Preprocessing of Object-Oriented Source Code for Code Retrieval

Jörg Rech
67663 Kaiserslautern, Germany
Joerg.Rech@gmail.com

Abstract

Object oriented source code occurs in diverse programming languages with documentation using miscellaneous standards, comments in individual styles, or associated test cases that are hard to exploit through information retrieval or knowledge discovery techniques. Typically, the information about object-oriented source code for a software system is distributed across several different sources, which makes processing complex. In this paper we describe the morphology of object-oriented source code and how we preprocess it to improve the retrieval of source code for further reuse. Results from two studies showed that the preprocessed index increases the precision of the search by at least 13% for queries encompassing a whole class and 33% for queries consisting of the class name.

1 Introduction

Traditionally, databases storing software artifacts were used to store manually classified components from in-house software systems. But as the manual classification of source code is time-consuming and costly, automated techniques for software retrieval and reuse are required to efficiently and effectively process large amounts of source code.

Today, we use *code warehouses* to store many software systems in different versions for further processing, which are based on the data warehouse framework described by Inmon [Inmon 1996]. Operational software configuration management systems (SCMS), similar to data marts, are tapped to integrate software artifacts into the code warehouse. Well-known representatives are CVS, Subversion, or SourceSafe. Typically, these software repositories consist of a vast quantity of different files with interconnected source code and additional information associated with the source code (e.g., documentation). Extraction, transformation, and loading processes (ETL) are used to extract the source code from different software repositories into the code warehouse. Due to the astonishing success and propagation of open source software (OSS) and large OSS repositories such as Sourceforge (cf. http://www.sourceforge.net), many SCMS are freely available in different shapes and sizes. By tapping these operational code marts, large amounts of reusable software artifacts in diverse languages, formats, and with additional information are available for further reuse, analysis, and exploration.

The infrastructure of the code warehouse as illustrated in Fig. 1 depicts the flow of complex data such as source code from a CVS into our code warehouse. As shown in Fig. 1, we extract source code via wrappers from a CVS, parse the code to extract structural information (e.g., classes, methods, attributes) and associated documents (e.g., license information), to finally integrate and store them in the code warehouse.

In this paper we describe the data our system is using and processing (i.e., object-oriented source code) and present the techniques to preprocess this data. The goal of our approach is the accumulation of large amounts of source code in a processable format to build a reuse engine and a knowledge discovery (KD) engine on top of it. While the KD engine will be used to identify potential libraries or defects in and between software projects the reuse engine is used to learn about what part of a software system is more likely to be reused and how we have to characterize code in order to improve its retrieval. This might be used in the (far) future for the automated assembly of simple services to build more complex services either via rule-based, learning, evolutionary, or agent-based systems.

After a section about relevant background concerning software reuse, a section is used to describe the morphology of the used data. Thereafter, we present the techniques of how we preprocess the data, and how we use it in software engineering for source code retrieval. Finally, we describe the evaluation of our approach in two studies using queries encompassing a whole class and queries consisting of terms from the class name.

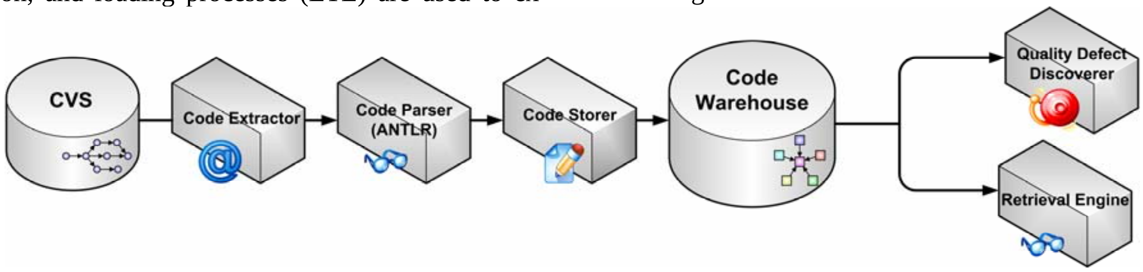


Fig. 1 Data flow in the Code Warehouse

2 Background

The reuse of existing knowledge and experience is a fundamental practice in many sciences. Engineers often use existing components and apply established processes to construct complex systems. Without the reuse of well-proven components, methods, or tools engineers have to rebuild and relearn these components, methods, or tools again and again.

Today, *reuse-oriented software engineering* covers the process of development and evolution of software systems by reusing existing software components. The goal is to develop complex software systems in shorter periods of time or with a higher quality by reusing proven, verified, and tested components from internal or external sources. By the systematic reuse of these components and feedback about their application, their internal quality (e.g., reliability) is continuously improved. But reuse of components is only appropriate if the cost of retrieving and adapting the component is either less costly or results in higher quality.

2.1 Traditional Software Reuse

Since the eighties the systematic reuse and management of experiences, knowledge, products, and processes was refined and named *Experience Factory* (EF) [Basili et al. 1994]. This field, also known as *Experience Management* [Jedlitschka et al. 2002] or *Learning Software Organization* (LSO) [Ruhe & Bomarius 1999], researches methods and techniques for the management, elicitation, and adaptation of reusable artifacts from SE projects. The *Component Factory* (CF) as a specialization of the EF is concerned with the capturing, managing, and reuse of software artifacts [Basili et al. 1992] and builds the framework in which further knowledge discovery and information retrieval techniques are embedded.

