A Java Reuse Repository for Eclipse using LSI

Ming-Yang (Jerry) Lin†, Robert Amor‡, Ewan Tempero‡
Department of Computer Science
University of Auckland
Auckland, New Zealand
†mhsi005@ec.auckland.ac.nz
‡{trebor,ewan}@auckland.ac.nz

Abstract

Software Reuse is a concept that is frequently mentioned as a way to improve software developers' productivity. However, there are a number of issues that need to be addressed in order for software reuse to be adopted by developers. One of those issues is providing enough reusable artifacts. The Java Standard API has been quite successful in this, with the latest version having over 3000 classes available. However this raises the issue of finding the right artifact to reuse. With the Java API, this means trawling through the JavaDoc webpages, which has the risk of not being able to find the right artifact, even though it is in the API. In this paper, we explore the use of latent semantic indexing *as a means to index the Java API JavaDoc pages. Specifically, we describe* **Prophecy***, an Eclipse plug-in that presents the Java API as a* software repository.

Keywords: Code Reuse, Software Repositories, Latent Semantic Analysis

1 Introduction

Software Reuse has been proposed as a solution to the software crisis since McIlroy [16]. The basic idea is that software developers use *reusable components* found in *software repositories* to reduce the amount of code that has to be written and so increase productivity. The success of a repository depends on the number of components it contains; if a repository has too few components, then a developer is unlikely to get much benefit from it. However repositories with many components face a different problem — how a developer finds the particular component she needs, or, failing to find something relevant, how to be sure that this is because there is nothing relevant in the repository. In this paper, we present an approach to dealing with the problem of efficiently locating components in a large repository. The approach uses *latent semantic indexing*[1] and is provided as a plug-in to the Eclipse Integrated Development Environment.

The Java Standard API [18] makes large number of useful classes available to a Java programmer. To some extent it is beginning to suffer from its success in that it is sometimes difficult to find relevant components. The standard mechanism for finding components in the API is to browse the web-based documentation provided by the JavaDoc mechanism. This documentation is organised as a set of web pages, roughly one per class, in a hierarchy that corresponds to the java package structure of the classes. The package structure tends to group related classes, and so the developer can find relevant classes by using the structure to guide her browsing.

When the API consisted of 200 or so classes in its first release in 1996, the browsing mechanism was an acceptable approach because it was feasible to look at every class. However the most recent release (5.0) has approximately 3200 classes and there are many other APIs now available to developers. It is now less feasible to look at every class, and so developers must rely on the package organisation to guide their searches. However this structure is becoming unwieldy, and it's also not always organised as expected.

Our plan is to using indexing to improve the efficiency of finding relevant classes in the Java API. The question then is how to index and how to present the index to the developer. We have developed a tool called **Prophecy**. Prophecy uses two tactics to provide more efficient access to the Java API. We use Latent Semantic Indexing (often referred to as Latent Semantic Analysis) to do the indexing. This is a free-text indexing and retrieval mechanism that involves statistical probability and correlation between terms in documents to reduce the semantic distance between words. [1].

Prophecy is provided as a plug-in to the Eclipse Integrated Development Environment. Developers can describe the desired functionality in a comment in the source code, and Prophecy will provide a candidate list of classes from the Java API.

The rest of this paper is organised as follows. We be-

gin by exploring in more detail the issues relating to Software Reuse that are relevant to our research, and discuss the specific problem we are addressing in more detail. In section 3, we provide an overview of Latent Semantic Indexing. Section 4 presents Prophecy. We then describe studies we carried out investigating the effectiveness of Prophecy in section 5. We discuss our results in section 6, and finally present our conclusions in section 7.

2 Software Repositories

The notion of software reuse has been widely discussed since 1968, when McIlroy promoted the idea of off-the-shelf reusable components that have been fully tested and ready to be reused without further testing[16]. Since then, considerable research has been devoted to the subject. Even though software reuse has been widely believed to be a technique with great potential to increase software productivity as well as quality, it is still struggling to become standard practise [17].

Many approaches to reuse have been developed and explored. Most involve a *software repository*, a repository of software entities (often referred to as *components*) that can be considered for reuse in any development project[12]. Software repositories have tool support for managing components, allowing components to be added, updated, or removed, usually subject to some form of quality control. In particular, they provide tool support to allow a software developer to find, assess, and retrieve components for reuse.

The research in the software repository area has focused on such topics as: tools for identifying and constructing components [3], tools providing access to components [15], and tools to find relevant components [10, 11]. It is this last topic that is most relevant to our work. As discussed in the introduction, the larger the repository, the more likely there will be a component to suit a developer's needs, but the harder it will be to actually find that component.

