Data Min Knowl Disc (2009) 18:300–336
DOI 10.1007/s10618-008-0118-x

Sourcerer: mining and searching internet-scale software repositories

Erik Linstead · Sushil Bajracharya · Trung Ngo · Paul Rigor · Cristina Lopes · Pierre Baldi

Received: 16 September 2007 / Accepted: 18 September 2008 / Published online: 15 October 2008
Springer Science+Business Media, LLC 2008

Abstract Large repositories of source code available over the Internet, or within large organizations, create new challenges and opportunities for data mining and statistical machine learning. Here we first develop Sourcerer, an infrastructure for the automated crawling, parsing, fingerprinting, and database storage of open source software on an Internet-scale. In one experiment, we gather 4,632 Java projects from SourceForge and Apache totaling over 38 million lines of code from 9,250 developers. Simple statistical analyses of the data first reveal robust power-law behavior for package, method call, and lexical containment distributions. We then develop and apply unsupervised, probabilistic, topic and author-topic (AT) models to automatically

Responsible editor: Eamonn Keogh.

Erik Linstead, Sushil Bajracharya, and Trung Ngo have contributed equally to this work.

E. Linstead · S. Bajracharya · T. Ngo · P. Rigor · C. Lopes · P. Baldi (✉)
Donald Bren School of Information and Computer Sciences, University of California, Irvine, USA
e-mail: pfbaldi@ics.uci.edu

E. Linstead
e-mail: elinstea@ics.uci.edu

S. Bajracharya
e-mail: sbajrach@ics.uci.edu

T. Ngo
e-mail: trungcn@ics.uci.edu

P. Rigor
e-mail: prigor@ics.uci.edu

C. Lopes
e-mail: lopes@ics.uci.edu

discover the topics embedded in the code and extract topic-word, document-topic, and AT distributions. In addition to serving as a convenient summary for program function and developer activities, these and other related distributions provide a statistical and information-theoretic basis for quantifying and analyzing source file similarity, developer similarity and competence, topic scattering, and document tangling, with direct applications to software engineering an software development staffing. Finally, by combining software textual content with structural information captured by our Code-Rank approach, we are able to significantly improve software retrieval performance, increasing the area under the curve (AUC) retrieval metric to 0.92– roughly 10–30% better than previous approaches based on text alone. A prototype of the system is available at: http://sourcerer.ics.uci.edu.

Keywords Mining software · Program understanding · Code search · Software analysis · Author-topic probabilistic modeling · Code retrieval

1 Introduction

Very large amounts of software source code are becoming available in large repositories and scattered across the Internet or, more privately, within large companies and other organizations. While these large amounts of software create new data mining challenges, they also create new data mining opportunities to better understand all aspects of software, from development to engineering, and to optimize software information retrieval (IR). Mining such repositories is important to understand software structure, function, complexity, and evolution, as well as to improve software IR systems and identify relationships between humans and the software they produce. Tools to mine source code for functionality, structural organization, team structure, and developer contributions are also of interest to private industry, where these tools can be applied to such problems as in-house code reuse and project staffing.

While some progress has been made in the application of statistics and machine learning techniques to mine and search software corpora, empirical studies have typically been limited to small collections of projects, often on the order of one hundred or less, one or several orders of magnitude smaller than publicly available Internet-scale repositories, where projects number in the thousands or more. Moreover, in most of these mining applications software code is treated as text. The same pure text-based approach holds for large-scale search engines, such as Google, that are commonly used to search for code over the Internet. However, efficient mining and searching of large amounts of software ought to take advantage not only of the textual aspect of source code, but also of its structural aspects, as well as any relevant metadata that surrounds it.

To improve the means by which code is mined and searched for structure, function, and developer contributions, we have developed Sourcerer, an infrastructure to collect, analyze, and search open source code at multiple levels with applications to, for instance, software reuse, bug triage, and project staffing. Sourcerer relies on a more complete analysis of code to extract not only pertinent textual information, but also structural and metadata information that can be used to improve the quality and performance of source code search, as well as augment the ways in which code can be analyzed in a manner not possible with textual content alone. By combining standard text IR techniques with source-specific heuristics and a relational representation of code, Sourcerer provides a comprehensive, multi-modal platform for searching and finding reusable software components, as well as a means to develop and apply machine learning techniques for a variety of tasks.

The remainder of the paper is organized as follows. For completeness, Sect. 2 presents the Sourcerer architecture in some detail, giving an overview of the code analysis process, which provides the data and foundation for all the subsequent analyses. It can be skipped by readers interested only in the main results. Section 3 briefly describes the data used in the experiments and analyzed at different levels of complexity in the following sections. Section 4 provides a simple statistical analysis of the code repository indexed by Sourcerer, including the identification of power-law behavior governing several entity and relation distributions. Section 5 develops and applies unsupervised topic and AT probabilistic modeling to automatically categorize source code and source code developers. As a byproduct, this approach also provides a novel way of defining and measuring topic scattering and document tangling in software engineering. Finally, turning to search problems, Sect. 6 demonstrates the use of fingerprints for structural searches and, more importantly, presents several ranking schemes for source code search and retrieval, as well as the methodology used in their comparison to determine how to best combine textual and structural information in code retrieval.

2 The Sourcerer infrastructure

The Sourcerer infrastructure comprises five major subsystems (Fig. 1): a system to crawl and manage software repositories, a system to parse and extract features from the code, a relational database to store the information, various tools to mine and search the database, and a Web-based graphical interface. In this section, we provide a short overview of the first three subsystems focusing on those elements (e.g. parsing, relational database, fingerprints) that are the most important for the data mining and IR applications which are the main focus of the paper (Sects. 4–6).

2.1 Overall architecture

Arrows in Fig. 1 depict the flow of information between the various components described below.

- *External code repositories*: these are the source code repositories available on the Internet. While the Sourcerer overall architecture is general and not specific to any particular programming language, the current prototype has been developed for software written in Java and gathered from a variety of sources. Although Sourceforge and Apache account for the majority of projects, we have also pulled in popular packages from less well-known archives (e.g. Colt and Weka, popular machine learning tools).

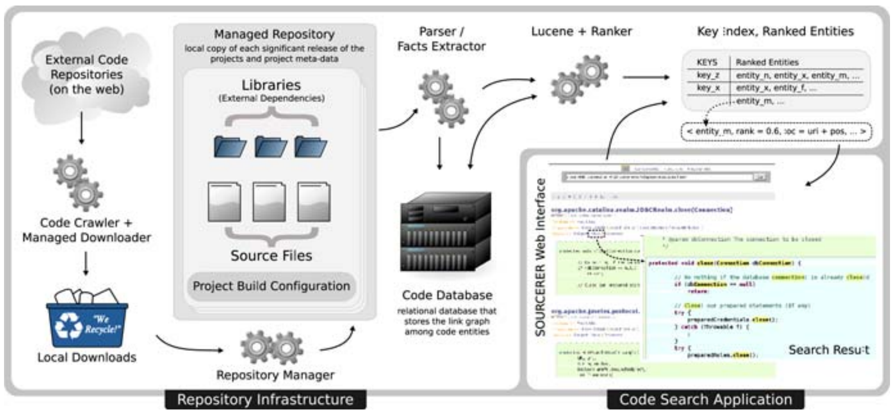
External Code Repositories (on the web)
Code Crawler + Managed Downloader
Local Downloads
"We Recycle!"
Managed Repository
local copy of each significant release of the projects and project meta-data
Libraries
(External Dependencies)
Source Files
Project Build Configuration
Repository Manager
Parser / Facts Extractor
Code Database
relational database that stores the link graph among code entities
Lucene + Ranker
Key index, Ranked Entities
KEYS
Ranked Entities
key_z
entity_n, entity_x, entity_m, ...
key_x
entity_x, entity_f, ...
entity_m, ...
< entity_m, rank = 0.6, loc = uri + pos, ... >
SOURCERER Web Interface
Search Result
Repository Infrastructure
Code Search Application

Fig. 1 Architecture of the Sourcerer infrastructure with five major subsystems: a system to crawl and manage software repositories, a system to parse and extract features from the code, a relational database to store the information, various tool to mine and search the database, and a Web-based graphical interface

- *Code crawlers and automated downloads*: Sourcerer includes several kinds of crawlers: some target well-known repositories, such as Sourceforge, while others act as web spiders that look for less well-known code archives found, for instance, in academic repositories and researchers' home pages. An automated dependency and version management component provides managed download of these repositories, keeping track of other libraries and projects required to build, which facilitates the parsing process.
- *Local code repository*: Sourcerer maintains a local copy of each significant release of the projects, as well as project specific meta-data. Code snapshots are stored in the file system on fast, local disks to improve retrieval performance as much as possible. The current version of Sourcerer does not keep track of incremental updates in an automated fashion, but is capable of indexing multiple versions of the same project. We are currently working on a plug-in to manage such updates without requiring web repositories to be recrawled completely.
- *Code database*: postgresSQL 8.0.1 provides a scalable and efficient platform in which to store the relational representation of the parsed source code entities and metadata. The database schema contains tables for entities (packages, classes, method, fields, etc) and the various relations among them. Since the overall performance of Sourcerer is heavily dependent on this database, care has been taken to optimize the ways in which the data is stored, though this is sometimes in conflict with best practices regarding normalized forms in schema design. More important than normalization, however, is the creation of secondary indices to support fast joins across tables.
- *Parser/feature extractor*: a specialized parser (Sect. 2.2) parses every source file from a project in the local repository and extracts entities and relations, as well as keywords and fingerprints (Sect. 2.3). These features are extracted in multiple passes and stored in the relational database and Lucene index.

- *Text search engine (Lucene)*: all entity keywords and metadata that require fast searching are indexed using Lucene 1.9.1 (http://lucene.apache.org), a high-performance text search engine library. Though Lucene comes with a rich API and class framework, some effort was spent designing custom analyzers and tokenizers to account for source-specific keyword conventions. To improve performance, we leverage Lucene's in-memory indexing capabilities, which when combined with the appropriate hardware, allow for faster entity indexing without the penalty of frequent disk access.
- *Ranker*: the ranker performs additional non text-based ranking of entities. The relations table from the code database is used to compute ranks for the entities using ranking techniques as discussed in Sect. 6. The ranker may also be run as an independent tool from the rest of the parsing and searching architecture, and depends only on the relational database. This allows one to tune and experiment with a wide variety of ranking schemes without incurring the cost of a full source code re-parse.

A custom task scheduler runs multiple instances of the last three components above (Parser + Lucene + Ranker) for parallel indexing of multiple repositories. We use a cluster of machines to parse projects individually, storing extracted information in a single database. Currently a Sunfire ×4100 machine with 2.6 Ghz Dual Core AMD Opteron Processor and 16 GB RAM is used for data storage and analysis. A latest instance of indexing 12,000 selected projects (4,632 with source code) amounting to more than 38 million SLOCs took about 2 hours to complete with this hardware setup, which indicates that performance should not be an issue as the repository scales up over time.

These components provide a general purpose infrastructure for indexing source code. The code search application described in Sect. 6 is built using the indexed keys and ranked entities. Query keywords entered by a user are matched against the set of keywords maintained using Lucene. Each key is mapped to a list of entities, and each entity has a rank associated with it. Each entity also has other associated information such as its location in the source code, version, and location in the local repository to fetch the source file from. Thus, the list of entities that are matched against the sets of matching keys can be returned as search results to the user with all this information attached in a convenient way through Sourcerer's Web-based graphical interface.