In the beginning only the reuse of source code was the focus of reuse-oriented software engineering. Today, the comprehensive reuse of all software artifacts and experiences from the software development process increases in popularity [Basili & Rombach 1991]. Besides source code artifacts such as requirements, design document, test cases, process models, quality models, and best practices (e.g., Design Patterns) are used to support the development and evolution of software systems. These artifacts are collected during development or reengineering processes and typically stored in specific artifact-specific repositories.

2.2 Agile Software Development and Reuse

Agile software development methods impose as little overhead as possible in order to develop software as fast as possible and with continuous feedback from the customers. These methods (and especially extreme programming (XP)) are based upon 12 principles [Beck 1999].

Traditional software reuse initiatives and approaches that were developed for process-driven software development are inadequate for highly dynamic and agile processes where the software cannot be developed for reuse and reuse cannot be planned in advance. Teams and organizations developing with agile methods need automated tools and techniques that support their work without consuming much time. Therefore, *agile software reuse* is a fairly new area where minimally invasive techniques are researched to support software engineers [Cinneide et al. 2004].

2.3 Repositories for Software Reuse

In the nineties research projects about software repositories were concerned with the development of software repositories with specific data models and interfaces for single organizations. For example, the Experience Factory of the SFB 501 [Feldmann 1999], the “Experience Management System” (EMS) of the University of Maryland [Basili et al. 2002], and the “Repository in a Box” of the University of Tennessee under direction by the National HPCC Software Exchange (NHSE) [Browne et al. 1998]. In-house products from software companies were, among others, ReDiscovery from IBM [IBM 1994], the Workstation Software Factory (WSF) Repository from Bellcore [Shklar et al. 1994], the Semantic Hypertext Object Repository (SHORE) from sd&m [Zündorf et al. 2001], or the SEEE from Fraunhofer IESE [Althoff et al. 1999].

All of these approaches are based on manual classification of software artifacts and use either classification values or textual descriptions (e.g., manpages) for retrieval. A new approach is the Software Warehouse concept [Dai et al. 2004] that builds a framework for retrieval and mining activities but has currently only presented the software cube (i.e., data cube for software) as an innovation and needs manual classification to describe software artifacts (e.g., domain features).

Until now relatively few approaches have exploited the information hidden in source code on a large scale. The success of open source software and the resulting availability of massive amounts of source code enabled this development. Several systems related to our research are described in the following list:

A commercial project similar to the code warehouse has recently been introduced. The free source-code search engine *Koders* and its commercial subproject KodeShare (http://www.koders.com/) offers the opportunity to search in source code of open source projects similar to the Google approach (i.e., a web search engine). Koders has indexed the sourcecode from tens of thousands of projects from free repositories in 15 different programming languages encompassing about 125 million lines of code. As no information was published about the inner working of Koders we can only assess its offered functionality. The results presented after a free-text search indicate that Koders not only indexes whole files containing classes but also parses their contents to identify methods and fields (a.k.a. attributes) as well as license information.

The research project *AMOS* (http://www.clip.dia.fi.upm.es/~amos/AMOS/) follows a similar approach but uses a taxonomy (i.e., an ontology) for search terms defined by domain experts in order to enable the search over source code from predefined projects [Carro 2002]. The functionality of packages (e.g., source file, part of source file, or collection of source files) is described manually based on a predefined ontology (i.e., a dictionary of related terms). Interestingly, the search engine is also capable of integrating several single packages in larger packages encompassing the whole search query. As it only searches over the signature of the code one can specify exact queries if the user knows what to look for. A more exploratory search based on code description, comments, or identifiers is not possible.

Another commercial project is DevX *Sourcebank* (http://archive.devx.com/sourcebank/) that represents a directory of links to source code, scripts, and papers from several sources around the Internet similar to the yahoo approach (i.e., a web directory). It enables searching and browsing over these resources by a simple query interface and supports the restriction to a programming language. Results are viewable via a link to the original place (if it still exists) and typically include information such as title, author, description (e.g., the javadoc), language, URL, and the date it was added to the repository.

Similar to the Sourcebank is DevDaily's repository search engine called the *Java Source Code Warehouse* (http://www.devdaily.com/java/jwarehouse/). Currently, it has indexed source code from about 20 free java repositories and its search is based on the Google™ search engine. Besides java it supports six programming languages. The results presented after the Google-based search indicate it only indexes whole files containing classes as filed in their subdirectories.

Finally, the project *JDocs* (http://www.jdocs.com/) is based on the open-source project *Ashkelon* (http://ashkelon.sourceforge.net/) and represents a type of repository that provides a knowledge base defined around the core Java API's (Application Programming Interfaces). It does not include source code but gives access to a collection of 132 API's from Java frameworks and libraries. While this approach is great for programmers wanting to search in API's, two problems remain. First, one cannot directly access the source code described by an API element, and second, even if the user finds a relevant class he has no information how to use it in a real context.