There are a number of kinds of indexing schemes a repository system might use. These can be classified as *uncontrolled vocabulary* and *controlled vocabulary*. Controlled vocabulary systems restrict the terms available to do the indexing and to those making the queries. Examples of controlled vocabulary include hierarchical structures similar to the Dewey Decimal system, or predefined lists of properties or *facets* a component might have. As we will discuss below, this kind of indexing usually requires human intervention, making it expensive, and even so cannot always provide good quality.

Uncontrolled vocabulary places minimal restrictions on the terms available for indexing and retrieval. The indexing process is often automated by software based on documents that are available rather than employing human indexers, and so it potentially much cheaper to create than controlled vocabulary systems. Prophecy uses an uncontrolled vocabulary index system using the API Specification, or "JavaDoc" pages.

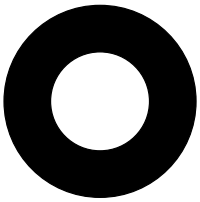
2.1 Indexing the Java Standard API

The Java Standard API is an extensive set of Java classes available to any Java developer. It is not a software repository as usually defined in the software reuse research community because it does not have the tool support normally associated with software repositories. It does have one characteristic of many repositories. The components are organised in a hierarchical fashion allowing developers to *browse* the set of available components. Developers can make reasonable guesses based on the hierarchy as to the location of what they are looking for.

Browsing a repository can be effective when all of its contents are in the "obvious" place and even if they aren't, if the the number of components is relatively small, or its content is already reasonably well known to the developer then it's still possible to effectively look at the whole repository to find something. But as the size of the repository gets bigger, not only is exhaustive search not practical, but it also becomes harder to organise its contents in a hierarchical manner so that everything is in the "obvious" place as there can be a number of different hierarchical organisations that might work. For example, in the Java API, it is not obvious where one would find something to format `double` as a string (it's neither in the `Double` class nor the `String` class). The goal of our research is to provide a search mechanism to the Java API, to allow developers to find relevant components more efficiently than through the use of browsing.

The usual approach to developing a reuse repository is to develop a tool set that manages the repository that includes the place where the components are stored, and means for adding, deleting, searching and retrieving of the components. Since the Java API comes already packaged in a Java distribution, repackaging it is both redundant and also interferes with the normal use of Java. An alternative would be to store *meta data* about the Java classes in the repository rather than the classes themselves, but this raises the question of how to capture and represent the data.

The key observation on which our work is based in that we already have quite a bit of useful information about the classes in Java API, namely their JavaDoc pages. We would like to extract what we need from these pages. This changes the problem to more that of an information retrieval problem. We index the JavaDoc pages so that queries for components become queries for particular documents. The conventional approach of retrieving textual information from a repository is lexical matching (often known as keyword matching) between the terms in user query and the terms



Student Responses

Ring
Doughnut
Tyre
Wheel
Circle

```
java.awt.Graphics.fillOval(int x, int y,
                           int width, int height)
```

Description

Fills an oval bounded by the specified rectangle with the current color.

Figure 1. The Vocabulary Problem

used to index documents.

Issues with performance in conventional free-text retrieval techniques can be broken down into two causes: synonymy and polysemy. Synonymy in a general sense is describing that there are words with similar or same meaning. For example, "hot" has synonym of "spicy", "warm", "eager" and "attractive" as English slang. Polysemy refers to the fact that one word can have multiple meanings. For example, the word "plant" can mean vegetation or machinery. The use of words to describe the desired component is based on different contexts, knowledge, needs, or linguistic habits of programmers. Programmers can therefore describe the same component differently. Furnas et al. measure the probability of two people choosing same keyword for a single well-known object is less than 20% [5].

Henninger showed the graphic shown in figure 1 to a group of students and asked them to name it as they would to a repository. Some of the responses are shown, and none of those given are keywords that appear in the description of the component one would use from the Java API to create the figure (also shown).

There are three major factors why conventional indexing and retrieval methods fail to overcome the problems due to synonymy and polysemy [4]. The first factor is the way index terms are identified incompletely. The terms used to describe or index documents are an abstraction that only contains a portion of the terms. The second factor is the lack of an automatic method to overcome the problem of polysemy. Approaches such as human intermediaries acting as translators, or using controlled vocabulary to solve the problem are too expensive and not very effective. The third factor is to do with the way conventional indexing and retrieval systems work. In such systems every term is treated independently from each other, the system omits the fact that there are correlations between terms that frequently appear together, and they are weighted as equally as ones that are rarely found together. This problem prohibits the user from using compound terms or long free text queries effectively in expanding or limiting a search [4]. Therefore, a suitable free-text indexing and retrieval mechanism for the purpose of this research has to include the semantic aspect of free-text information.

