Morphology, Processing, and Integrating of Information from Large Source Code Warehouses for Decision Support

JÖRG RECH

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, 67661 Kaiserslautern, Germany
joerg.rech@iese.fraunhofer.de, http://www.iese.fraunhofer.de

A chapter proposal for the upcoming book,
"Processing and Managing Complex Data for Decision Support"

Morphology, Processing, and Integrating of Information from Large Source Code Warehouses for Decision Support

Abstract: *Today, source code occurs in diverse programming languages with documentation in miscellaneous standards, comments in individual styles, extracted metrics, or associated test cases that is hard to exploit through information retrieval or knowledge discovery techniques. Typically, the information about object-oriented source code for a software system is distributed over several different sources that make its processing complex. In this chapter we describe the morphology of object-oriented source code and how we (pre-) process, integrate and use it in software engineering for knowledge discovery in order to support decision making about the refactoring, reengineering, and reuse of software systems.*

Keywords: *complex data integration, complex data mining, knowledge discovery in source code, source code structure, source code retrieval, source code mining, software refactoring, software reuse*

INTRODUCTION

Traditionally, databases storing software artifacts were used to store manual 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, that is very similar to the Data Warehouse framework described by Inmon (Inmon, 1996). Operational configuration management systems (CMS), similar to data marts, are tapped to integrate software artifacts in the code warehouse. Typically, these software repositories consist of a plethora 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 and varying languages and formats into the code warehouse. Due to the astonishing success and propagation of open source software (OSS) and large OSS repositories like Sourceforge (cf. http://www.sourceforge.net) many CMSs 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.

But as code warehouses and single software systems are getting larger the more complex is it to decide on changes or enhancements. The complexity and dissimilarity of the source code has to be unified in order to adjust knowledge discovery (KDD) or information retrieval (IR) techniques. Tasks like (agile) software reuse, refactoring, or reengineering are getting increasingly complex and intensify the need for decision support in SE (Ruhe, 2003). Techniques from Artificial Intelligence (AI) are being used to support managerial decisions (e.g., where to focus testing effort) as well as design decisions (e.g., how to structure software systems) (Rech & Althoff, 2004). Several approaches for the discovery of knowledge to support decisions in reuse and quality improvement have been developed (Rech, Decker, & Althoff, 2001) but the support of decisions for reuse and refactoring based on large code warehouses is still unsatisfactory.

The architecture of our code warehouse as depicted in Figure 1 explains the flow of complex data like source code from a configuration management system (CMS) repository into our code warehouse and beyond to support decisions. Data sources with source code content are typically

known as configuration management systems (CMS) with the well-known representatives CVS, Subversion, or SourceSafe (Frühauf & Zeller, 1999). Typically, they manage several versions of a software system in order to step back to previous versions if unsolvable problems occur. As shown in Figure 1, we extract source code via wrappers from a CMS, 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.

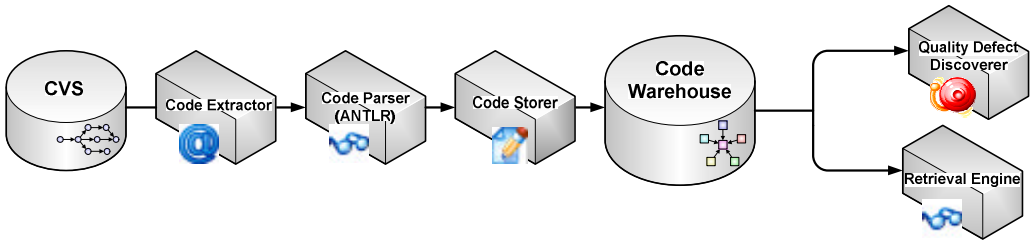

Figure 1. The simplified data flow in the Code Warehouse

After the information is stored in the code warehouse we use KDD and software analysis techniques in order to discover quality defects in a software system or retrieve source code or subsystems that might be used to build or improve a new software system. The extracted defects or similar software artifacts are than used to support decision making about the refactoring, reengineering, and reuse in software engineering.

In the remainder of this chapter we describe the morphology and complexity of object-oriented source code, how we integrate information from different sources, how we (pre-) process the data, and how we use it in software engineering for defect discovery and source code retrieval.

BACKGROUND

The reuse of existing knowledge and experience is one of the fundamental parts 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 we had to rebuild and relearn them again and again.

The discipline of *Software Engineering (SE)* was born 1968 at the NATO conference in Garmisch-Partenkirchen, Germany (Simons, Parmee, & Coward, 2003), where the term “software crisis” was coined to describe the increasing lack of quality in software systems that were continuously growing in size. On the same conference the methodic reuse of software components was motivated by Dough McIllroy (McIllroy, 1968) to improve the quality of large software systems by reusing small high-quality components.