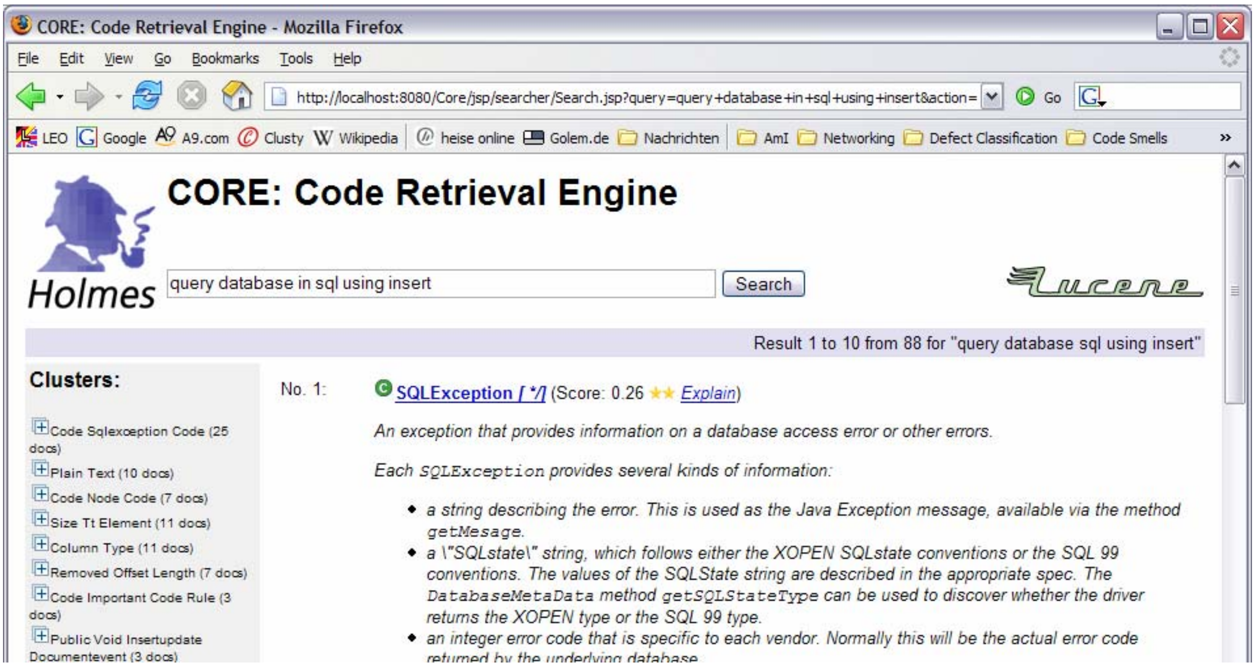
3 Code Retrieval for Software Reuse

In software reuse previously constructed source code is reused to save the time of redevelopment. Our goal is to support software developers, designers, analysts, or testers in deciding what to reuse, to increase their options what they could reuse, and augment their decisions in reusing (parts of) software systems.

To support the reuse of complex information as source code we build a source code search component based on technology similar to Google™ but specialized for source code. The basic application is to find answers to questions such as "How does the quicksort algorithm looks like in C#?" or "How should the JDOM API be used?" By the integration of information about the inherited functionality, used methods, or used licenses we can also support the user in the following questions:

- *May I reuse the source code found?* Based on the license information attached to the source code and a definition of the license the user is informed if he or she might or might not directly copy the code.
- *What do I need to make the code work in my context?* By seeing and browsing the associated relations (e.g., imports or method calls) the user can quickly oversee what libraries or functionalities has to be included in order to make the code work. Additionally, the documentation of the source code (e.g., javadoc) often describes the functionality and similar code fragments that might be used to decide over the adaptation effort.
- *What is the quality of the code?* Based on the metrics data attached to the source code the user might deduce qualities about the code. In the future we will integrate the specification of quality models in order to, for example, calculate the maintainability of the code.
- *How do I test the functionality?* As test cases are associated with the respective source code if they exist at all the user can easily get source code to test the reused functionality after appropriate adaptations.

As depicted in Fig. 2 our system presents search results similar to Google and includes information such as the signature, license, type of language, project, version, or documentation. The score is calculated by the underlying search engine and is essentially based upon the term frequency in the document and its length. Currently we also analyze techniques for the clustering of search results (e.g., see http://www.clusty.com) in order to give a better overview and discover similar elements.


CORE: Code Retrieval Engine - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
LEO Google A9.com Clusty Wikipedia heise online Golem.de Nachrichten AmI Networking Defect Classification Code Smells
Holmes
Search
Lucene
Clusters:

http://localhost:8080/Core/jsp/searcher/Search.jsp?query=query+database+in+sql+using+insert&action=

CORE: Code Retrieval Engine

query database in sql using insert

Result 1 to 10 from 88 for "query database sql using insert"

- Code Sqlexception Code (25 docs)
- Plain Text (10 docs)
- Code Node Code (7 docs)
- Size Tt Element (11 docs)
- Column Type (11 docs)
- Removed Offset Length (7 docs)
- Code Important Code Rule (3 docs)
- Public Void Insertupdate Documentevent (3 docs)

No. 1: SQLException [*/] (Score: 0.26 ★★ Explain)

An exception that provides information on a database access error or other errors.

Each `SQLException` provides several kinds of information:

- *a string describing the error. This is used as the Java Exception message, available via the method `getMesage`.*
- *a \"SQLstate\" string, which follows either the XOPEN SQLstate conventions or the SQL 99 conventions. The values of the SQLState string are described in the appropriate spec. The `DatabaseMetaData` method `getSQLStateType` can be used to discover whether the driver returns the XOPEN type or the SQL 99 type.*
- *an integer error code that is specific to each vendor. Normally this will be the actual error code returned by the underlying database*

Fig. 2 Screenshots from the Code Warehouse

4 Preprocessing of Object-Oriented Source Code

Until today, the functionality or semantics of a code element can not be extracted from the syntax of arbitrary source code. A computer does not understand what an algorithm like “quicksort” does. Nevertheless, valuable information can be extracted from associated and internal sources. While source code is typically represented in a single file for every class of the system, inside these files additional blocks of information can be identified such as methods, attributes, comments, or documentation in JavaDoc.