2.2 Parsing and relational database storage

With the major architectural components in place, the next critical step is the actual parsing of open source code so as to populate the database and Lucene index with searchable features. In order to leverage the structural and symbolic aspects of code for the code-search application, the parsing process must go beyond the simple tokenization employed by other code-search engines. While the use of a custom parser allows for the extraction of arbitrarily fine-grained information, performance and storage considerations impose some practical limitations on the type and quantity of data that Sourcerer can maintain.

Various storage models have been used to represent source code and each has its set of advantages and drawbacks (Cox et al. 1999). Given the scale of our system, we use a relational database to store the data. One direct benefit of using a relational model is that the structural information between program entities and their relations is explicitly laid out in the database. However, breaking down the model into finer levels of granularity can make querying inefficient, both in expression and execution. Having tables for each type of program entity, corresponding directly to a relational representation of the Abstract Syntax Tree would be inefficient for querying. To address these efficiency problems, Sourcerer uses a source model consisting of only two tables: (1) program entities; and (2) their relations.

- *Entities*: entities are uniquely identifiable elements from the source code. Declarations produce unique program entity records in the database. Program entities can be identified with a fully qualified name (FQN) or they can be anonymous. The list of entities that are extracted during the parsing steps are given in Appendix A. When the parser encounters any of these declarations it records an entry in the database assigning the following attributes to the entity: a FQN, document location in the local repository that associates version and location information (where declared) with the entity, position and length of the entity in the original source file, and a set of meta-data as name-value pairs. These include a set of keywords extracted from the FQN.
- *Relations*: any dependency between two entities is represented as a relation. A dependency d originating from a source entity s to a target entity t is stored as a relation r from s to t. The various types of relations extracted and stored by Sourcerer are also given in Appendix A.

2.3 Keywords and fingerprints

In addition, we also store compact representations of attributes for fast retrieval of search results, including keywords and fingerprints.

- *Keywords*: the parser extracts keywords from two different sources: FQNs and comments. Keyword extraction from FQNs is based on language-specific heuristics that follows the commonly practiced naming conventions in that language. For example, the Java class name "QuickSort" will generate the keywords "Quick" and "Sort". Keyword extraction from comments is done by parsing the text for natural and meaningful keywords, with common English stopwords being filtered and discarded. These keywords are then mapped to the entities that have unique IDs and that are close to the comment, for example the immediate entity that follows the comment. These keywords are also associated with the immediate parent entity inside which the comment appears. In addition to keywords, Javadoc tags can be used both heuristically (giving higher weight to comments that use Javadoc) and to infer additional relations from the code.
- *Fingerprints*: fingerprints are used in Sourcerer to support structural searches of source code. In some applications, one is interested in retrieving pieces of code with particular syntactical signatures irrespective of semantic aspects. For instance, for the purpose of testing a new compiler, one may be interested in retrieving pieces of

code containing three nested loops of a given length. Similarly, possible examples of race conditions in code can be generated by searching for snippets that exercise several wait or notify calls without the protection of a synchronized block. Thus we also extract and store syntactical signatures in the form of vectors, or fingerprints, where each component is associated with a particular numerical feature of the corresponding entity, such as the presence/absence or number of occurrences of a concurrency construct, or a maintainability metric (Oman and Hagemeister 1992; Welker and Oman 1995). This allows similarity between entities to be measured by the similarity of the corresponding fingerprint vectors using well-known techniques in IR (Baeza-Yates and Ribeiro-Neto 1999), such as cosine or Euclidean distance. Fingerprint technology is widely used in other domains, for instance in chemoinformatics (Chen et al. 2005; Swamidass and Baldi 2007), and have been applied to other software development tasks such as automated code completion (Hill and Rideout 2004) and source pattern matching (Paul 1992; Paul and Prakash 1994). Fingerprints are created for each code entity as the source is parsed, and stored along with other information in the database. A structure-based query needs only to refer to entity fingerprints to find relevant hits, and can thus avoid unnecessarily complicated SQL statement generation and the corresponding processing. Sourcerer currently uses three kinds of fingerprints corresponding to structure, type, and micro pattern (Gil and Maman 2005) occurrences. A detailed description of each fingerprint type is available in Appendix B.

3 Data

In order to assess the performance of our techniques it is first necessary to populate the code database with a suitable number of open source projects. Using the crawling infrastructure, we downloaded approximately 12,000 distinct projects, primarily from Sourceforge and Apache, and filtered out distributions packaged without source code (binaries only). The end result is a repository consisting of 4,632 projects, containing 244,342 source files, with 38.7 million lines of code (LOC), written by 9,250 developers. Parsing the multi-project repository described above yields a repository of over 5 million entities organized into 48 thousand packages, 560 thousand classes, 3.2 million methods, and participating in over 23.4 million relations which form the building blocks of the graph-based ranking algorithms described in Sect. 6. Table 1 shows the counts of specific entities considered by our parser.

Table 1 Type and number of entities extracted

Entity type	Count
Package	47,640
Class	560,669
Interface	49,093
Method	3,205,741
Field	1,699,205

4 Statistical analyses of source code

To gain further insight into the nature of these projects, we developed an analysis module to automatically extract relevant statistics, by leveraging the query capabilities of the Sourcerer database. While the statistics extracted by this module are too numerous to present in totality, here we describe a sample of pertinent results.

If Sourcerer is to be successful as a search engine and empirical platform, it must be capable of parsing projects of all sizes. Figure 2 shows the distributions of SLOCs, files, classes, and methods by project. Summary statistics are reported in Table 2 for entities indexed by Sourcerer. In addition to many small and medium sized projects, it can be seen that the repository contains a sufficient number of very large projects so as to validate the scalability of the architecture to projects of all sizes.

In examining the results produced by the statistic module, we found evidence of robust power-law behavior. Power-law distributions have been discovered in a

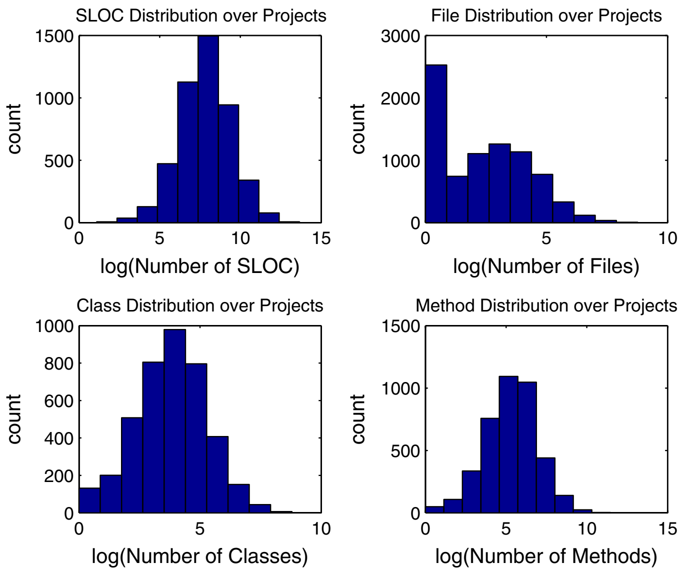

Fig. 2 Distributions of SLOC, files, classes, and methods

Table 2 Selected summary statistics (per Project)

	Max	Median	Mean	Standard deviation
Files	6,415	10	55.28	182.98
Lines of code	857,308	2,529.50	8,368.92	24,687
Packages	570	5	10.98	23.49
Classes	6,599	47	126.10	290.68
Methods	94,654	216	695.35	2,353
Fields	21,867	117	339.02	820.80

variety of areas including physics, geology, biology, computer science, linguistics, and economics, as well as in the structure of the internet and social networks (see Zipf 1932; Baldi et al. 2003; Mitzenmacher 2003 for examples and reviews of power-law distributions). Software is no exception, and power-laws have been reported for several phenomena in both procedural and object-oriented programming (Knuth 1971; Wheeldon and Counsell 2003; Concas et al. 2007). Java is a relatively new programming language, however, and little work has been done to verify the occurrence of power-laws on the large scale. Previously the largest such study consisted of 56 Java projects (Frean et al. 2006), compared to the 4,600 parsed by our infrastructure. Power-law behavior of the form $y = Cx^{\alpha}$ can be identified in a log-log plot by a line with slope α. Here we find power-law distributions for package, method call, and inside relation distributions, as illustrated by the log-log plots in Figs. 3–5, ignoring the usual boundary effects. In these figures, the y axis corresponds to the log of the counts of a given attribute (number of packages, method calls, and inside relations) being measured for each project, and the x axis corresponds to the log of the rank of the projects, ranked by decreasing counts.

In addition to these statistics focused on the *structure* of open source projects, we also collect data as to the use of the Java language itself. For example, Table 3 contains the frequencies of Java keywords across all 4,632 projects. Upon examining this data we can see that the 'default' keyword occurs about 6% less frequently than

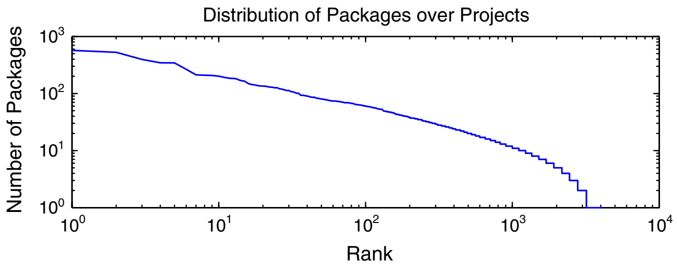

Fig. 3 Power-law distribution for packages over projects

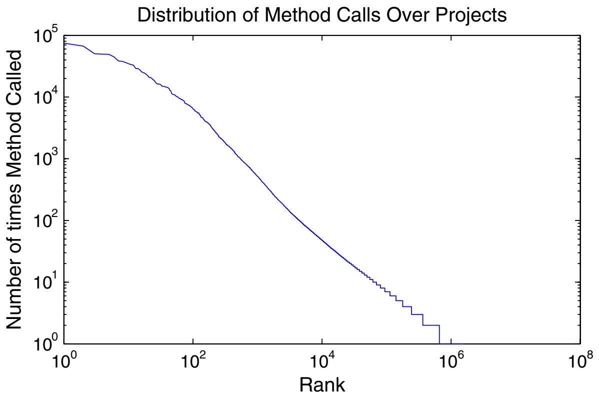

Fig. 4 Power-law distribution for method calls over projects

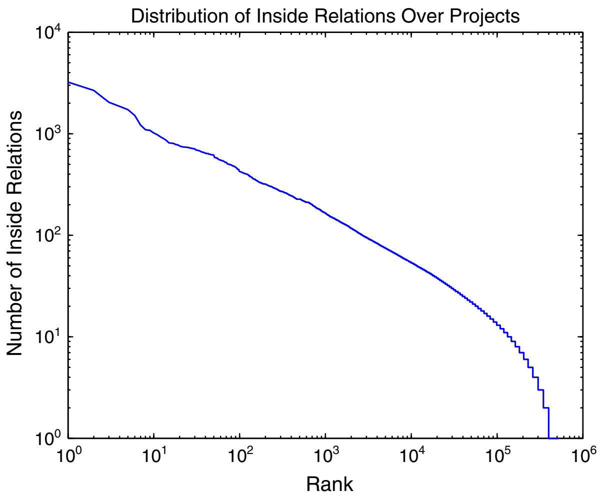

Fig. 5 Power-law distribution for inside relations over projects

the ‘switch’ keyword, despite the fact that best practice typically mandates all switch statements contain a default block. Moreover, we can also see that the ‘for’ loop is about twice as pervasive as the ‘while’ loop, suggesting that the bound on the number of iterations is more likely to be known or based on the size of a known data structure. With such information we hope we will be able to learn as much about *how* open source developers create their programs as we can about the programs themselves. In a process-driven development environment, especially common in industry, even simple statistical analysis of source code can provide useful insight as to the everyday practice of developers. For example, one can consider the case of error handling, rules for which are often provided in work instructions for software development projects. Based on the statistics in Table 3 it can be seen that the occurrence of the “try” keyword is 1.22%, whereas the occurrence of the “catch” keyword is only 1.33%. Given the rich Java exception handling mechanism, as well as the ability to define inheritance hierarchies, it would appear many developers use only a single catch block per try block (likely catching only general exceptions rather than dealing with specific cases). Such information is of practical use to an organization when assessing work instruction compliance, which can have far-reaching consequences from a project management perspective.