Others have addressed this problem. The closest to our work is by Ye [20, 22]. He developed CodeBroker, which provides retrieval of *methods* from the Java API. It uses *latent semantic analysis* (LSA) to provide free-text indexing, and augments the retrieval process by using *signature matching* to extract the more suitable components identified by the LSA process. CodeBroker is developed for Emacs. The questions we address in our work are, how effective is LSA, in particular how well does it work without augmentation, and is the method level the most effective level for component retrieval.

3 Latent Semantic Indexing

As described above, there is a strong motivation to allow programmers to search for reusable components through a description comprising keywords or normal text. However, simple free-text indexing and retrieval methods perform poorly with search terms requiring almost exact word matches to be effective.

Latent Semantic Indexing (LSI) [1] is a free-text indexing and retrieval mechanism that involves statistical probability and correlation between terms in documents. The basic idea being that a particular concept is described not just with a unique word, but with a distribution of words focused around the concept. If indexing and retrieval is based around matching concept distributions then the reliance on matching against a particular word is removed from the process. This approach should reduce the effect of synonyms as the match is not on a single word (which may have synonyms), but a distribution of words around that topic. This approach should also reduce the impact of polysemy as the distribution of words will differ for each unique meaning (e.g., for machine plant versus the organic plant). While a full description of LSI is beyond the scope of this paper a brief introduction to the technique is presented below.

LSI is based on the Vector Space Model (VSM), which is a technique commonly used in traditional Information Retrieval (IR) [9]. VSM represents a document as a vector of multiple dimensions, where each element of the vector represents the presence, or otherwise, of a keyword of interest to the domain. A query comprising free-text or keywords can similarly be represented as a multi-dimensional vector. If a query contains the same keywords as a given document, then the vectors for the document and query will be identical. Queries that have terms that aren't in the document will

have vectors that are further away from the document's vector. This means that the closeness of a query to a set of documents can be ranked according to the angle between the documents' vectors and that of the query, with those having the smallest angle considered to be the closest match to the query. In the VSM it is usual to choose a particular angle as the cut-off between relevant documents and non-relevant documents when ranking documents.

VSM requires an enormous vector space for most real problems, with the size driven by the number of keywords for the domain versus the number of documents to index. This vector is typically sparse and it is possible to apply statistical techniques to reduce the vector size while maintaining the most important relationships between terms in the documents. LSI assumes there are latent semantic relationships between terms and documents, and the relationships can be revealed by statistically analysing the overall word usage patterns across the whole document collection. LSI uses Singular Value Decomposition (SVD) [7] to reduce the term-document space, essentially merging dimensions. Since each dimension corresponds to a term in a document, the effect is as if the higher level concepts corresponding to the merged dimensions have been identified. Since SVD produces lower-dimension vectors that still retain many characteristics of the original vector, the argument is that the terms that are merged are mainly those that have a semantic relationship.

As can be imagined from the description of VSM and SVD the accuracy of these approaches is contingent upon careful choice of the parameters around which these methods are based. The number of keywords and their spread across the domain of interest are vital to the later retrieval results. Weighting schemes for keywords can also vary the results significantly. Local schemes look at the number of occurrences of a particular keyword in a document as a measure of its importance whereas global schemes provide weighting factors for each keyword up-front. When performing the SVD analysis the choice of the 'k' factor, representing the number of dimensions to be reduced down to, has a marked effect on the maintained dependencies. In order that the 'correct' values are derived during SVD a 'training set' of documents is usually established to provide a fair coverage of terms and concepts for the domain.

As an example of how LSI works, suppose we are searching for information in a LSI indexed database containing Java programming articles. If there are enough articles that contain all three words "String", "append", and "Stringbuffer", the LSI algorithm will reveal the correlation between these three terms in the article collection. A search query of "Append String" will find a set of articles containing these two terms, as well as articles that contain just the word "Stringbuffer".

4 Prophecy

4.1 Usage

As mentioned earlier, Prophecy is a plug-in to the Eclipse IDE[19]. The interface has been designed to be consistent with the overall Eclipse design and to be as minimalistic as possible.

There are four issues to consider when displaying retrieved reusable components: what information to present, how much to present, where to present it, and when to present it. Prophecy shows the class name and the beginning of the class description, since this is usually enough for developers to identify the relevancy of a class to the current requirements. The number of candidates to present is dependent on how the user has customised the IDE, but is typically 10. This information is presented in the Results View (a *view* is a standard concept in the Eclipse IDE) and located where other common views are found in the IDE (see figure 2). Programmers can also view the complete API specification of any chosen class, and the specification will be displayed in Browser View (figure 3).