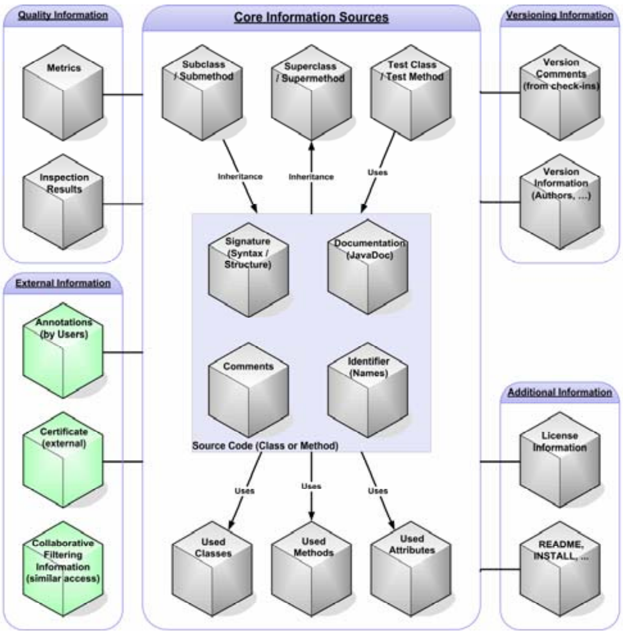


Fig. 3 Structure of object-oriented source code

In Fig. 3, every box symbolizes an individual block of information, which can be seen as a self-containing (or relatively independent) set of *features* (i.e., a bag of words or data).

In the following description of information associated to source code we refer to Java source code from open source projects and libraries found on the open source repositories Sourceforge (http://www.sourceforge.net/) and Freshmeat (http://www.freshmeat.net).

4.1 Object-oriented Source Code

Object-oriented software systems consist of objects that package data and functionality together into units that represent objects of the real world (e.g., a sensor or string). These objects can perform work, report on and change their internal states, and communicate with other objects in the system without revealing how their features are implemented. This ensures that objects cannot change the internal state of other objects in unexpected ways and that only the objects own internal methods are allowed to access their states. Similarly, more abstract building blocks such as packages and projects hide additional information that should not be visible to other resources in order to minimize maintenance effort if a unit has to be changed (e.g., if hardware such as a sensor is exchanged).

In software reuse several of these units might be of interest to a potential user. The user might want to reuse a whole database (e.g., mySQL or PostgreSQL) or need a solution to implement a fast sorting algorithm (e.g., quicksort). Fig. 4 shows several blocks that might be of interest to the user that are returned on a query. *Classes* describe these objects and group their functionality (i.e., their *methods)* and *attributes* into a single file. While every class of the system is grouped into a *package* that, in general, represents a subsystem, these subsystems are not defined in the source code. Furthermore, software *projects* are typically developed and improved over longer periods of time and stored in *builds* after specific tasks are completed (e.g., a release is finalized).

While most information blocks can be automatically extracted from the existing source code, several blocks might also be attached manually (e.g., subsystem information or annotations by (re-) users). Associated information can be integrated into retrieval and mining processes in order to improve their precision and recall.