5 Topic and author-topic probabilistic modeling of source code

Automated topic and AT modeling has been successfully used in text mining and IR where it has been applied, for instance, to the problem of summarizing large text corpora. Recent techniques include latent Dirichlet allocation (LDA), which probabilistically models text documents as mixtures of latent topics, where topics correspond to key concepts presented in the corpus (Blei et al. 2003). AT modeling is an extension of topic modeling that captures the relationship of authors to topics in addition to

Table 3 Frequency of java keyword occurrence

Keyword	Percentage	Keyword	Percentage
Public	12.53	This	0.89
If	8.44	Break	0.85
New	8.39	While	0.63
Return	7.69	Super	0.57
Import	6.89	Instanceof	0.56
Int	6.54	Double	0.55
Null	5.52	Long	0.54
Void	4.94	Implements	0.43
Private	3.66	Char	0.30
Static	3.16	Float	0.28
Final	3.01	Abstract	0.25
Else	2.33	Synchronized	0.25
Throws	2.16	Short	0.20
Boolean	2.12	Switch	0.19
False	1.69	Interface	0.17
Case	1.60	Continue	0.15
True	1.60	Finally	0.14
Class	1.36	Default	0.13
Protected	1.33	Native	0.08
Catch	1.33	Transient	0.06
For	1.22	Do	0.05
Try	1.22	Assert	0.03
Throw	1.16	Enum	0.02
Package	0.96	Volatile	0.004
Byte	0.93	Strictfp	2.49E-06
Extends	0.89		

extracting the topics themselves. An extension of LDA to probabilistic AT modeling has been developed in (Steyvers et al. 2004) based on the distribution of authors over topics in addition to topics over documents. In the literature (Newman et al. 2006) these more recent approaches have been found to produce better results than more traditional methods such as latent semantic analysis (LSA) (Deerwester et al. 1990) (see also Buntine 2005; Blei and Lafferty 2006).

Past applications of these methods, however, have typically been limited to traditional text corpora such as academic publications, news reports, emails, and historical documents (Rosen-Zvi et al. 2004; Newman and Block 2006). While attempts have been made to analyze source code based on concepts, the techniques employed have either required manual definition of topics (Ugurel et al. 2002), or have been restricted to latent semantic analysis (Marcus et al. 2004; Kuhn et al. 2007). Recently LDA has been applied to log traces of program execution (Andrzejewski et al. 2007), providing a framework for statistical debugging, but has not been leveraged for analysis of actual source code, despite the many possibilities for doing so. Though vocabulary, syntax, and conventions differentiate a programming language from a natural language, the tokens present in a source file are still indicative of its function (ie. its topics). Thus here we develop and apply LDA probabilistic models for authors and topics to software data in order to mine function and developer contributions directly from source code.

5.1 Latent Dirichlet allocation (LDA) and author-topic (AT) models

In AT models for text, the data consists of a set of documents. The authors of each document are known and each document is treated as a "bag of words". Let A be the total number of authors, D the total number of documents, W the total number of distinct words (vocabulary size), and T the total number of topics present in the documents. While non-parametric Bayesian (Teh et al. 2006) and other (Griffiths and Steyvers 2004) methods exist to try to infer T from the data, here we assume that T is fixed (e.g. $T = 100$). Different values of T are explored in the simulations.

As in (Rosen-Zvi et al. 2004), the model assumes that each topic t is associated with a multinomial distribution $\phi_{\bullet t}$ over words w, and each author a is associated with a multinomial distribution $\theta_{\bullet a}$ over topics. More precisely, the parameters of the model are given by two matrices: a $T \times A$ matrix $\Theta = (\theta_{ta})$ of AT distributions, and a $W \times T$ matrix $\Phi = (\phi_{wt})$ of topic-word distributions. Given a document d containing N_d words with known authors, in generative mode each word is assigned to one of the authors a of the document uniformly, then the corresponding $\theta_{\bullet a}$ is sampled to derive a topic t, and finally the corresponding $\phi_{\bullet t}$ is sampled to derive a word w. A fully Bayesian model is derived by putting symmetric Dirichlet priors with hyperparameters α and β over the distributions $\theta_{\bullet a}$ and $\phi_{\bullet t}$. So for instance the prior on $\theta_{\bullet a}$ is given by

$$D_\alpha(\theta_{\bullet a}) = \frac{\Gamma(T\alpha)}{(\Gamma(\alpha))^T} \prod_{t=1}^{T} \theta_{ta}^{\alpha-1}$$

and similarly for $\phi_{\bullet t}$. If $\mathcal{A}$ is the set of authors of the corpus and document d has A_d authors, it is easy to see that under these assumptions the likelihood of a document is given by:

$$P(d|\Theta, \Phi, \mathcal{A}) = \prod_{i=1}^{N_d} \frac{1}{A_d} \sum_{a} \sum_{t=1}^{T} \phi_{w_i t}\theta_{ta}$$

which can be integrated over ϕ and θ and their Dirichlet distributions to get $P(d|\alpha, \beta, \mathcal{A})$. The posterior can be sampled efficiently using Markov Chain Monte Carlo Methods (Gibbs sampling) and, for instance, the Θ and Φ parameter matrices can be estimated by MAP (Maximum A Posteriori) or MP (Mean Posterior) estimation methods.

In the case where the author component is not needed, one can apply a basic LDA algorithm to approximate document-topic distributions instead of AT distributions.

This provides a basis for software function analysis, as well as a means for comparing source file similarity, document tangling, and document scattering, as described below.

5.2 Processing of source code

Once the data is obtained, applying topic models to software requires the development of several tools to facilitate the processing and modeling of source code. In addition to the crawling infrastructure described above, the primary functions of the remaining tools are to extract and resolve author names from source code, as well as convert the source code to bag of words format for the AT algorithm.

The author-document matrix is produced from the output of our author extraction tool. It is a binary matrix where entry $[i, j] = 1$ if author i contributed to document j, and 0 otherwise. While extraction of author information from a corpus of, for instance, scientific papers, is often straightforward automatic extraction of author information for software files is complicated by several matters. A best practice of software engineering, adhered to in the open source community, is to embed useful metadata in code comments, such as file name, author name, version number, and other similar data. Thus in principle extracting author information then becomes a matter of tokenizing the code and associating developer names with file (document) names when this information is available. For Java software, this process ought to be further facilitated by the prevalence of Javadoc tags which present this metadata in the form of attribute-value pairs. However, exploratory analysis of the Eclipse 3.0 code base shows that most source files are credited to “The IBM Corporation” rather than specific developers. Therefore, to generate a list of authors for specific source files, we had to resort to the Eclipse bug data available in (Schröter et al. 2006). Parsing the provided XML files for version 3.0 allows us to derive a list of contributing developers, as well as the source files modified by each developer.

While leveraging bug data is necessary to generate the developer list for Eclipse 3.0, it is also desirable to develop a more flexible approach that uses only the source code itself, and not other data sources. Thus to extract author names from source code we also develop a lightweight parser that examines the code for Javadoc ‘@author’ tags, as well as free form labels such as ‘author’ and ‘developer.’ Occurrences of these labels are used to isolate and identify developer names. An attempt is made to identify lists of developers as well, extracting all names in the list. In the case where HTML is used to format author email addresses or web pages, the tags are processed and identifiers stripped out. Ultimately author identifiers may come in the form of full names, email addresses, url’s, or CVS account names. This multitude of formats, combined with the fact that author names are typically labeled in the code header, is key to our decision to extract developer names using our own parsing utilities, rather than part-of-speech taggers (Brill 1994) leveraged in other text mining projects.

Complicating author name extraction is the fact that the same developer may write his name in several different ways. For example, “John Q. Developer” alternates between “John Developer,” “J. Q. Developer,” or simply “Developer.” To account for this effect, we implement also a two-tiered approach to name resolution using the q-gram algorithm (Ukkonen 1992; Navarro 2001). Author names are first resolved on a per-project basis, using similarity threshold $t1$, to produce a global author list. The global list of names is then resolved with similarity threshold $t2$, such that $t2 > t1$. In practice, we find that setting $t1 = .65$ and $t2 = .75$ gives reasonable results, and author extraction on the multi-project repository yields 9,250 distinct author names.

With author name extraction complete, a key remaining task is the conversion of the source code into compatible representations for the AT algorithm, required as input parameters. Of these parameters, the most important are the author-document matrix and the word-document matrix. We have seen how the author-document matrix can be generated by assigning author to documents. To produce the word-document matrix for our input data, representing the occurrence of words in individual documents, we have developed a comprehensive tokenization tool tuned to the Java programming language. This tokenizer includes language-specific heuristics that follow the commonly practiced naming conventions. For example, the Java class name "QuickSort" will generate the words "quick" and "sort". All punctuation is ignored. As an important step in processing source files our tool removes commonly occurring English stop words as well as class names from the Java SDK in order to specifically avoid extracting common topics relating to the Java collections framework.

The LDA-based AT algorithm is run on the input matrices with additional input parameters specifying that 100 topics (a number determined from experimentation) should be extracted from the code. The number of iterations, i, to run the algorithm is determined empirically by analyzing results for i ranging from 500 to several thousands. The results presented in the next section are derived using 3,000 iterations, which were found to produce interpretable topics in a reasonable amount of time (a week or so). Because the algorithm contains a stochastic component we also verified the stability of the results across multiple runs. In total, for the larger repositories, the process of parsing and topic modeling requires several days to run to completion on a single Sun SunFire X2200 M2 Server, using two dual-core AMD Opteron processors along with 8GB of RAM.

As output, the algorithm produces a document-topic matrix or AT matrix specifying the number of times a document or author was assigned to each of the 100 topics extracted from the code. The topics themselves are defined by representative words from the corpus.