Queries are made to the Java API by giving the requirements for the component as a Java multi-line comment. As the comment is completed (by typing `*/`) Prophecy passes the content of the comment to the search subsystem. The results of the search are then immediately presented in the Results View.

Figure 2 shows an example of this. Highlighted in the central panel is the query being made. Upon the completion of comment "change the file name", the candidate Java classes are shown in the Result View in the panel at the bottom. Developers can access the complete set of results using the arrows on the top right hand side corner of Result View.

Prophecy in fact provides two search mechanisms. As well as one based on LSI, it also provides one based on Google[8]. The user can choose which of the mechanisms is used for the search. As well as providing an alternative search mechanisms, it also allows us to make comparisons between conventional lexical keyword matching retrieval techniques and those based on LSI. We discuss this further in section 5.

As mentioned above, the intent of the design of Prophecy's user interface was to be consistent and minimalistic. This is achieved by incorporating the query mechanism into the code development activity, rather than having a separate dialog or similar mechanism. The results are provided in a non-obtrusive fashion, in that focus is not changed and users are not forced to acknowledge them before doing anything else (as with pop-up dialogs for example). The results remain until the next comment is completed, so providing context for the current development.

In the event that the class name and start of the descrip-

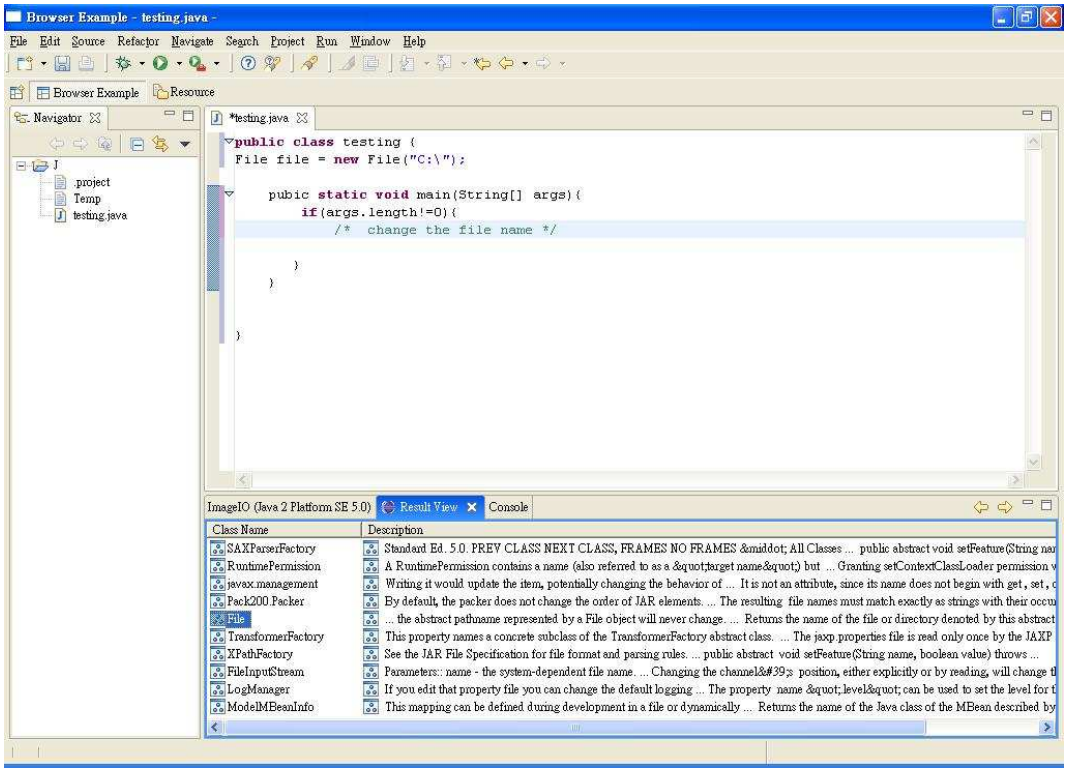


Figure 2. Queries made in comments (centre-right panel) yield a list of candidates (bottom-right panel).

tion is insufficient to identify the required component, the full API specification can be accessed. The Browser view is exactly the same as any web browsers that render HTML file except that it is implemented as an Eclipse view plug-in rather than used as an external program. The benefit of having a fully integrated environment is to prevent programmers from shifting their focus "too far away" from their current tasks. As shown in Figure 3, having the API specification displayed in the same environment allows programmers to immediately reflect what they've comprehended in the specification to the current task in the JavaEditor.