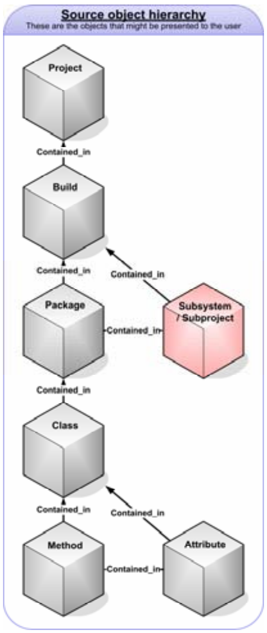


Fig. 4 Source code related structure of software projects

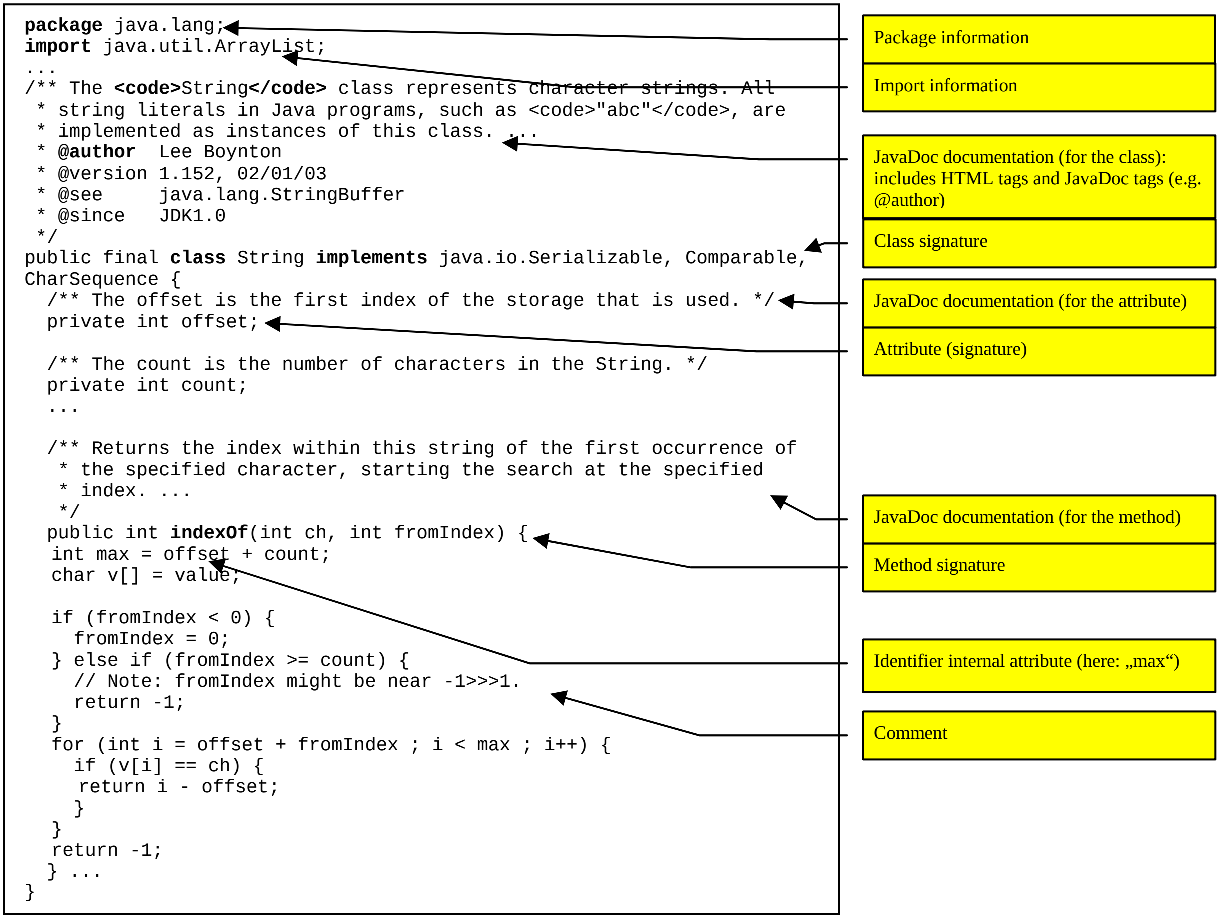
While it is possible to write multiple classes in one file or use internal classes (i.e., a sub-class in a class) source code is typically encoded in classes that are written in single file. These files contain the description of their membership in a specific *package* and define which additional classes (beside sub-classes or package-neighbors) are needed and have to be *imported* in order to compile this class. Other information such as *method calls* or *inheritance relations* that describe external links and requirements of the code can be exploited with several techniques (e.g., the Pagerank algorithm [Brin & Page 1998]).
Beside these external links source code contains several internal information blocks. As depicted in Fig. 3 these blocks build the core information sources to describe classes in the following order:

- **Signature**: The signature of a method, class or attribute defines it name (i.e., an identifier), visibility (e.g., public), and inheritance relationships (i.e., used superclasses and interfaces). All these blocks represent valuable information sources but require additional processing to be useful. While relations and modifier (e.g., the visibility) are unambiguously defined for a software system, the name of a class is basically free text and encoded in *camelcases* (e.g., “`StringBuffer`”) that has to been parsed and normalized (e.g., into “string” and “buffer”).
- **Documentation**: The description of the artifact in a specific markup language (here JavaDoc with text in HTML and special tags (e.g., the `@author` tag is used to associate multiple authors that created or modified (parts) of the code)). Other documentation markup languages exist for nearly every programming language (e.g., ePyDoc for Python or doxygen for multiple languages). As the documentation is based on a specific markup language the semantic of the text within is semi-structured and specific parsers for HTML or

JavaDoc-Tag help to extract additional information (e.g., links to related documents).

- **Comments**: Single or multiple line comments represent in general either notes of the developers to describe the specific code semantics or declare dead source code (that should not be executed but might be used in the future). Typically, there is no structure in comments and they can be seen as free text fields.
- **Identifier**: Names and types of variables, constants, or methods defined in a software system and used in a class represent additional information to characterize the semantics of the class or method. For example, the variable definition "`public String authorName = 'Erich Gamma'`" includes the information that the name about an author is stored in a string. Typically, in programming languages these identifiers are encoded in camelcases (e.g., "`MalformedMessageException`") or upper cases (e.g., "`CLASS_NAME`").

Fig. 5 shows parts of the original source code of the "String" Class from the standard Java libraries.



```
package java.lang;
import java.util.ArrayList;
...
/** The <code>String</code> class represents character strings. All
 * string literals in Java programs, such as <code>"abc"</code>, are
 * implemented as instances of this class. ...
 * @author  Lee Boynton
 * @version 1.152, 02/01/03
 * @see     java.lang.StringBuffer
 * @since   JDK1.0
 */
public final class String implements java.io.Serializable, Comparable,
CharSequence {
  /** The offset is the first index of the storage that is used. */
  private int offset;

  /** The count is the number of characters in the String. */
  private int count;
  ...

  /** Returns the index within this string of the first occurrence of
   * the specified character, starting the search at the specified
   * index. ...
   */
  public int indexOf(int ch, int fromIndex) {
   int max = offset + count;
   char v[] = value;

   if (fromIndex < 0) {
     fromIndex = 0;
   } else if (fromIndex >= count) {
     // Note: fromIndex might be near -1>>>1.
     return -1;
   }
   for (int i = offset + fromIndex ; i < max ; i++) {
     if (v[i] == ch) {
      return i - offset;
     }
   }
   return -1;
  } ...
}
```