Today, *reuse-oriented software engineering (ROSE)* covers the process of development and evolution of software systems by reusing existing software components. The goal is to the development of 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 than a redeveloped component.

Traditional Software Reuse

Since the eighties the systematic reuse and management of experiences, knowledge, products, and processes was refined and named *Experience Factory* (EF) (V. R. Basili, G. Caldiera, & H. D. Rombach, 1994). This field, also known as *Experience Management* (Jedlitschka et al., 2002) or *Learning Software Organization* (LSO), 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, Caldiera, & Cantone, 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 in the focus of ROSE. Today, the comprehensive reuse of all software artifacts and experiences from the software development process increases in popularity (Basili & Rombach, 1991). Beside source code artifacts like requirements, design document, test cases, process models, quality models, best practices (e.g., Design Patterns), etc. 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.

Repositories for Software Reuse

Until today, many organizations have developed software repositories with specific data models and interfaces for their own needs. While the content in software companies is often domain-specific, dependent from the strategic goals, and centered around source code repositories in software consulting organizations additionally manage domain- and artifact-encompassing repositories.

Current research projects concerned about software repositories are 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, Dongarra, Horner, McMahan, & Wells, 1998). In-house products from software companies are among others ReDiscovery from IBM (IBM, 1994), the Workstation Software Factory (WSF) Repository from Bellcore (Shklar, Thattle, Marcus, & Sheth, 1994), the Semantic Hypertext Object Repository (SHORE) from sd&m (Zündorf, Schulz, & Mayr, 2001), or the SEEE from Fraunhofer IESE (Althoff et al., 1999).

All these approaches are based on manual classification of code and use either the classification values or textual descriptions (e.g., man-pages) for retrieval. A new approach is the Software Warehouse concept (Dai, Dai, & Li, 2004) that builds a framework for retrieval and mining activities but 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., the domain feature).

Up to now relatively few approaches have been undertaken to exploit the information hidden in source code on a large scale. Especially the success of open source software and the resulting availability of massive amounts of source code enabled this development.

- A very similar commercial project to ours 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 (e.g., ASP, C, C#, C++, Delphi, Fortran, Java, JavaScript, Perl, PHP, Python, Ruby, SQL, Tcl, VB & VB.NET) encompassing about 125 million lines of code. As no information was published about the inner working of Koders we can only assess its processes by its offered functionality. The results presented after a free-text search indicate that Koders not only indexes whole files containing classes but also parses its content to identify methods and fields (a.k.a. attributes) as well as license information. Furthermore, it measures the lines of code (LOC) and calculates the cost to reproduce the code based on approximations of how many LOC are produced per month and the labor cost per month. In addition, one can browse through the file structures of the individual projects and identify or download parts of the sourcecode.

- The research project **AMOS** (http://www.clip.dia.fi.upm.es/~amos/AMOS/) follows a similar approach but uses a defined taxonomy (i.e., an ontology) for search terms to enable 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 to integrate several single packages in larger packages encompassing the whole search query. As it only searches over the signature of the code one can specify very exact queries if he knows what to look for. A more exploratory search based on code description, comments, or identifiers is not possible. Sadly, it seems not to work anymore as the search produces no results.
- 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 like title, author, description (e.g., the javadoc), language, URL, and the date it was added to the repository. Currently, it has indexed 30767 searchable and 12873 browseable resources from free repositories in 9 different programming languages (e.g., C, C++, Java, Perl, Assembler, JavaScript, ASP, PHP, and XML) based on comments in the source code as well as several text based sources (i.e., Web Sites, Research Papers, and Articles). While this approach is great for programmers wanting to search in API's and one can even access the source code directly the user gets no information on how to use it in a real context.
- 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 sourcecode from about 20 free java repositories and its search is based on the parsed source code. From several trials with the search we assume that it seems to have some problems in the search engine as some results are found even if they don't have any keywords included (e.g., search for quicksort) and camel cases seems not to be supported. It appears that source code from projects is parsed and indexed ignoring any class or method boundaries.
- Yet another code directory is **CodeArchive** (http://www.codearchive.com/) that offers a topic and domain specific browsing over 12 programming languages (Visual Basic, VB.NET, Java, JavaScript, Tcl/Tk, Linux, Delphi, PHP, Perl, C #, ASP, C/C++) and various

application domains like (Controls, Databases, Encryption, Algorithms, Coding Standards, Applets, Games, etc.). Projects and fragments were manually entered by voluntaries and the results after browsing or searching the archive include links to the source code or a compressed archive, requirements, and comments. Sadly, there is only a small amount of projects included in the repository (e.g., 46 Java projects) that results in incomprehensive search and might be retraced on the manual entry of project.