4.2 Design and Implementation

Prophecy involves several Eclipse plug-ins and utilises existing Eclipse interface components. The general software architecture of the Prophecy system is shown in figure 4. Prophecy consists of three subsystems: Eclipse, General Text Parser (GTP) and Google Web API.

Eclipse is a platform-independent open-source software framework written in Java. A major application of the framework is to provide an Integrated Development Environment (IDE) for Java (hereafter referred to as the Eclipse IDE). The Eclipse framework has a plug-in architecture. Three plug-ins (Result View, Browser View, and JavaEditor) have been created for Prophecy

General Text Parser (GTP) [6] is an existing implementation of LSI, and in this research GTP is adopted and slightly modified to construct the component repository described and perform query matching described in section 3. GTP is an integrated software package for creating data structures and the encoding needed for information retrieval models. The functionalities implemented in GTP include parsing ASCII files, creating term-by-document matrix with options of several local and global weighting scheme, producing k-dimensional space via matrix decompositions of the SVD, and performing query matching which returns a cosign-ranked list of documents. GTP is implemented in both C++ and Java, with the Java version used for Prophecy. GTP was developed and tested on a Linux platform, and so some minor modifications were made to make it run under Windows platform. GTP is also used to index the Java API specifications to create the Java Class Component Repository shown in figure 4. We explored several indexing schemes, as discussed in section 5.

Google Web APIs [8] is an open-source Java package

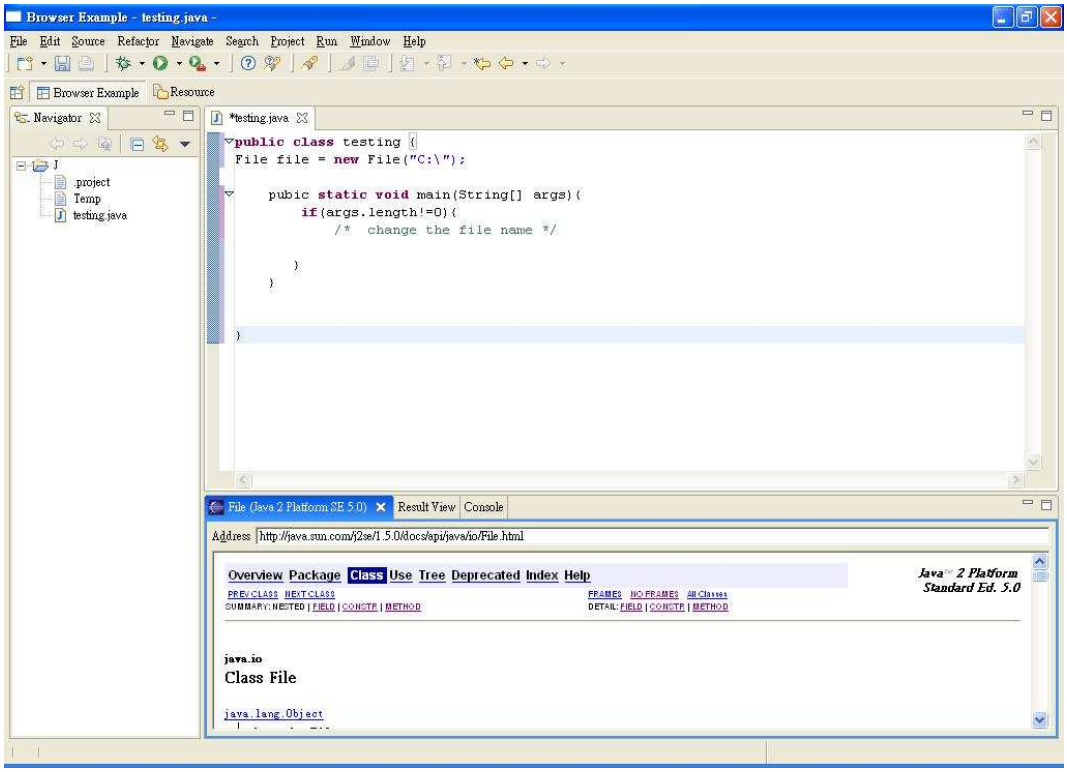


Figure 3. Figure Browser View. Displays the API Specification of selected Java class, File, from figure 2.

that enables developers to incorporate Google web search as a resource in their applications. Google Web APIs provides a programming interface to send query requests to Google's index and retrieve web pages in a structured data format that is easy for further processing.