Package information
Import information
JavaDoc documentation (for the class): includes HTML tags and JavaDoc tags (e.g. @author)
Class signature
JavaDoc documentation (for the attribute)
Attribute (signature)
JavaDoc documentation (for the method)
Method signature
Identifier internal attribute (here: „max")
Comment

Fig. 5 Java source code (String.java from the standard Java libraries)

4.2 Processing of Source Code

In order to use retrieval or mining techniques on source code, we recognized that the textual documents had to be further analyzed and processed. In programming languages such as Java, names and identifiers of classes or methods indicate their functionality and are written in so-called camelcase (e.g., "QueryResultWrapper"; see http://en.wikipedia.org/wiki/CamelCase). To include the information enclosed in these and other word constructs, filters have to be use to extract additional words (i.e., features) that characterize the document.

Therefore, we developed and adapted several techniques to preprocess source code. The preprocessing of source code in our code warehouse is partitioned in 9 phases as described below:

- First, we parse the textual data to identify tokens (i.e., everything that is not divided by whitespaces) and obtain a stream of these tokens that are processed by the following filters.
- The first filter decomposes identifier tokens `java.sql.ResultSet` into their subtokens `java`, `sql`, and `ResultSet` before returning them to the next filter.
- The next Filter splits *camelcased* tokens like `ResultSet` into the subtokens `Result` and `Set` as well as `IOException` into `IO` and `Exception`. It also has the task that uppercase abbreviations like `URL` are not splitted, titlecase tokens like `DATA_DIRECTORY` are broken into `DATA` and `DIRECTORY`, and that digits in tokens are associated with the previous subtoken (e.g., `Index6pointer` is broken into `Index6` and `pointer`).
- After the camelcase filter we change all uppercase characters in a token to *lowercase* (e.g., `DATA`, `data`, or `Data` are all changed to `data`) in order to normalize different writing or coding styles (e.g., typically, constants in Java are written in titlecase).

- Then we use the *cleaning filter* to remove unnecessary punctuation characters like commas or semicolons at the start or end of the token that might have been inserted at formulas or (for example, `name=` or `’rech’` from an expression like `int name=‘rech’` are changed to `name` and `rech`). Special characters that represent multiplications, equals, additions, subtractions, or divisions from formulas should have been eliminated in this process (e.g., from `a =b * c +d` only `a`, `b`, `c`, and `d` should get through).
- As detached *numbers* typically do not carry any meaning the next filter identifies and removes tokens that, in the specified programming language, represents numbers like `0x000`, `.9`, or `-200` as well as Unicode characters like `\u0123`.
- After the cleaning we use a programming-language-specific stopword filter to remove reserved words that are typically included in every code fragment of this language. For example, *Java stopwords* (a.k.a. reserved word) are `abstract`, `package`, or `boolean`.
- After programming-language specific stopwords are removed we also remove natural-language-specific stopwords from the token stream. Currently, we only remove *English stopwords* (e.g., `and`, `into`, or `will`) as source code is almost exclusively written with English acronyms and comments.
- Finally, we use the standard *Porter stemmer* [Porter 1980] to stem the remaining tokens (i.e., removing endings with `ed` or `ing` like in the words `generated` or `billing`) and reduce the number of available features with similar meanings

5 Evaluation

Open source repositories like Sourceforge consist of large amounts of data. Currently, the Sourceforge repository alone comprises of 103.094 projects of which 15.461 are using the Java programming language. Furthermore, every project consists of different versions and releases that represent the change and extension of the system over time.

From this mass of information we only used 23 projects including the java standard libraries. These 23 projects with 748 packages, 13.869 classes, and 110.083 methods for one (i.e., the last) build of the software system. The data encompassed 78.4 MB stored in the class table. In the mean we have 30 packages per project, 18 classes per package and 8 methods per class.

Based upon the data from this small extract of sourceforge we assume that the 15.461 Java projects consist of about 4,7 GB of data per version. If they are similarily distributed this would summarize to 500.000 packages, 8.5 million classes, and 66 million methods for each version and build. From our experience with software systems we assume roughly 10 releases and 100 versions per system in the mean.

5.1 Experimental Design

To evaluate the difference between unprocessed und preprocessed source code we constructed two types of queries from classes randomly drawn from the database. In order to spread the result we draw 100 classes and constructed 200 queries. The goal was to determine if a query on the preprocessed index will generate a result that lists the queried class with a lower rank than on the unprocessed index.

Two different *strategies* were used to construct the queries. First, we used the keywords from the name (i.e., identifier) of the class by extracting single words or abbreviations as described in section 4.2. This simulates a query where only few characteristics about a class (to be developed) are known. Second, we used all terms in a class and preprocessed them in the same way as described in section 4.2. That many characteristics about a class might be available if an already existing system is about to be re-engineered or if a class that is in development is incrementally extended. Examples for the named queries are stated in Table 1.