5.3 Applying author-topic modeling to source code

A representative subset of 6 AT assignments extracted via AT modeling on the selected 2,119 source files from Eclipse 3.0 is given in Table 4. Each topic is described by several words associated with the topic concept. To the right of each topic is a list of the most likely authors for each topic with their probabilities. Examining the topic column of the table it is clear that various functions of the Eclipse framework are represented. For example, topic 1 clearly corresponds to unit testing, topic 2 to debugging, topic 4 to building projects, and topic 6 to automated code completion. Remaining topics range from package browsing to compiler options.

Table 5 presents 6 representative AT assignments from the multi-project repository. This dataset yields a substantial increase in topic diversity. Topics representing

Table 4 Representative topics and authors from Eclipse 3.0

#	Topic	Author probabilities	#	Topic	Author probabilities
1	Junit	Egamma 0.97065	4	Nls-1	Darins 0.99572
	Run	Wmelhem 0.01057		Ant	Dmegert 0.00044
	Listener	Darin 0.00373		Manager	Nick 0.00044
	Item	Krbarnes 0.00144		Listener	Kkolosow 0.00036
	Suite	Kkolosow 0.00129		Classpath	Maeschli 0.00031
2	Target	Jaburns 0.96894	5	Type	Kjohnson 0.59508
	Source	Darin 0.02101		Length	Jlanneluc 0.32046
	Debug	Lbourlier 0.00168		Names	Darin 0.02286
	Breakpoint	Darins 0.00113		Match	Johna 0.00932
	Location	Jburns 0.00106		Methods	Pmulet 0.00918
3	Ast	Maeschli 0.99161	6	Token	Daudel 0.99014
	Button	Mkeller 0.00097		Completion	Teicher 0.00308
	Cplist	Othomann 0.00055		Current	Jlanneluc 0.00155
	Entries	Tmaeder 0.00055		Identifier	Twatson 0.00084
	Astnode	Teicher 0.00046		Assist	Dmegert 0.00046

Table 5 Representative topics and authors from the multi-project repository

#	Topic	Author probabilities	#	Topic	Author probabilities
1	Servlet	Craig r Mcclanahan 0.19147	4	File	Adam Murdoch 0.02466
	Session	Remy Maucherat 0.08301		Path	Peter Donald 0.02056
	Response	Peter Rossbach 0.04760		Dir	Ludovic Claude 0.01496
	Request	Greg Wilkins 0.04251		Directory	Matthew Hawthorne 0.01170
	Http	Amy Roh 0.03100		Stream	lk 0.01106
2	Sql	Mark Matthews 0.33265	5	Token	Werner Dittmann 0.09409
	Column	Ames 0.02640		Key	Apache software foundation 0.06117
	Jdbc	Mike Bowler 0.02033		Security	Gert van Ham 0.05153
	Type	Manuel Laflamme 0.02027		Param	Hamgert 0.05144
	Result	Gavin King 0.01813		Cert	Jcetaglib.sourceforge.net 0.05133
3	Packet	Brian Weaver 0.14015	6	Service	Wayne m Osse 0.44638
	Type	Apache directory project 0.10066		Str	Dirk Mascher 0.07339
	Session	Opennms 0.08667		Log	David Irwin 0.04928
	Snmpwalkmv	Matt Whitlock 0.06508		Config	Linke 0.02823
	Address	Trustin Lee 0.04752		Result	Jason 0.01505

major sub-domains of software development are clearly represented, with the first topic corresponding to web applications, the second to databases, the third to network applications, and the fourth to file processing. Topics 5 and 6 are especially interesting, as they correspond to common examples of crosscutting concerns from aspect-oriented programming (Kiczales et al. 1997), namely security and logging. Topic 5 is also demonstrative of the inherent difficulty of resolving author names, and the shortcomings of the q-gram algorithm, as the developer "gert van ham" and the developer "hamgert" are most likely the same person documenting their name in different ways.

Examining the author assignments for the various topics provides a simple means by which to discover developer contributions and infer their competencies. It should come as no surprise that for Eclipse the most probable developer assigned to the JUnit framework topic is "egamma", or Erich Gamma. In this case, there is a 97% chance that any source file in our dataset assigned to this topic will have him as a contributor. Based on this rather high probability, we can also infer that he is likely to have extensive knowledge of this topic. This is a particularly attractive example because Erich Gamma is widely known within the software engineering community for being a founder of the JUnit project, a fact which lends credibility to the ability of the topic modeling algorithm to assign developers to reasonable topics.

One can interpret the remaining AT assignments along similar lines. For example, developer "daudel" is assigned to the topic corresponding to automatic code completion with probability 0.99. Referring back to the Eclipse bug data it is clear that the overwhelming majority of bug fixes for the codeassist framework were made by this developer. One can infer that this is likely to be an area of expertise of the developer.

While manual inspection provides a method for verifying reasonable AT assignments, ultimately objective evaluation is best accomplished via third-party sources with domain expertise of the software projects under consideration. To carry out such an evaluation of our methods, we submitted our initial Eclipse results to the 2007 International Workshop on Mining Software Repositories (MSR) Mining Challenge in Minneapolis, MN. A key task of this challenge was to mine Eclipse for any interesting data, and present the results as a short paper to a reviewer and jury panel consisting of several Eclipse developers, including Erich Gamma and Darin Swanson. The reviewers confirmed that our AT assignments were both indicative of the contributions of individual Eclipse developers as well as the general and specific functional components of the Eclipse code base. Our work received best-paper award for the MSR Mining "Scale" Challenge (Linstead et al. 2007), providing further independent evidence for the quality of our statistical models.

When the granularity of the AT model is changed from a single project to an Internet-scale repository, the question arises as to whether or not AT assignments become less meaningful. From our observations we have noted, that Internet-scale repositories yield substantially more general topics, but topics that are indicative of major functional categories nonetheless. In this case AT assignments are useful to categorize developer high-level contributions (database programmers versus web programmers for example), whereas in the single project case one can often map developer contributions to domain specific functional components (such as automated code completion for a development environment). In this sense the assignment for a large multi-project repository are still meaningful, but perhaps serve a different purpose.

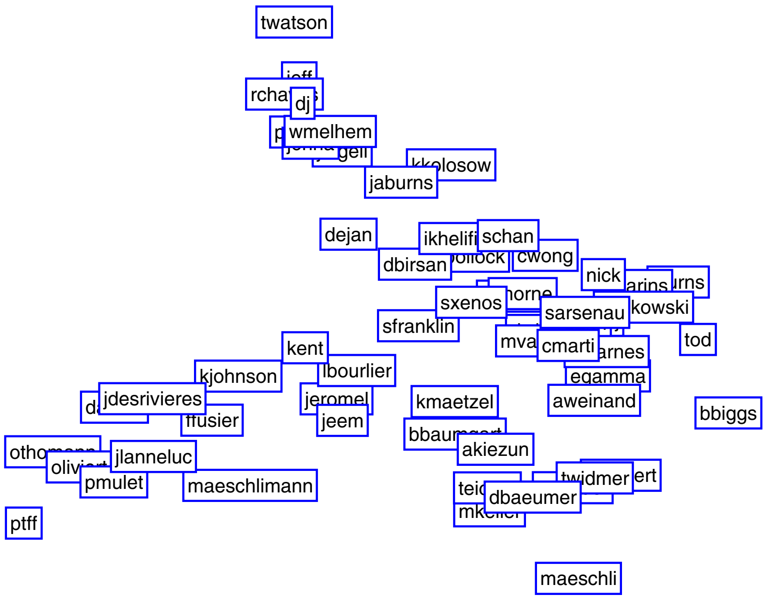

Fig. 6 All 59 Eclipse 3.0 authors clustered by KL Divergence

The breadth of an author can be measured automatically by the entropy of the corresponding distribution over topics. Likewise, the similarity between two authors can be measured by comparing their respective distributions over topics. Several metrics are possible for this purpose, but one of the most natural measures is provided by the symmetrized Kullback-Leibler (KL) divergence. Multidimensional scaling (MDS) is employed to further visualize author similarities, resulting in Fig. 6 for the Eclipse project. The boxes represent individual developers, and are arranged such that developers with similar topic distributions are nearest one another. This information is especially useful when considering how to form a development team, choosing suitable programmers to perform code updates, or bug fixes, or deciding to whom to direct technical questions. As a concrete example, consider the situation that occurs when a developer leaves a project, and shortly thereafter a software trouble report is filed for the software components he was working on. Using a visual representation such as Fig. 6, a project manager could rapidly identify another developer with similar expertise, assign the bug fix to him, and thus mitigate some of the difficulties that arise from attrition in large software projects.

By ignoring the author component of the software, one can focus topic analysis on identifying program concepts rather than developer contributions. A representative subset of 7 topics extracted via LDA modeling on our test data is given in Table 6 together with the top word probabilities.

Examining the topic column of the table it is clear that various functional domains are represented. Topic 1 clearly corresponds to database programming, topic 2 to file processing, topic 3 to networks, topic 4 to multi-threading, topic 5 to event listeners, and topic 6 to Java server pages and web programming. Though the majority of topics can be intuitively mapped to their corresponding domains, some topics are too noisy

Table 6 Selected topics with word probabilities

Topic number	Topic words with probabilities
1	Sql 0.10167
	Database 0.05753
	Update 0.03423
	Jdbc 0.02837
	Connection 0.01899
2	File 0.15861
	Path 0.15815
	Dir 0.05695
	Directory 0.04789
	Filename 0.02962
3	Server 0.10314
	Client 0.06729
	Host 0.05388
	Address 0.03657
	Port 0.03569
4	Current 0.07450
	Pool 0.03590
	Run 0.02940
	Thread 0.02889
	Start 0.02751
5	Listener 0.18784
	Event 0.11507
	Change 0.08566
	Remove 0.03827
	Fire 0.02781
6	Tag 0.17629
	Page 0.14592
	Jsp 0.05015
	Jspx 0.03705
	Body 0.03597
7	Log 0.26697
	Debug 0.13044
	Logger 0.11477
	Level 0.06333
	Logging 0.03249

to be able to associate any functional description to them. For example, one topic extracted from our data consists of Spanish words unrelated to software engineering which seem to represent the subset of source files with comments in Spanish. Other topics appear to be very project specific, and while they may indeed describe a function of code, they are not easily understood by those who are only casually familiar with the software artifacts in the codebase. In general noise appears to diminish as repository

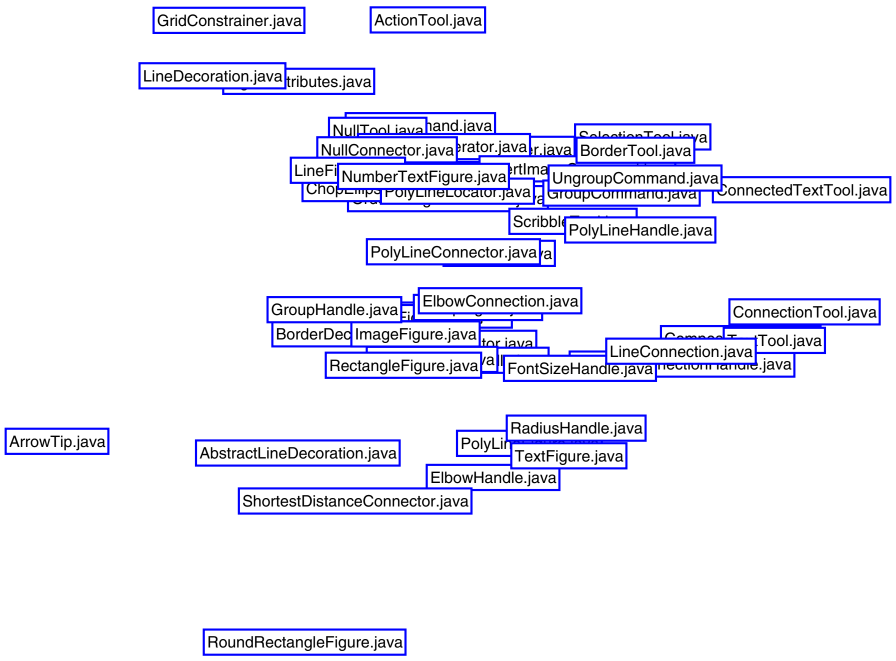

Fig. 7 Subset of JHotDraw files clustered by KL Divergence

size grows. Noise can be controlled to some degree with tuning the number of topics to be extracted, but of course can not be eliminated completely.