JavaEditor is a part of the Java development environment provided with the Eclipse deployment. It includes a scanner that continuously scans the input to the editor to provide for such things as formatting and colouring code, providing completions, and so on. This has been modified to, on completion of a multi-line comment, send the contents of the comment to the designated search system, as discussed above.

Corpus ID	Corpus	#Terms	#Documents
C1	J2SE class documents	12797	6590
C2	C1 + extra training documents	23406	9578
C3	J2SE method documents	10224	45666
C4	C3 + extra training documents	22066	48654

Table 1. The four corpora

4.3 Indexing for Prophecy

In order to be used in Prophecy the Java API had to be indexed. Initially the API was preprocessed to generate a set of keywords for the LSI process (approximately 11,000 keywords were identified) and to create a document set to index. Within Prophecy two different types of document set were examined. One where each document consisted of all the text (ignoring markup tags and images) for a class, similar to what is presented in the Java API web pages, the other where each document consisted of the text of an individual method description. For each document set the normal stemming techniques and stopword removal was applied prior to creating vectors.

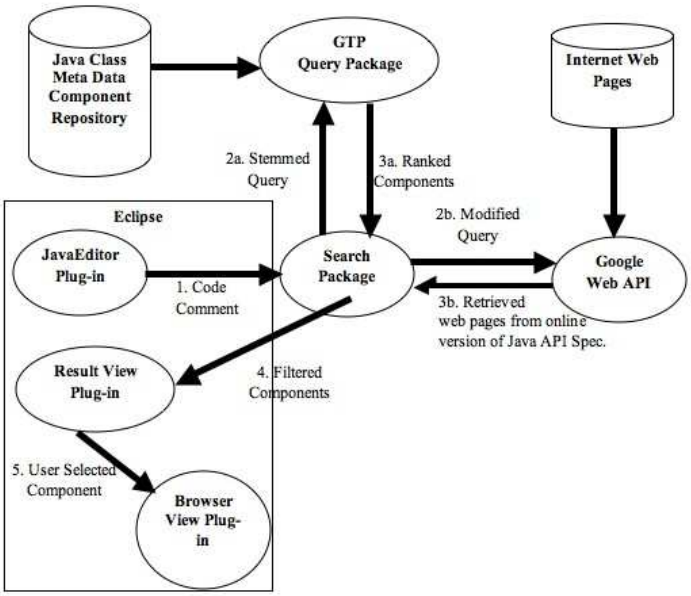


Figure 4. Software Architecture of Prophecy System

5 Evaluation Study

In this section we present a study we conducted to determine the effectiveness of our use of LSI. As mentioned earlier, the effectiveness of LSI depends on the number of dimensions that are reduced, the weighting that is used, and the training documents that are used. Our goal was to determine a solution that is computationally feasible and that yields acceptable retrieval results to support programmers finding reusable components.

There are a number of variables to consider: the LSI configuration (in particular the value of k), the type of search, and the corpora. We also compared the results from using LSI with those achieved using Google.

5.1 LSI Configuration

As discussed earlier, the effectiveness of LSI depends in particular on the amount of dimension reduction that is done, that is, the value of k. Finding the value of k is still an open research problem, with the value often determined by empirical testing [2]. Furthermore, the best value varies depending on the number of unique terms and documents in the corpus. As this was not the goal of our work, we choose to use the configuration used by Ye [21]. This also had the advantage of allowing us to compare our results with Ye's.

Ye's value of 328 was used, though this is outside the range of 100 to 300 often recommended [13] for LSI (though the value is often constrained for reasons of computational feasibility rather than retrieval accuracy). In this implementation no global weighting scheme was applied to the keywords but log-based local weighting was incorporated (weighting keywords with multiple occurrences higher than those with a single occurrence).

5.2 Query type

We considered two kinds of search query — free-text queries, which were the main target of our research, and keyword queries. Keyword queries were considered because experienced programmers might use these as shortcuts. They also allowed for a fairer comparison with Google, as discussed below.

We used 15 free-text queries, which included 8 from Ye's work (see table 2). The queries were chosen to be representative of the kind of free-text query a developer might make in the midst of writing code, and no attempt was made to "sanitise" them. From the free-text queries we created 18 keyword queries, with 3 free-text queries generating 2 queries each (table 3).