Table 1 Preprocess examples for Name-Queries

Name before preprocessing	DefaultGraphEdgeRenderer
Name after preprocessing	default graph edge renderer

To compare both approaches we applied the queries to search in a) an index based upon the unprocessed and b) an index based on the preprocessed data. The *precision* of the search is calculated as shown in Form. 1 using the position (i.e., rank) of the original class that was used to create the query in the result set.

$$P(q) = 1 - \frac{rank(q)}{hits(q)}$$

Form. 1 Calculation of the precision of query q

As the definition of a function to decide if a class is relevant or similar to the query the *recall* is either 0 (class not in result list) or 1 (class in result list).

5.2 Data Analysis and Interpretation

In the following we present the results using the two queries on the preprocessed and unprocessed index as described above. The unprocessed index consists of 263.396 terms from 13.869 classes in 116 MB while the preprocessed index consists of 17.365 terms from 13.869 classes in 86 MB.

As shown in Fig. 6 with queries using the whole class and Fig. 7 with queries using only terms from the name the preprocessed index performs better than the unprocessed one. Only 5% of the code-queries were not found at all (i.e., were not elements of the resultset) using the preprocessed index while 17% were not found using the unprocessed index.

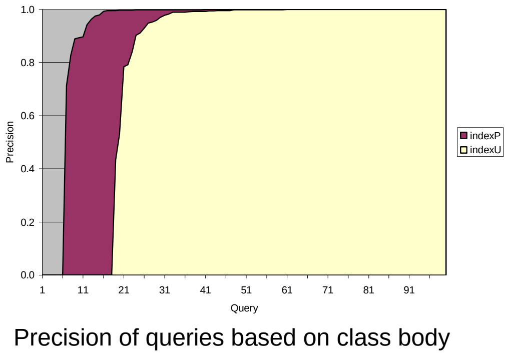


Fig. 6 Precision of queries based on class body

If the name-queries are applied to the indexes 45% of the name-queries were not using the preprocessed index and 93% were not found using the unprocessed index.

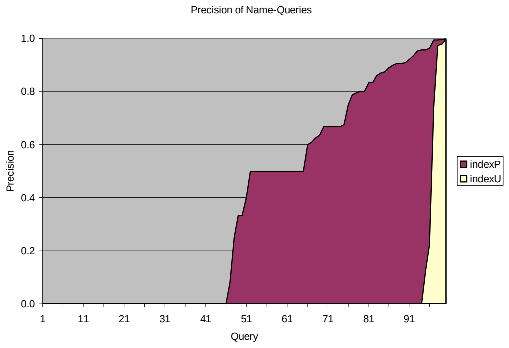


Fig. 7 Precision of queries based on class names

If comparing the queries on the same indexes as depicted in Fig. 8 and Fig. 9 it clearly shows that the queries based on all terms of the class are far more efficient than queries only based on the class names.

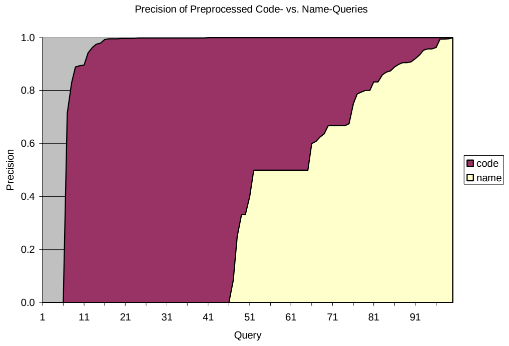


Fig. 8 Precision of preprocessed code queries vs. name queries

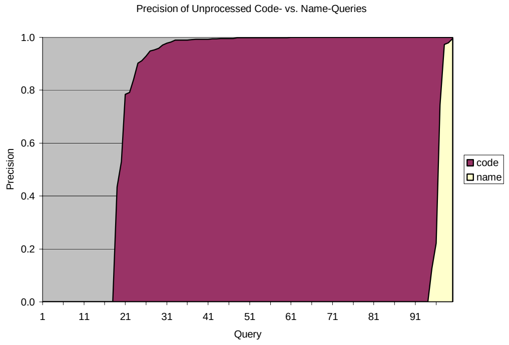


Fig. 9 Precision of unprocessed code queries vs. name queries

The comparison of the aggregated characteristics of precision from the studies are shown in Fig. 10. The minimum precision is always 0 as in all studies some queries resulted in a result set without the original class. As many queries returned the searched class as the first element the maximum in all studies is 1.

On average the precision of code-queries returned the best results while searching on the preprocessed data code-queries have a 13% higher precision than on the unprocessed index. The name-queries even have 33% higher precision on the preprocessed index than on the unprocessed index. Mode shows that the most common value using the name-queries is 0 and 1 using the code-queries. The median for the code queries draws a similar picture – more than 50% of the code-queries have a precision of 1 while name-queries on the unprocessed index have a precision of 0 and on the preprocessed index a precision of 0.37.

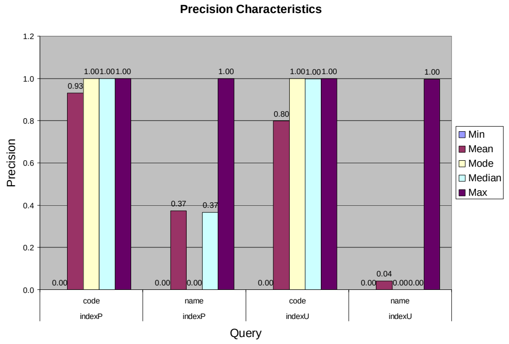


Fig. 10 Precision Characteristics

6 Conclusion

Recapitulating, we described the morphology and complexity of object-oriented source code that we use in our approaches for preprocessing object-oriented source code for code retrieval. We showed that the preprocessed index increases the precision of the search by at least 13% for queries encompassing a whole class and 33% for queries consisting of the class name – even if the precision only reaches 37% in the last case.