In addition to concept extraction, LDA provides an intuitive solution for computing the similarity between two source files by comparing their respective distributions over topics using the document-topic matrix. Figure 7 presents a MDS for a subset of 50 files from the JHotDraw project, clustered by KL divergence. The boxes represent individual files, and are arranged such that files with similar topic distributions are nearest one another. For example, by examining the figure one easily see that the files UnGroupCommand.java and GroupCommand.java are very similar, while files such as ArrowTip.java and ConnectionTool.java have little in common in terms of functional concepts. A key advantage of this approach is that topics are derived from actual source files, and thus provide a realistic and accurate basis for comparison. Using such visualizations it may be possible to predict what files are likely to be impacted by results to a given file. Moreover, such functional clustering can be leveraged for requirements mapping, especially in situations where requirement engineers need to determine which source files are responsible for providing functionality called out in design documents or requirements databases.

5.4 Document tangling and topic scattering

Two other important distributions that can be retrieved from the LDA models are the distribution of topics across documents, and the distribution of documents across topics. For each word, the topic-word distribution can be normalized to derive the corresponding word-topic distribution over topics $\phi^*_{w\bullet} = (\phi_{wt}/\sum_t \phi_{wt})$. By averaging the word-topic distributions of all the words contained in a document d, we can derive the document-topic distribution $\psi_{d\bullet} = \sum_{w \in d} \phi^*_{w\bullet}/N_d$ corresponding to the distribution over topics associated with the document d. Likewise, for a given topic t, we can derive the corresponding distribution $\psi^*_{\bullet t} = (\psi_{dt}/\sum_d \psi_{dt})$ over documents. The entropies of these distributions

$$H(t) = H(\psi^*_{\bullet t}) \quad \text{and} \quad H(d) = H(\psi_{d\bullet})$$

provide an automated and novel way to precisely formalize and measure topic scattering and document tangling (Baldi et al. 2008), respectively. Scattering and tangling are two fundamental concepts of software design, put forward in intuitive form in the aspect oriented programming (AOP) approach (Kiczales et al. 1997), which have proved to be hard to quantify and measure in previous practice. The primary goal of AOP is the separation of concerns, essentially decomposing a piece of software into modules with as little functional overlap as possible. Functionality, such as logging, which overlaps several modules, is said to be a cross-cutting concern. By computing scattering and tangling via topic distributions we provide an unsupervised framework for identifying candidate cross-cutting concerns—specifically documents and topics with high entropy. Measurements of scattering and tangling provide valuable information to software architects, for instance in terms of changing the modularity of the code (refactoring), although application of these concepts is beyond the scope of this paper.

Table 7 presents tangling results over 100 topics for the three highest and lowest entropy documents from our Eclipse 3.0 dataset. As one would expect, source files implementing general project features have high tangling, whereas source files corresponding to small or specialized features have low tangling. For example, a source file implementing features for general code editing has an entropic tangling of 6.0059 bits, while a document corresponding to a specific action has only 2.96 bits of entropy, with 6.64 bits being the maximum entropy for 100 topics. Similar observations can be made with respect to the scattering of topics, with general concepts such as text manipulation occurring in a large number of source files, and specific concepts such as unit testing occurring in a much more concentrated number of files.

Table 7 Tangling for selected Eclipse 3.0 files

File Name	Entropy (bits)
Bundle.java	6.0376
JavaEditor.java	6.0059
TextMergeViewer.java	6.0021
Locker.java	3.0572
JavaReplaceWithPreviousEditionAction.java	2.9685
IConsoleHyperlink.java	2.9129

6 Improving code search

Recent commercial work, particularly by Google with its general-purpose search engine as well as its special-purpose Google CodeSearch (and others discussed in Sect. 7), is bringing the capabilities of web search engines to code search. While developing open source code search engines can, indeed, be done these days with off-the-shelf software components and large amounts of disk space, the quality of search results is a critical issue for which there are no well-known solutions. Google general-purpose search engine appears to treat code as pure text, as it does for Web pages. Likewise, other tools such as JSearch (Sindhgatta 2006) and JIRiss (Poshyvanyk et al. 2006) employ standard IR techniques, but without the benefit of graph-based approaches. An over-arching goal in the design of Sourcerer is to allow for the combination of text and graph-based heuristics to improve the efficiency with which code is searched and retrieved. In this section we describe the search capabilities of Sourcerer, and present the results of a major experiment comparing the effectiveness of several heuristics on code retrieval.

6.1 Search by fingerprints

Fingerprint-based queries are useful when users are more interested in the general structural features of a piece of code, rather than its precise function. Though for example, researchers concentrating on bug detection may want to identify candidate race conditions in multi-threaded applications. A structural pattern that may be indicative of such a condition in Java code is the presence of wait and notify operations occurring without the protection of synchronized blocks. Using Sourcerer one can search for code that has structural fingerprints containing at least one wait or notify statement, but no synchronized keywords. An example of a class returned when executing this query against our repository is “org.oobench.threads.ThreadContentionPerformance”, which tests the contention of multiple threads for processor time. Other queries may not focus on bug detection, but rather on algorithmic efficiency; code with a high level of loop nesting may be suspect. Querying our database identifies the “checkAllTables” method of the class “com.versant.core.jdbc.sql.SqlDriver” as containing 5 nested “for” loops. While this may be necessary to implement the desired functionality of the code, the high level of nesting does impact readability, as well as the time required for a potential user to understand the implementation.

Fingerprint search is also of interest to compiler engineers or software developers attempting to assemble suites of test code containing a wide variety of structural patterns. Perhaps a programmer is interested in a piece of test code containing a switch statement and 3 loops, the nesting of which is no greater than 2. This fingerprint query can be constructed using the Sourcerer web interface, providing convenient access to real-world program examples with these features. An example of a result returned by this query is the “printNodeType” method of the class “edu.jhuapl.idmef.XMLUtils”, the source code of which is given in Appendix C. Ranking of fingerprint query results is straightforward and based on the cosine similarity measure applied to the fingerprint vectors. In contrast, ranking of keyword query results is more complex and requires taking into accounts several factors, from source-specific heuristics to graph-based ranking, and balancing their contributions, as described below.

6.2 Source-specific heuristics

Source-specific heuristics are designed to take advantage of the organizational and syntactic features that are prevalent in most well-written source files. Many of these features can be traced back to software development best-practices (url, a) and are thus widely used by the open-source community. We consider several heuristics in the development of our ranking schemes:

- *Descriptive naming conventions*: packages, classes, and methods provide the essential building blocks of Java code, and encapsulate code functionality in a hierarchical fashion. Additionally, coding standards mandate that these entities be named in a meaningful manner, providing insight as to the purpose of the code within. To this end, a first heuristic is to narrow the search to package, class, and method fully qualified names, rather than tokenizing entire code blocks. These names are further broken down using common rules for naming (such as changing case or placing underscores between terms).
- *Specificity*: we also know that there is a strict containment relation between packages, classes, and methods, in this order, and that developers use this to organize their designs. So a hit on a package name is less valuable than a hit on a class name, which, in turn, is less valuable than a hit on a method name. This can then be applied to fully qualified names, which provide unique identifiers of the form *package.class.method*. We can then take specificity into account by boosting hits where terms appear in the rightmost part of the name, as this segment of the FQN denotes the entity at the finest level of granularity.
- *Complexity*: a crude measure of complexity is the number of LOC that comprise an entity. While simple, LOC can be used as an effective ranking filter by reducing the weight of incomplete or stub implementations.
- *Identifying Test Code*: many open source projects come bundled with test code which can be useful in validating builds and configurations. Test code is seldom of use, however, when searching for reusable implementations, and can be a source of false positives. For example, a user searching for a Binary Tree is probably not interested in a JUnit test class TestBinaryTree. By reducing the weight of entities with the term test in their FQNs one hopes to reduce the number of false positives occurring at the top of the result list.

6.3 Graph-based ranking

Programs can be modeled as graphs, with code entities comprising the nodes and various relations the edges. As such, it is worth exploring other possible ranking methods that leverage the underlying graph structure, stored in the Sourcerer database, along the same lines as the PageRank algorithm of Google (Page et al. 1998). In this way we gain another vital ranking heuristic: popularity. We used Google's PageRank almost verbatim. The Code Rank of a code entity (package, class, or method) A is given by: $PR(A) = (1 - d) + d(PR(T_1)/C(T_1) + \cdots + PR(T_n)/C(T_n))$ where $T_1 \ldots T_n$ are the code entities referring to A, $C(A)$ is the number of outgoing links of A, and d is a damping factor.

Using the PageRank algorithm as a basis it is possible to devise a multitude of ranking schemes by building graphs from the many entities and relations stored in our database, or subsets thereof (Table 9). For example, one may consider the graph of only method call relationships, package dependencies, or inheritance hierarchies. Also important is the differentiation between local and global ranking schemes. Local ranking schemes are devised by building graphs for each project, and running the algorithm without taking inter-project dependencies into account. Though simple to implement, local rank disregards important relations between disparate projects. Yet it is these relationships that are perhaps the most revealing in terms of popularity, and so a more sophisticated global ranking scheme is required.

Global CodeRank is computed by inferring inter-project relations from the code repository. Java code is typically bundled with jar files containing classes on which the project is dependent, including other projects. During parsing we store all relations to jar entities. During global ranking we take the additional step of resolving the fully qualified names of jar entities to source entities in our database, allowing the source dependency graph to be augmented with these global relations.

6.4 Assessment methodology

To assess code retrieval performance, we had to manually curate a benchmark. We used carefully chosen control queries and studied the results for those queries and manually ranked them. Control queries have the following properties: (1) they result in a reasonable number of hits, large enough for diversity of results but small enough to be manually analyzable by people (typically between 50 and 200 hits); and (2) their search intent is obvious, so that it is easy to agree which hits are the most relevant.

Once control queries are selected, human experts analyze the resulting hits and decide on the N best hits for each query, where N is a number between 2 and 10. The following criteria were used to select “good hits” from the result set in the most systematic way possible: (1) content corresponding to the search intent; (2) quality of the result in terms of completeness of the solution; (3) “reputation” of the project from which the solution originates; and (4) perceived ease of reusing the component. In the final dataset, we used the consensus from three expert programmers.