5.3 Corpora

For the training set, we considered the two document sets discussed earlier (one document per class versus one per method) and nothing else, and then those document set with extra documents added. These documents were: the

Query ID	Free Text Query	Desire Components (Java Class)	Rank	
			C1	C2
F1	judge if a file is a directory	java.io.File	33	3870
F2	split a string into substrings	java.lang.String	7	1244
F3	determine if it is a leap year	Java.util.GregorianCalendar	2	956
F4	return true if the string is capitalized	java.lang.Character	14	643
F5	change the file name	java.io.File	251	4159
F6	append two strings	java.lang.String	1066	2221
		java.lang.StringBuffer	2	2623
		java.lang.StringBuilder	1	4934
F7	create a directory on a floppy disk	java.io.File	5	5081
F8	create a random number between two numbers	java.lang.Math	324	6497
		java.util.Random	3	2046
F9	check if a character is a digit	java.lang.Character	69	308
F10	a listener for mouse movement	java.awt.event.MouseWheelListener	2	6341
F11	draw a circle fill with red color	java.awt.Graphics	24	1621
		java.awt.Color	2	2756
F12	establish a connection between two computers over internet	java.net.Socket	64	3547
F13	round a double to integer	java.lang.Math	21	969
F14	sort a long array into descending order	java.util.Arrays	59	3552
F15	create a application that runs on browser	java.applet.Applet	3	2443

Table 2. Ranking of relevant components retrieved from corpus C1 and C2 using free-text query.

Java 2 SDK 5.0 documentation, the Java virtual machine specification 2.0, the Java language specification 2.0 and a Java textbook.

In total, there are 4 corpora being examined (see Table 1): J2SE class documents, J2SE class documents and extra training documents, J2SE method documents, J2SE method documents and extra training documents. As can be seen in the Table 1, the training documents have approximately double the vocabulary of C1 and C3.

5.4 Results

We show the results only for C1 and C2 due to space limitations. Tables 2 and 3 show the results for free-text and keyword search respectively. The rank is the position the "desired component" in the list provided by the retrieval method, so smaller is better. For this document set, the desired component was considered to have been the best ranked method belonging to the same class. The best ranked method is not necessarily the one that could be reused for the task described by the query, but it takes programmers to the same class containing the desired method in Java API.

We also carried out the same queries using Google, by modifying the query to include `+"Standard E. 5.0" +site:java.sun.com` which then restricted the search to to the web pages of the Java API for J2SE 5.0 on the Java official website. The results are summarised below, with the full results available from [14], although these results must be viewed with some caution as they Google API does not appear to use the same data as the `google.com` website.

5.5 Analysis

There are a number of points of interest in the results shown above. The first is that despite the fact that the queries were taken as is without careful wording, a number of the free-text queries performed quite well. That said, quite a number of queries did not perform so well. If we assume that 10 candidates are displayed at a time in the Prophecy Results View, then not quite half of the queries would have the desired component appearing in the first page, and going two more pages would only add 3 more, and for some it is unlikely the developer would be patient enough to page though to them.

Next is the comparison with keyword queries. In fact most performed worse than the free-text queries they were derived from with several performing significantly worse.

Finally there are the very poor results with the C2 corpus,

Query ID	Free Text Query	Desire Components (Java Class)	Rank	
			C1	C2
K1	file directory	java.io.File	33	3870
K2.1	string split	java.lang.String	80	1397
K2.2	substring		9	1093
K3	leap year	Java.util.GregorianCalendar	2	1614
K4	capitalized String	java.lang.Character	8	488
K5	change file name	java.io.File	251	4159
K6	append string	java.lang.String	1600	2221
		java.lang.StringBuffer	2	2623
		java.lang.StringBuilder	1	4934
K7	create directory	Java.io.File	8	5091
K8	random number	java.lang.Math	130	4928
		java.util.Random	2	3981
K9.1	character digit	java.lang.Character	60	467
K9.2	digit		26	4428
K10	listener mouse movement	java.awt.event.MouseWheelListener	2	6341
K11	color circle	java.awt.Graphics	21	2317
		java.awt.Color	66	4561
K12	connection internet	java.net.Socket	62	5447
K13	round integer	java.lang.Math	21	639
K14	sort array descending	java.util.Arrays	20	4375
K15.1	application runs on browser	java.applet.Applet	3	1884
K15.2	application browser		1	2524

Table 3. Ranking of relevant Java components retrieved from corpus C1 and C2 using keyword query

with all queries performing spectacularly poorly.

Not surprisingly, Google did not perform so well with free-text search, often not returning any documents as those queries often used terms that didn't appear in any documents. Google did perform somewhat better than LSI for keyword search, with 12 queries having their desired component ranked in the first 10 and another 4 in the second 10 However 4 queries returned no results at all. We also tried synonym search with mixed results. In some cases it performed slightly better, and in others slight worse.

