI, USA

Littman DC, Pinto J et al (1986) Mental models and software maintenance. First Workshop on Empirical Studies of Programmers. Ablex, Washington, DC, USA

Lund K, Burgess C (1996) Producing high-dimensional semantic spaces from lexical co-occurrence. Behav Res Meth Instrum Comput 28(2):203–208

Lund K, Burgess C (1997) Producing high-dimensional semantic spaces from lexical co-occurrence. Behav Res Meth Instrum Comput 28:203–208

Manning CD, Raghavan P et al (2007) Introduction to information retrieval. Cambridge University Press, Cambridge

Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. 25th International Conference on Software Engineering (ICSE 2003). ACM/IEEE, Portland, OR, USA

Marcus A, Feng L et al (2003) Comprehension of software analysis data using 3D visualisation. 1st IEEE International Workshop on Program Comprehension (IWPC'03). IEEE Computer Society Press, Portland, OR, USA

Marcus A, Sergeyev A et al (2004) An information retrieval approach to concept location in source code. 11th Working Conference on Reverse Engineering (WCRE 2004). Delft, The Netherlands.

Merlo E, McAdam I et al (2003) Feed-forward and recurrent neural networks for source code informal information analysis. J Softw Maint Evol Res Pract 15(4):205–244

Murphy GC, Kersten M et al (2006) How are java software developers using the eclipse IDE? IEEE Softw 23(4):76–83

Nemenyi PB (1963) Distribution-free multiple comparisons. PhD thesis, Princeton University

Poshyvanyk D, Marcus A (2007) Combining formal concept analysis with information retrieval for concept location in source code. 15th IEEE International Conference on Program Comprehension (ICPC '07)

Poshyvanyk D, Marcus A et al (2006a) JIRiSS—an eclipse plug-in for source code exploration. 14th IEEE International Conference on Program Comprehension (ICPC 2006). Athens, Greece.

Poshyvanyk D, Marcus A et al (2006b) Combining probabilistic ranking and latent semantic indexing for feature identification. 14th IEEE International Conference on Program Comprehension (ICPC 2006). IEEE Computer Society Press, Athens, Greece

Rajlich V, Wilde N (2002) The role of concepts in program comprehension. 10th International Workshop on Program Comprehension, (IWPC 2002). IEEE Computer Society Press, Paris, France

Robillard MP (2003) Representing concerns in source code. The University of British Columbia

Rohde D (2007) SVDLIBC. http://tedlab.mit.edu/~dr/SVDLIBC/

Salton G (1989) Automatic text processing the transformation analysis and retrieval of information by computer. Addison-Wesley, Reading, MA

Salton G, Buckley C (1987) Term weighting approaches in automatic text retrieval. Cornell University, NY, USA

Salton G, Wong A et al (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620

Sayyad-Shirabad J, Lethbridge TC et al (1997) A little knowledge can go a long way towards program understanding. Fifth International Workshop on Program Comprehension (IWPC '97). IEEE Computer Society Press, Dearborn, MI, USA

Schneidewind N, Kitchenharn B et al (1999) Resolved: software maintenance is nothing more than another form of development. IEEE International Conference on Software Maintenance (ICSM '99). IEEE Computer Society Press, Oxford, UK

Shepherd D, Fry Z et al (2007) Using natural language program analysis to locate and understand action-oriented concerns. International Conference on Aspect Oriented Software Development (AOSD'07)

Simonyi C (2005) Intentional programming. www.intentionalsoftware.com

Song D, Bruza P (2001) Discovering information flow suing high dimensional conceptual space. Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, New Orleans, LA, USA, pp 327–333

Song D, Bruza P (2003) Towards context-sensitive information inference. J Am Soc Info Sci Technol (JASIST) 4(54):321–334

TREC (2007) Text RETrieval Conference. http://trec.nist.gov/

Wilde N, Scully MC (1995) Software reconnaissance: mapping program features to code. J Softw Maint Res Pract 7(1):49–62

Wilde N, Page H et al (2001) A case study of feature location in unstructured legacy Fortran code. 5th European Conference on Software Maintenance and Reengineering (CSMR 01). IEEE Computer Society Press, Lisbon, Portugal

Zayour L, Lethbridge TC (2001) Adoption of reverse engineering tools a cognitive perspective and methodology. 9th International Workshop on Program Comprehension (IWPC 01). IEEE Computer Society Press, Toronto, Canada

Zhai C, Lafferty J (2004) A study of smoothing methods for language models applied to information retrieval. ACM Trans Info Syst 22(2):179–214

Zhao W, Zhang L et al (2004) SNIAFL: towards a static non-interactive approach to feature location. International Conference on Software Engineering (ICSE 04). ACM/IEEE, Edinburgh, Scotland

Zimmermann T (2006) Knowledge collaboration by mining software repositories. 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KSCD2006). IEEE/ACM, Tokyo, Japan

Dr. Brendan Cleary is currently a Research Fellow at the Enterprise Research Centre in the University of Limerick where his work focuses on novel applications of recommendation systems in education. Brendan graduated from the University of Limerick in 2003 with a first class honours degree in computer systems. He received an advanced scholar's scholarship to pursue a Ph.D. in the area of source code analysis and software comprehension in 2003. In 2007 Brendan received his Ph.D. from the University of Limerick. His research interests include; software comprehension, software visualisation, domain specific languages, legacy application modernisation, source code analysis and information retrieval.

Dr. Chris Exton is currently a lecturer in the department of Computer Science and Information Systems at University of Limerick. He has worked extensively in the commercial software development field in a variety of different institutions including software houses, manufacturers and food retailers. His last industrial appointment before embarking on his Ph.D. was as a senior technical consultant with ANZ Bank, Australia for a period of three years. His work in industry has included Software Engineering positions in Australia, Ireland and the UK, where he worked in companies as diverse as Ashling Microsystems Limerick, and Coca-Cola, London. In addition to this wealth of industrial, software-development experience he has completed a Ph.D. in the area of Distributed Systems at Monash University, Melbourne, Australia.

Dr. Jim Buckley obtained an Honours B.Sc. degree in Biochemistry from the University of Galway in 1989. In 1994 he was awarded an M.Sc. degree in Computer Science from the University of Limerick and he followed this with a Ph.D. in Computer Science from the same University in 2002. He currently works as a lecturer in the Computer Science and Information Systems Department at the University of Limerick, Ireland. His main research interests are in theories of information seeking, software reengineering and software maintenance. In this context, he has published actively at many peer-reviewed conferences / workshops and is currently a Visiting Scientist with the IBM Centre for Advanced Studies in Dublin. He has also co-organized a number of international conferences and workshops in this domain. He currently coordinates 2 research projects at the University: both in the area of software information seeking.

Dr. Michael English is a Junior Lecturer in the CSIS Department, University of Limerick. He received a B.Sc. Hons. in Mathematics and Statistics from University College Cork in 1996. In 1999 he received an M.Sc. in Computer Science from the University of Limerick. He received his Ph.D. from the University of Limerick in 2007. His Ph.D. thesis title was "An Analysis of Coupling Mechanisms in C++ Software using Fine-Grained Software Metrics". His research interests are in the areas of software metrics, software quality, software evolution, software refactoring and the analysis of source-code He has published his research in a number of peer-reviewed international forums.