We defined 25 control queries (Table 8) to be run against our indexed code repository. The queries themselves were chosen to represent the various intentions of searchers as much as possible. To this end, queries were selected that would be both indicative of a casual user, perhaps searching for an implementation of a standard data structure (e.g. a binary heap), as well as a more advanced user interested in implementations on a larger scale (e.g. a complete ftp client). The 25 control queries were extracted from a larger candidate set of over 100 queries based on a list of recent

Table 8 Experimental control queries

Database connection manager	Email validator
Depth first search	Tic tac toe
Voted perceptron	Decision tree
Binary heap	NQueens
Quick sort	Sql validator
Red black tree	Histogram plot
Fibonacci heap	PCA (principal component analysis)
Ftp client	Binary tree
Regular expression	Zip deflater
Directed acyclic graph	Pdf reader
Syntax highlight	Deadlock detection
Sigmoid function	Lock manager
Decision tree	

submissions to the Sourcerer system. This extraction was done by an independent group of three software engineers, who were also charged of determining for each query the N best hits, using the criteria above. Total number of hits was determined programmatically, independent of specific ranking schemes. In this way the final 25 queries are not biased toward any particular ranking scheme, or the Sourcerer system in general.

Finally, the different ranking schemes are compared with respect to the positions at which they place these N best hits for each control query. For each query, the N best hits are defined to be true positives, and the rest false positives. The performance of each scheme is assessed using standard IR metrics, such as Precision, Recall, receiver operating characteristic (ROC) curves, and the corresponding AUC. For this paper we selected the Google general search engine (http://www.google.com) and the Google CodeSearch engine (http://codesearch.google.com) as baselines. While the Google general search engine was not designed with code search in mind, it continues to be the means by which many developers search for reusable code, and thus merits inclusion in the comparison. Google CodeSearch allows users to search for code written in a variety of languages from an even more diverse number of repositories. The system's web interface allows one to essentially search code as text, allowing for fine-grained query tuning using a regular expression syntax. The fact that the system is source-code specific and based on standard IR techniques makes it an ideal baseline for our experiments.

A complicating factor with baseline assessment is the fact that Google has the capability to index a larger repository than the one used in these experiments. To this end, the same control queries were used to analyze performance on the Google systems, but the N best hits for each of the baselines had to be evaluated separately. This resulted in a different set of best hits than the Sourcerer system. Since only the position of the best hits in the results list was taken into account, any difference in the N best hits for the systems had negligible impact on the performance assessment. In the case where the Google search engines returned an extremely large set of results, only the first 100–200 hits were considered.

In expressing queries to the various systems, queries with multiple terms were written to require each term in the results (conjunctive queries). For the Google general search engine queries were expanded to include the terms 'java source' to attempt to focus results as much as possible on Java source code. The Google code search appliance allows the user to explicitly restrict their search to specific languages, which for these experiments was chosen to be Java.

6.5 Comparison of ranking schemes

In assessing performance of the Sourcerer system, we considered 13 distinct ranking schemes in addition to the 2 Google baselines. The 13 ranking schemes used for Sourcerer fall into 3 broad categories:

- *Code keywords only*: indexed text is derived from actual source code by tokenizing class names, method names, variable names, etc.
- *Comment keywords only*: indexed text is derives only from keywords found in comments.
- *Code and comment keywords*: indexed text consists of keywords from both code and comments.

For each of these categories we consider the effect of adding the heuristics and graph-based algorithms described in Sect. 6.

Table 9 presents the average AUC for each of the ranking schemes and baselines. It is immediately clear that the Google general search engine, with AUC of approximately 0.31, does not perform well when used to locate code on the Internet. This is of course neither a surprise nor a criticism of the application, but merely evidence of the fact that general search engines index too large a corpus using code-independent techniques. Finding relevant source code among all the other documents quickly reduces to finding the proverbial needle in a haystack. By restricting its dataset to code alone, Google's code search engine yields substantial improvement over the general engine. In fact, the AUC for this method more than doubles to almost 0.66. While far from perfect, this establishes a solid baseline for the "code as text" approach.

In terms of AUC, the Sourcerer ranking scheme with the strongest performance (0.921) uses code keywords with a combination of heuristics and global ranking. As shown in Table 10, however, restricting search to only code keywords results in an average Recall of approximately 74%. When both code and comment keywords are included, however, 100% Recall is achieved. Of the schemes considering both comment and code keywords, a combination of these keywords with heuristics and global rank yields the highest AUC at 0.841. Though 0.08 less than the best code-only approach, the improvement in Recall would indicate that this scheme is the best overall. Perhaps the ultimate decision should rest in the hands of the user, who is best equipped to determine the value of including comments (and the resulting increase in recall) in light of their particular interest. In any case, by leveraging code specific parsing, heuristics, and ranking, Sourcerer is able to deliver an AUC at least 0.21

Table 9 Mean area under curve by ranking scheme

Scheme	Mean AUC
Google	0.31
Google CodeSearch	0.658
Code keywords only	0.736
Comment keywords only	0.447
Code + heuristics	0.909
Code + heuristics + local rank	0.913
Code + heuristics + global rank	0.921
Code + boosted comments + heuristics	0.797
Code + boosted comments + heuristics + local rank	0.814
Code + boosted comments + heuristics + global rank	0.810
Code + discounted comments + heuristics	0.832
Code + discounted comments + heuristics + local rank	0.835
Code + discounted comments + heuristics + global rank	0.841
Code + heuristics - reordered by local rank	0.640
Code + heuristics - reordered by global rank	0.646

Table 10 Mean recall by ranking scheme

Keyword source	Mean recall
Code and comment keywords	1.0
Code keywords	0.740
Comment keywords	0.485

greater than Google code search. In fact, with code-specific parsing techniques alone Sourcerer outperforms Google with an AUC of .736.

From Table 9 we can see that heuristics account for the most substantial improvement in search results. Global ranking, however, provides a non-negligible improvement on top of heuristics (about 1% on average). When local rank is added to heuristics performance decreases slightly in most cases. It would appear that considering inter-project relations is an important step in the ranking process. While local rank is useful for determining the relative popularity of components within a single project, this information is not beneficial when ranking results across projects. While CodeRank provides additional refinement in addition to keywords and heuristics, results show it is ineffective as a primary ranking scheme. The last 2 rows of Table 9 present the results of experiments where query results were completely reordered based on CodeRank alone, causing the AUC to decrease to 0.64.

A final observation from the results is in regards to the usefulness of comments in code search. The figures show that considering comments alone when searching for code is of very little value. The comments-only rankings scheme yields an AUC of only 0.447, with a recall of 49%. When combined with code keywords comments prove more useful. However, if boosted above code keywords, search result quality falls below schemes where the value of hits in comment fields are discounted.

The fact that comments have relatively little impact on search result quality is validated by inspection of the source code itself. Consider a scenario where one is looking for a linked list implementation. The situation where comments are most helpful is when the source code does not employ descriptive naming conventions. For example, if searching for a "linked list" a class named "LList" will be difficult to retrieve. If, however, the developer explains that the class implements a “linked list” in the comments this can prove very useful. More often than not, however, developers use comments to indicate *how* they are doing something rather than *what* they are doing. A comment might indicate that a piece of code “stores all elements in a linked list.” Rather than providing an implementation of a linked list, the code simply creates an instance of a Java container class and uses it in support of a higher level task.

In general comments are useful in very specific cases, such as resolving abbreviations. However, those who take time to comment their code also take the time to employ descriptive naming conventions, and these naming conventions provide the greatest leveraging point for source-specific search. Nevertheless, the best balance of Precision and Recall is achieved when both code and comment keywords are combined and weighted appropriately. The addition of heuristics and global ranking yields results that surpass what is possible with a general-purpose, text-based, search engine, or with a special-purpose code search engine that treats code as pure text.

7 Related work

Our work builds on a large body of work from different parts of machine learning, IR, and software engineering.

7.1 Topic modeling of code

The notion of modeling source code with topics is not a new one. For instance, (Ugurel et al. 2002) explores code topic classification using support vector machines, but the technique is significantly different from our own. Firstly, a training set must be manually partitioned into categories based on project metadata. Topics (consisting of commonly occurring keywords) are extracted for each category, and used to form features on which to train the model. The training and testing set comprise only 100 and 30 projects, respectively. The work in Anvik et al. (2006) is similar to our own work in that it shares the goal of automated bug assignment through text categorization. Unlike our approach, however, this work uses support vector machines trained on bug reports, rather than modeling source code directly with probabilistic models, and considers a repository of only 3 projects (Eclipse, Firefox, and GCC). The approach in Marcus et al. (2004) uses latent semantic analysis to locate concepts in code. The goal is to enhance software maintainability by easily identifying related pieces of code in a software product that work together to provide a specific functionality (concept), but may be described with different keywords (synonyms, etc). In this sense the work shares some of our goals, but does not consider the problem of automatically extracting topic distributions from arbitrary amounts of source code. The work in Kuhn et al. (2007) makes progress in this area, this time using LSA to cluster related software artifacts. However, new approaches for defining topic scattering and document tangling are not considered. Andrzejewski et al. (2007) applies LDA to log traces of program execution, providing a framework statistical debugging, but does not consider more recent probabilistic techniques.

Closely related to our own work is (Kawaguchi et al. 2004), which explicitly considers the need for unsupervised categorization techniques of source code, and develops such a technique as a basis for software clustering with the aim of information sharing. Unlike our work, however, MUDABlue utilizes LSA rather than a probabilistic framework, nor does it evaluate the technique on an Internet-scale software repository consisting of thousands of projects.

Recently (Minto and Murphy 2007) has proposed a technique for mining developer expertise to assist in bug fixes. While the goal of this work is shared with the AT modeling framework presented here, the approach is substantially different, relying on author and file update metadata of the configuration management system rather than source code directly. Additionally, the approach is validated on only 3 software projects rather than the thousands considered in this paper.

Our previous work has added to this area by describing techniques for applying statistical topic models to source code. Recently, (Linstead et al. 2007) has applied AT models to source code to extract developer contributions from a subset of files of the Eclipse 3.0 codebase, but does not consider the statistical analysis or code search techniques addressed in this paper. The work in (Linstead et al. 2008) expanded the scope of topic modeling techniques to multi-project repositories, as well as presented a preliminary statistical analysis of a large software repository, but did not present the Sourcerer framework or results of detailed experiments to improve source code retrieval.

7.2 Search engines

General purpose search engines, such as Google, are "live", in the sense that they have the ability to constantly find information without the sources of that information having to report or register it explicitly, or even know that they are being analyzed. That is one of the goals of Sourcerer, and, as such, it diverges from traditional software management tools that operate over well-identified sets of source code that are collected together through some external protocol. Google's PageRank (Page et al. 1998) was also the inspiration behind our code rank algorithm, as it was for other work before ours. It is known that Google evolved from that simple heuristic to include dozens of additional heuristics pertaining to the structure of the web and of web documents. But general-purpose search engines are unaware of the specificities of source code, treating those files as any other text file on the web. Searching for source code in Google, for example, is not easy, and Google, by itself, is incapable of supporting the source-code-specific search features that we are developing.