Although we can integrate and use the preprocessed index of this complex information for several applications we currently do not know if the system scales to large amounts of source code in terms of performance and space requirements and how we integrate additional information such as annotations, experiences, or inspection reports. Yet another obstacle is the maintenance of source code in our code warehouse as changes to software systems are common and they invalidate the meaning of experiences or annotations to older versions of source code. In a typical text retrieval context (i.e., web search engine) this is similar to the question if old or defunct web pages should be integrated into the retrieval process for current pages.

Furthermore, we plan to use the retrieval engine in order to support software architects during software design so that information about the planned system (e.g., in a class diagram) will be used to synthesis a query in order to retrieve software systems (i.e., subsystems or classes) that can be reused.

References

[Althoff et al. 1999]K.-D. Althoff, A. Birk, S. Hartkopf, W. Muller, M. Nick, D. Surmann, and C. Tautz, "*Managing software engineering experience for comprehensive reuse*," presented at SEKE'99. Eleventh International Conference on Software Engineering and Knowledge Engineering., Skokie, IL, USA, 1999.

[Basili et al. 2002] V. Basili, P. Costa, M. Lindvall, M. Mendonca, C. Seaman, R. Tesoriero, and M. Zelkowitz, "*An experience management system for a software engineering research organization*," presented at Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop, 2001, 2002.

[Basili et al. 1992] V. R. Basili, G. Caldiera, and G. Cantone, "*A reference architecture for the component fac-*

tory," ACM Transactions on Software Engineering and Methodology, vol. 1, no. 1, pp. 53-80, 1992.

[Basili et al. 1994] V. R. Basili, G. Caldiera, and H. D. Rombach, "*Experience Factory*," in Encyclopedia of Software Engineering, vol. 1, J. J. Marciniak, Ed. New York: John Wiley & Sons, 1994, pp. 469-476.

[Basili & Rombach 1991] V. R. Basili and H. D. Rombach, "*Support for Comprehensive Reuse*," Software Engineering Journal, vol. 6, no. 5, pp. 303-16, 1991.

[Beck 1999]K. Beck, "*eXtreme Programming eXplained: Embrace Change*." Reading: Addison-Wesley, 1999.

[Brin & Page 1998] S. Brin and L. Page, "*The Anatomy of a Large-scale Hypertextual Web Search Engine*," presented at 7th International World Wide Web Conference, Brisbane, Australia, 1998.

[Browne et al. 1998] S. Browne, J. Dongarra, J. Horner, P. McMahan, and S. Wells, "*Technologies for repository interoperation and access control*," Proceedings of Digital Libraries '98, Pittsburgh, PA, USA, 23 26 June 1998 * New York, NY, USA: ACM, 1998, p 40 8, no., 1998.

[Carro 2002] M. Carro, "*The AMOS Project: An Approach To Reusing Open Source Code*," presented at First CologNet Workshop on Component-Based Software Development and Implementation Technology for Computational Logic Systems, Madrid, Spain, 2002.

[Cinneide et al. 2004] M. O. Cinneide, N. Kushmerick, and T. Veale, "*Automated Support for Agile Software Reuse*," in *ERCIM News*, 2004.

[Dai et al. 2004] H. Dai, W. E. I. Dai, and G. Li, "*Software Warehouse*," International Journal of Software Engineering and Knowledge Engineering 14, vol. 04, no. 395-406, 2004.

[Feldmann 1999] R. L. Feldmann, "*On developing a repository structure tailored for reuse with improvement*," presented at Workshop on Learning Software Organizations (LSO) co-located with the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslautern, Germany, 1999.

[IBM 1994] IBM, "*Software Reuse: Overview and ReDiscovery*," IBM, ISBN 0738405760, 1994.

[Inmon 1996] W. H. Inmon, "*The Data Warehouse and Data Mining*," Communications of the ACM, vol. 39, no. 11, pp. 49-50, 1996.

[Jedlitschka et al. 2002] A. Jedlitschka, K.-D. Althoff, B. Decker, S. Hartkopf, M. Nick, and J. Rech, "*The Fraunhofer IESE Experience Management System*," KI, vol. 16, no. 1, pp. 70-73, 2002.

[Porter 1980] M. F. Porter, "*An algorithm for suffix stripping*," Program, vol. 14, no. 3, pp. 130-137, 1980.

[Ruhe & Bomarius 1999] G. Ruhe and F. Bomarius, "*Proceedings of Learning software organizations (LSO): methodology and applications*," presented at 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslautern, Germany, 1999.

[Shklar et al. 1994] L. Shklar, S. Thattle, H. Marcus, and A. Sheth, "*The InfoHarness Information Integration Platform*," presented at The Second International WWW Conference `94, Chicago, USA, 1994.

[Zündorf et al. 2001] B. Zündorf, H. Schulz, and D. K. Mayr. *SHORE – A Hypertext Repository in the XML World*. Retrieved 3rd January, 2005, from http://www.openshore.org/A_Hypertext_Repository_in_the_XML_World.pdf, (2001).