6 Evaluation

There are a number of complications to determining just how good Prophecy is. Measuring the effectiveness of an information retrieval method is usually done with recall (what proportion of relevant documents are retrieved) and precision (what proportion of the retrieved documents are relevant). The **similarity** value computed by LSI between query and document is a continuous value and so there is no obvious point at which a document is "not similar enough" to retrieve. Others (e.g., Ye [20]) set a specific threshold (either in similarity value or absolute number to retrieve). With Prophecy, it effectively returns everything, meaning recall is always 1 and precision is always very small. For this reason we have reported *rank*. This makes it difficult to compare to Ye's results, who provides recall and precision values. The values he reports are (for example), 100% recall gives about 13% precision.

Another point of difference is that in the results reported in tables 2 and 3 the "desired components" are *classes* not *methods*, whereas Ye performed indexing on a per-method basis. We also carried out the studies where the documents contained individual methods, but the results were uniformly poor (see [14] for details). This is perhaps not surprising. Documents of longer length generally have more semantic content and higher term frequency of semantically important keywords compared with shorter documents in the same corpus. However it does not explain why Ye's results appear so much better than ours.

We worked with Java SDK 5.0, whereas Ye worked with an earlier version. We also used a different set of extra training documents than Ye did (for example he included Linux on-line manuals) with a total 78,475 documents and 10,988 different terms. The expectation is that adding more training documents will reduce the semantic distance between

terms used to mean the same thing and so improve performance. It is difficult to determine what caused such poor performance. Perhaps the increase in the number of terms (roughly double that of C1) introduced too much noise, or perhaps the k value used was inappropriate for the C3 corpus. Again, our results are not consistent with Ye's. It is possible we should have tuned the LSI better, however in fact we had the best results with the values provided by Ye.

7 Conclusions

We have presented Prophecy, a tool that provides software repository support for the Java Standard API in Eclipse. Prophecy's user interface is intended to be consistent with the Eclipse look and feel and minimalistic in terms of how much it controls the user's focus.

We have tried to reproduce Ye's results from CodeBroker, but have not been able to do so, although we believe the results we have do suggest that Prophecy could be an effective tool with better tuning of the indexing. However the effort required to get good behaviour raises questions as to the effectiveness of LSI in this context.

Ultimately, the question we really want to answer is whether using Prophecy with LSI is faster than other means of finding appropriate components from the Java Standard API, such as just using a web browser and Google. For this, we need to perform empirical studies, which will be the subject of future work.

References

[1] M. W. Berry and M. Browne. *Understanding Search Enginers: Mathematical Modelling and Text Retrieval*. SIAM, 1999.

[2] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[3] R. Biddle and E. Tempero. Towards tool support for reuse. In M. Purvis, editor, *Software Engineering: Education and Practice '98*, pages 126–133, Dunedin, New Zealand, Jan. 1998. IEEE Computer Society.

[4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[5] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[6] J. Giles, L. Wo, and M. Berry. GTP (General Text Parser) software for text mining. In H. Bozdogan, editor, *Statistical Data Mining and Knowledge Discovery*, pages 455–471. CRC Press, Boca Raton, 2003. `http://www.cs.utk.edu/~lsi`.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1989.

[8] Google Web APIs. Develop your own applictionas using google. `http://www.google.com/apis/`, Retrieved May 5 2005.

[9] L. Gravano, H. García-Molina, and A. Tomasic. Gloss: text-source discovery over the internet. *ACM Transactions on Database Systems*, 24(2):229–264, 1999.

[10] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994.

[11] S. Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering Methodology*, 6(2):111–140, 1997.

[12] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[13] T. A. Letsche and M. W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Sciences*, 100(1-4):105–137, 1997.

[14] M. Y. Lin. Supporting software component reuse with active-retrieving component repository system. Master's thesis, University of Auckland, 2005.

[15] S. Marshall, R. Biddle, and J. Noble. A web user interface for an interactive software repository. In *Fifth Australasian User interface Conference*, pages 57–64, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[16] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, volume 1, pages 138–150. NATO Science Committee, Jan. 1969. Presented at the NATO conference on software engineering, Garmisch, Germany, 7-11 October, 1968.

[17] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[18] Sun Microsystems. Java 2 platform standard edition 5.0. api specification. `http://java.sun.com/j2se/1.5.0/docs/api/`, 2004.

[19] The Eclipse Foundation. The eclipse project. `http://www.eclipse.org`, 2005.

[20] Y. Ye. *Supporting component-based software development with active component repository systems*. PhD thesis, University of Colorado, 2001.

[21] Y. Ye. Personal communication, 2005.

[22] Y. Ye and G. Fischer. Promoting reuse with active reuse repository systems. In *6th International Conerence on Software Reuse*, pages 302–317, London, UK, 2000. Springer-Verlag.