Recent commercial work is bringing the capabilities of web search engines into code search. Koders (url, b), Krugle (url, c), Codase (url, d), csourcesearch (url, e), and Google CodeSearch (url, f) are a few prominent ones. While developing open source code search engines can, indeed, be done these days with off-the-shelf software components and large amounts of disk space, the quality of search results is a critical issue for which there are no well-known solutions. While researching these systems, we found a variety of user interfaces to search and explore open source code, but we were very often disappointed with the search results themselves. Those systems being proprietary, we were unable to make comparisons between their ranking methods and our own, yet were often left with the impression that the tools essentially provided nothing more than a way to grep source code.

There have been several code search tools developed in the academic community with the shared goal of improving developer efficiency and code reuse. JQuery (McCormick and Volder 2004) provides an Eclipse plug-in for browsing Java projects, allowing users to quickly navigate among code entities by parsing abstract syntax trees and populating a knowledge base of code entities and their relations. CodeQuest (Hajiyev et al. 2006) is similar to JQuery, but uses a richer Datalog representation in combination with a relational database to more efficiently explore software projects. Both projects are similar to Sourcerer in the sense that they leverage structural information for code search, but the intent of Sourcerer is to be able to search for general code reuse on an Internet-scale at multiple granularities, rather than within the context of a specific project. CodeBroker (Ye and Fischer 2002) seeks to maximize code reuse by providing an Eclipse plug-in that monitors a developers progress when writing new software. Using developer comments the system queries a reuse repository, attempting to present the developer with existing code modules that can be leveraged to implement the functionality described in the comments. The system is different from Sourcerer in that queries are inferred autonomously, rather than explicitly stated by the user. CodeBroker integrates the notion of conceptual similarity with the search agent, again using traditional latent semantic analysis trained on external documentation, rather than the probabilistic, code-only techniques presented in this paper. The evaluation of CodeBroker considered a reuse repository of 673 java classes, compared to the over 560,000 classes considered here in our evaluation of Sourcerer.

Two projects have recently been described in the research literature that use variants of the PageRank technique to rank code, namely Spars-J and GRIDLE. Spars-J (url, g; Inoue et al. 2003, 2005) shares some similar goals with our own work. Their Component Rank technique performs pre-processing on the graph in order to cluster classes with similar (copy-and-paste) code. The reported results of that work confirmed that it is possible to detect some notion of relevancy of a component (class) based solely on analyzing the dependency graph of those components. However, when analyzing that work we found important questions that were left unanswered. Specifically, it was unclear how the improvements shown by this graph-based heuristic compared to other, simpler, heuristics for source code search.

GRIDLE (Puppin and Silvestri 2006) is a search engine designed to find highly relevant classes from a repository. The search results (classes) are ranked using a variant of PageRank algorithm on the graph of class usage links. GRIDLE parses Javadoc documentation instead of the source code to build the class graph. Thus it ignores more fine grained entities and relations.

Stratchoma uses structural context from the code the user is working on and automatically formulates a query to retrieve code samples with similar context from a repository (Holmes and Murphy 2005). A combination of several heuristics is used to retrieve the samples. The results are ranked based on the highest number of structural relations contained in the results. The search purpose Stratchoma fulfills is only one of the many possible scenarios possible. Nevertheless, the heuristics implemented in it are good candidates to employ in searching for usage of framework classes.

JSearch (Sindhgatta 2006) and JIRiss (Poshyvanyk et al. 2006) are two other tools that employ IR techniques to code search. JSearch indexes source code using Lucene after extracting interesting syntactic entities whereas JIRiss uses Latent Semantic

Indexing (Marcus et al. 2004). Both these tools lack graph based techniques and thus are limited to their specific IR specific ranking in presenting results.

7.3 Software engineering tools

Modern software engineering tools are bringing more sophisticated search capabilities into development environments extending their traditionally limited browsing and searching capabilities. Of particular interest to this paper are the various ranking techniques and the search space these tools use.

Prospector uses a simple heuristic to rank *jungloids* (code fragments) by length (Mandelin et al. 2005). Given a pair of classes (T_{in}, T_{out}) it searches the program graph and presents to the user a ranked list of jungloids, each of which can produce class T_{out} given a class T_{in}. Its ranking heuristic is conceptually elegant but too simple to rank the relevance of search results in a large repository of programs where every search need not be for getting T_{out} given T_{in}.

XSnippet is a tool designed to allow developers to query a code repository for pertinent examples to their current programming task (Sahavechaphan and Claypool 2006). In addition to general queries, XSnippet includes facilities for satisfying context-sensitive queries by considering graph-based program structure to mine program paths that best match the users' needs, but has yet to be applied to the problem of Internet-scale code search.

Stratchoma uses structural context from the code user is working on and automatically formulates a query to retrieve code samples with similar context from a repository (Holmes and Murphy 2005). A combination of several heuristics is used to retrieve the samples. The results are ranked based on the highest number of structural relations contained in the results. The search purpose Stratchoma fulfills is only one of the many possible scenarios possible. Nevertheless, the heuristics implemented in it are good candidates to employ in searching for usage of framework classes. Additionally, the work in Holmes et al. (2006) presents similar techniques for finding instances of API usage by locating structural clones in a software repository, and thus shares our goal of leveraging structural information from software to improve and augment retrieval methods for code exploration.

JSearch (Sindhgatta 2006) and JIRiss (Poshyvanyk et al. 2006) are two other tools that employ IR techniques to code search. JSearch indexes source code using Lucene after extracting interesting syntactic entities whereas JIRiss uses Latent Semantic Indexing (Marcus et al. 2004). Both these tools lack graph based techniques and thus are limited to their specific IR specific ranking in presenting results.

7.4 Fingerprints and structure based search

A common use of fingerprinting in the past has been for plagiarism detection techniques, such as the work in (Schleimer et al. 2003), which uses approximate string matching to detect copied documents on the web, but does not consider software data. The work in (Liu et al. 2006) is more similar to our own work in that it considers the problem of source code similarity, though the work is again done in the context of plagiarism of open source rather than structure-based search, and lacks the focus on Internet-scale software repositories.

We found very few examples of using code fingerprints specially for the purposes of software search but that may be because the software community uses alternative terms for denoting the same concept. Hill and Rideout use method fingerprints for automatic method completion (Hill and Rideout 2004). A similar idea is employed in the Software Bookshelf project (Finnigan et al. 1997), for representing software entities in terms of metrics. Our fingerprints also have similar goals to work focusing on finding source code patterns (Paul 1992; Paul and Prakash 1994). Our fingerprints are less expressive, in the sense that they don't support full pattern matching, but they are very efficient for search, as they involve only simple numerical computations.

8 Conclusion

The ever-increasing quantity of publicly available source code presents new challenges and opportunities for mining and searching software repositories. Here we have presented Sourcerer, an extensive infrastructure for downloading, parsing, storing, and analyzing Internet-scale software corpora, as well as a search engine supporting keyword-based and structure-based querying. After populating Sourcerer with over 4,600 open source projects, simple statistical analysis verified the presence of robust power-law distributions among Java entities and relations. We then developed and applied probabilistic topic and AT models for software as a means for automatically extracting program function, similarity, and developer contributions, as well as quantifying entropic scattering and tangling, with applications ranging from software staffing to refactoring. The methods developed are applicable at multiple scales, from single projects to Internet-scale repositories. Finally, by combining term-based IR techniques with graphical information derived from program structure, we were able to significantly improve software search and retrieval performance, increasing the efficiency with which software developers can identify, explore, and reuse existing open source implementations.

Acknowledgements Work in part supported by National Science Foundation MRI grant EIA-0321390 and a Microsoft Faculty Research Award to PB, as well as National Science Foundation grant CCF-0725370 to CL and PB. We thank Ian Harris and Padhraic Smyth (UCI) for useful discussions.

Appendix A: Entities and relations indexed by Sourcerer

A.1 Entities

Entity
Package
Class
Method
Field
Constructor
Static initializer

A.2 Relations

Relation	Description
Inside	Lexical encapsulation of one entity inside another
Use	One relation uses another to achieve functionality
Extends	One class subclasses another
Implements	A class implements a given interface
Calls	One method calls another
Throws	One entity throws another as an exception
Returns	A method returns an entity
Overrides	A class overrides a method
Overload	One entity overloads a method
Instantiates	One entity instantiates another via the ‘new’ keyword
Assigned	A method call assigns a value to a field
Holds	A field holds an entity of a given type
Receives	A method receives an entity as an input parameter
Accesses	An entity reads a field

Appendix B: Fingerprints

B.1 Structural fingerprints

Attribute	Description
Synchro_count	Number of synchronization statements
Wait_count	Number of wait’s
Notify_count	Number of notify’s
Loop_count	Number of loops
If_count	Number of if statements
Switch_count	Number of switch statements
Path_count	Number of branches
Dynamic_count	Number of dynamic memory allocations (i.e. new)
Avg_loop_length	Average number of statements in a loop
Max_loop_nest	Maximum number of nested loops

B.2 Type fingerprints

Attribute	Description
Modifiers	Modifiers applied to entity
Field_self_type	Whether or not class holds reference to its own type
Class_count	Number of classes
Interface_count	Number of interfaces
Decl_method_count	Number of declared methods
Method_count	Number of methods (declared and inherited)
Decl_constructor_count	Number of declared constructors
Constructor_count	Total number of constructors
Decl_field_count	Number of declared fields
Field_count	Number of fields (declared and inherited)
Static_init_count	Number of static initializers
Param_count	Number of parameters

Attribute	Description
Decl_overload_count	Number of declared methods with overload
Overload_count	Number of methods that overload methods in superclasses
Override_count	Number of declared methods that override methods in superclasses
Implements_count	Number of interfaces implemended
Parents_count	Number of ancestors in inheritance hierarchy

B.3 Micro pattern fingerprints

Attribute	Description
Designator	Interface with no member
Taxonomy	Interface extend others with no member
Joiner	Empty interface extends two or more interfaces
Pool	Class with only static and final fields, no method
Function_pointer	Only one public instance method, no field
Function_object	Only one public instance method, at least one instance field
Cobol_like	Class with single static method, no instance member
State_less	No field, other static final
Common_state	Class, all fields are static
Immutable	Class with serveral instance field, which is assign exactly one during the instantiation
Restricted_creation	Class with no public constructor, at least one static field of the same type of the class
Sampler	Class with one ore more public constructor at least one static field of the same type of the class
Box	A class which has exactly one muttable instance field
Compound_box	Class with exactly one, not primitive instance field
Canopy	Class with exactly one instance field, that is assigned exactly one during the instance construction
Record	A class in which all fields are public, and no declared method
Data_manager	A class all methods are either getter setter
Sink	A class whose methods do not propagate calls to any other class
Outline	A class where at least two methods invoke an abstract method on ‘this’
Trait	An abstract class which has no state (field)
State_machine	An interface whose methods accept no parameters
Pure_type	A class with only abstract method, no static member, no field
Augmented_type	Only abstract methods, 3 or more static final fields of the same type
Pseudo_class	A class which can be rewritten as an interface (it has no concrete method, it has only static fields)
Implementor	A concrete class where all methods override inherited abstract methods
Overrider	A class in which all methods override inherited non abstract methods
Extender	A class which extends the inherited protocol, without overriding any method

Appendix C: Example of fingerprint query result

A switch statement and 3 loops, the nesting of which is no greater than 2.

```
/**prints the type of the input node
  * @param node node to print type of
  * @param ident amount to indent*/
  public static void printNodeType(Node node, int ident)
  {
  System.out.print("Node: " + node.getNodeName() + " ");
   switch (node.getNodeType()) {
     case Node.DOCUMENT_NODE:
       System.out.println("Document Node");
       break;
     case Node.ELEMENT_NODE:
       System.out.println("Element Node");
       for (int j = 0; j<2 *ident; j++)
         System.out.print(" ");
       System.out.println("It has the following
        Children");
       NodeList children = node.getChildNodes();
       if (children != null) {
         for (int i=0; i<children.getLength(); i++) {
           for (int j = 0; j<ident; j++)
             System.out.print(" ");
         System.out.print ("Child " + ident + "."+
          i + " = ");
         printNodeType(children.item(i), ident + 1);
         }
       System.out.println();
       }
       break;
     case Node.TEXT_NODE:
       System.out.println("->"+node.getNodeValue()
         .trim()+"<-");
       break;
     case Node.CDATA_SECTION_NODE:
       System.out.println("CData Node");
       break;
     case Node.PROCESSING_INSTRUCTION_NODE:
       System.out.println("Proposing Instruction Node");
       break;
     case Node.ENTITY_REFERENCE_NODE:
       System.out.println("Entity Node");
       break;
     case Node.DOCUMENT_TYPE_NODE:
       System.out.println("Document Node");
       break;
     default:
     }
}
```

References

Sun Java Coding Standards. http://java.sun.com/docs/codeconv/
Koders web site. http://www.koders.com
Krugle web site. http://www.krugle.com
Codase web site. http://www.Codase.com
Csourcesearch web site. http://csourcesearch.net/
Google CodeSearch web site. http://www.google.com/codesearch
SparsJ Search System. http://demo.spars.info/
Andrzejewski D, Mulhern A, Liblit B, Zhu X (2007) Statistical debugging using latent topic models. In: Matwin S, Mladenic D (eds) 18th European conference on machine learning. Warsaw, Poland (to appear)
Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: ICSE '06: proceeding of the 28th international conference on Software engineering. ACM, New York, pp 361–370
Baeza-Yates RA, Ribeiro-Neto BA (1999) Modern information retrieval. ACM Press/Addison-Wesley
Baldi P, Frasconi P, Smyth P (2003) Modeling the internet and the web: probabilistic methods and algorithms. Wiley
Baldi P, Lopes C, Linstead E, Bajracharya S (2008) A theory of aspects as latent topics. In: OOPSLA '08: proceedings of the 23rd annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications, Nashville, TN. ACM, New York, NY (to appear)
Blei D, Lafferty J (2006) Correlated topic models. In: Weiss Y, Schölkopf B, Platt J (eds) Advances in neural information processing systems, vol 18. MIT Press, Cambridge, MA, pp 147–154
Blei D, Ng A, Jordan M (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022
Brill E (1994) Some advances in transformation-based part of speech tagging. In: National conference on artificial intelligence. pp 722–727
Buntine W (2005) Open source search: a data mining platform. SIGIR Forum 39(1):4–10
Chen J, Swamidass SJ, Dou Y, Bruand J, Baldi P (2005) ChemDB: a public database of small molecules and related chemoinformatics resources. Bioinformatics 21:4133–4139
Concas G, Marchesi M, Pinna S, Serra N (2007) Power-laws in a large object-oriented software system. IEEE Trans Softw Eng 33(10):687–708
Cox A, Clarke C, Sim S (1999) A model independent source code repository. In: CASCON '99: proceedings of the 1999 conference of the centre for advanced studies on collaborative research. IBM Press, p 1
Deerwester S, Dumais S, Landauer T, Furnas G, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407
Finnigan P, Holt R, Kalas I, Kerr S, Kontogiannis K, Mueller H, Mylopoulos J, Perelgut S, Stanley M, Wong K (1997) The software bookshelf. IBM Sys J 36(4):564–593
Frean GBM, Noble J, Rickerby M, Smith H, Visser M, Melton H, Tempero E (2006) Understanding the shape of java software. In: OOPSLA '06. ACM Press, New York, pp 397–503
Gil JY, Maman I (2005) Micro patterns in java code. In: OOPSLA '05: proceedings of the 20th annual ACM SIGPLAN conference on object oriented programming systems languages and applications. ACM Press, New York, pp 97–116
Griffiths TL, Steyvers M (2004) Finding scientific topics. Proc Natl Acad Sci U S A 101(Suppl 1):5228–5235
Hajiyev E, Verbaere M, de Moor O (2006) Codequest: scalable source code queries with datalog. In: Thomas D (ed) ECOOP'06: proceedings of the 20th European conference on object-oriented programming, vol 4067 of lecture notes in computer science. Springer, Berlin, pp 2–27
Hill R, Rideout J (2004) Automatic method completion. In: ASE. IEEE Computer Society, pp 228–235
Holmes R, Murphy GC (2005) Using structural context to recommend source code examples. In: ICSE '05: proceedings of the 27th international conference on software engineering. ACM Press, New York, pp 117–125
Holmes R, Walker RJ, Murphy GC (2006) Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans Softw Eng 32(12):952–970
Inoue K, Yokomori R, Fujiwara H, Yamamoto T, Matsushita M, Kusumoto S (2003) Component rank: relative significance rank for software component search. In: ICSE '03: proceedings of the 25th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 14–24

Inoue K, Yokomori R, Yamamoto T, Kusumoto S (2005) Ranking significance of software components based on use relations. IEEE Trans Softw Eng 31(3):213–225

Kawaguchi S, Garg PK, Matsushita M, Inoue K (2004) Mudablue: an automatic categorization system for open source repositories. In: APSEC '04: proceedings of the 11th Asia-Pacific software engineering conference (APSEC'04). IEEE Computer Society, Washington, DC, pp 184–193

Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J, Irwin J (1997) Aspect-oriented programming. In: Akşit M, Matsuoka S (eds) Proceedings European conference on object-oriented programming, vol 1241. Springer, Berlin, pp 220–242

Knuth DE (1971) An empirical study of fortran programs. Softw Pract Exp 1(2):105–133

Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: identifying topics in source code. Info Softw Technol 49(3):230–243

Linstead E, Rigor P, Bajracharya S, Lopes C, Baldi P (2007) Mining eclipse developer contributions via author-topic models. MSR 2007: proceedings of the fourth international workshop on mining software repositories. pp 30–33

Linstead E, Rigor P, Bajracharya S, Lopes C, Baldi P (2008) Mining internet-scale software repositories. In: Platt JC, Koller D, Singer Y, Roweis S (eds) Advances in neural information processing systems, vol 20. MIT Press, Cambridge, MA, pp 929–936

Liu C, Chen C, Han J, Yu PS (2006) Gplag: detection of software plagiarism by program dependence graph analysis. In: KDD '06: proceedings of the 12th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, New York, pp 872–881

Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: helping to navigate the api jungle. In: PLDI '05: proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation. ACM Press, New York, pp 48–61

Marcus A, Sergeyev A, Rajlich V, Maletic J (2004) An information retrieval approach to concept location in source code. In: Proceedings of the 11th working conference on reverse engineering (WCRE 2004). The Netherlands, pp 214–223

McCormick E, Volder KD (2004) Jquery: finding your way through tangled code. In: OOPSLA '04: companion to the 19th annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications. ACM. New York, pp 9–10

Minto S, Murphy GC (2007) Recommending emergent teams. In: MSR '07: proceedings of the fourth international workshop on mining software repositories. IEEE Computer Society. Washington, DC, p 5

Mitzenmacher, M. (2003) A brief history of generative models for power law and lognormal distributions. Internet Math 1(2)

Navarro G (2001) A guided tour to approximate string matching. ACM Comput Surv 33(1):31–88

Newman D, Block S (2006) Probabilistic topic decomposition of an eighteenth-century american newspaper. J Am Soc Inf Sci Technol 57(6):753–767

Newman D, Chemudugunta C, Smyth P, Steyvers M (2006) Analyzing entities and topics in news articles using statistical topic models. In: ISI. pp 93–104

Oman P, Hagemeister J (1992) Metrics for assessing a software system's maintainability. In: Proceedings of the international conference on software maintenance 1992. IEEE Computer Society Press, pp 337–344

Page L, Brin S, Motwani R, Winograd T (1998) The pagerank citation ranking: bringing order to the web. Stanford Digital Library working paper SIDL-WP-1999-0120 of 11/11/1999 (see: http://dbpubs.stanford.edu/pub/1999-66)

Paul S (1992) Scruple: a reengineer's tool for source code search. In: CASCON '92: proceedings of the 1992 conference of the centre for advanced studies on collaborative research. IBM Press, pp 329–346

Paul S, Prakash A (1994) A framework for source code search using program patterns. IEEE Trans Softw Eng 20(6):463–475

Poshyvanyk D, Marcus A, Dong Y (2006) Jiriss—an eclipse plug-in for source code exploration. ICPC 0:252–255

Puppin D, Silvestri F (2006) The social network of java classes. In: Haddad H (ed) SAC. ACM, New York, pp 1409–1413

Rosen-Zvi M, Griffiths T, Steyvers M, Smyth P (2004) The author-topic model for authors and documents. In: UAI '04: proceedings of the 20th conference on uncertainty in artificial intelligence. AUAI Press, Arlington, pp 487–494

Sahavechaphan N, Claypool K (2006) Xsnippet: mining for sample code. SIGPLAN Not 41(10):413–430

Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: SIGMOD '03: proceedings of the 2003 ACM SIGMOD international conference on management of data. ACM, New York, pp 76–85

Schröter A, Zimmermann T, Premraj R, Zeller A (2006) If your bug database could talk …. In: Proceedings of the 5th international symposium on empirical software engineering, vol II: short papers and posters. Rio de Janeiro, pp 18–20

Sindhgatta R (2006) Using an information retrieval system to retrieve source code samples. In: Osterweil LJ, Rombach HD, Soffa ML, (eds) ICSE. ACM, pp 905–908

Steyvers M, Smyth P, Rosen-Zvi M, Griffiths T (2004) Probabilistic author-topic models for information discovery. In: KDD '04: proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM Press, New York, pp 306–315

Swamidass S, Baldi P (2007) Bounds and algorithms for exact searches of chemical fingerprints in linear and sub-linear time. J Chem Inf Model 47(2):302–317

Teh YW, Jordan MI, Beal MJ, Blei DM (2006) Hierarchical Dirichlet processes. J Am Statistical Assoc 101(476):1566–1581

Ugurel S, Krovetz R, Giles CL (2002) What's the code? automatic classification of source code archives. In: KDD '02: proceedings of the eighth ACM SIGKDD international conference on knowledge discovery and data mining. ACM Press, New York, pp 632–638

Ukkonen E (1992) Approximate string-matching with q-grams and maximal matches. Theor Comput Sci 92(1):191–211

Welker KD, Oman PW (1995) Software maintainability metrics models in practice. Crosstalk, J Def Softw Eng 8:19–23

Wheeldon R, Counsell S (2003) Power law distributions in class relationships. In: International workshop on source code analysis and manipulation, pp 45–54

Ye Y, Fischer G (2002) Supporting reuse by delivering task-relevant and personalized information. In: ICSE '02: proceedings of the 24th international conference on software engineering. ACM, New York, pp 513–523

Zipf GK (1932) Selective studies and the principle of relative frequency in language. Harvard University Press