- Finally, the project **JDocs** (http://www.jdocs.com/) that's based on the open-source project **Ashkelon** (http://ashkelon.sourceforge.net/) represents a type of repository that provides a knowledge base defined around the major Java API's (Application Programming Interface). It does not include source code but give access to a collection of currently 132 API's from java frameworks and libraries. These APIs are built from the javadoc of the projects and are made searchable by a lucene powered engine. Additionally, it is possible for users to annotate each individual project, class, field, or method so that the knowledge in the javadoc API documentation can be enhanced by user contributed questions, answers, tips, links, example code and other relevant information. While this approach is great for programmers wanting to search in API's two problems remain. First, one can not access directly the source code described by an API element and second even if one finds a relevant class or method he has no information how to use it in a real context.

Agile Software Development & 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 esp. extreme programming (XP)) are based upon 12 principles (Beck, 1999). We mention four of these principles as they are relevant to our work. *The Planning Game* is the collective planning of releases and iterations in the agile development process and necessary to quickly determine the scope of the next release. *Small releases* are used to develop a large system by first putting a simple system into production and then releasing new versions in short cycles. *Simple design* means that systems are built as simple as possible, and complexity in the software system is removed if at all possible. *Refactoring* or the restructuring of the system without changing its behavior is necessary to remove qualitative defects that are introduced by the quick and often unsystematic development.

Traditional software reuse initiatives and approaches that were developed for process-driven software development are inadequate for these highly dynamic processes, where the software cannot be developed for reuse and reuse cannot be planned in advance. Teams and organizations developing with agile methods need heavily automated tools and techniques that support their work without consuming much time. Therefore, *agile software reuse* is a fairly new area where minimal invasive techniques are researched to support software engineers (Cinneide, Kushmerick, & Veale, 2004). Especially in the refactoring phase where the software is revised automation can be used to detect *quality defects* like code smells (Fowler, 1999), antipatterns (Brown, 1998), design flaws (Riel, 1996), design characteristics (Whitmire, 1997), or bug patterns (Allen, 2002). Here techniques from KDD support refactoring of software systems (Rech, 2004) and techniques from knowledge management can foster experience based refactoring (Rech & Ras, 2004).

Refactoring and Quality Defect Discovery

The primary goal of agile methods is the rapid development of software systems that are continuously adapted to customer requirements without large process overhead. Refactoring is typically done between development cycles and is the manual process to remove or weaken quality defects in order to improve the quality of software systems. Due to the fact that activities in software product maintenance account for the majority of the cost in the software life-cycle (Bennett & Rajlich, 2000) refactoring is an effective approach to prolong the software lifetime and to improve its maintainability. Especially in agile software development, methods as well as tools to support refactoring become more and more important (Mens, Demeyer, Du Bois, Stenten, & Van Gorp, 2003).

Current research in the field of software refactoring is very active and has identified the use of metrics for refactoring as an important research issue (Mens et al., 2003). Previous research in the sub-field refactoring has resulted in behavior-preserving approaches to refactor object-oriented software systems in general (Opdyke, 1992), tool support to automatically refactor or restructure applications (Griswold, 1991; Roberts, 1999), or methods for pattern based refactoring (Cinneide, 2000).

Today, various methods for the discovery of quality defects are known. For example, inspections are used to discover defects in early development phases, code analysis is used to quantify code characteristics, testing is used to detect functional defects after implementation, and coding standards are used to ensure quality attributes and prevent quality defects. Research by Simon on code smell discovery resulted in metrics-based and visually supported quality assurance with a similarity measure (Simon, 2001) to reclassify classes and minimize inter-class coupling. Zimmermann and colleagues pursue the goal to support developers in maintenance activities by using a technique similar to case-based reasoning. In order to support software maintainers with related experiences in form of association rules about changes in software systems they mined association rules from a versioning system by collecting transactions including a specific change (Zimmermann, Weißgerber, Diehl, & Zeller, 2004). Demeyer et al. proposed a framework to predict refactoring activities with time series analysis based on several metrics (Demeyer, Ducasse, & Nierstrasz, 2000). Grant and Cordy proposed another approach to automatic code smell detection by supporting a software developer in the rule-based discovery of code smells as well as the belonging refactorings (Grant & Cordy, 2003). Finally, van Emden and Moonen analyzed code for two code smells and visualized them for manual discovery (Moonen, 2002).

All of the mentioned research approaches support specific kinds of manual and (semi-) automatic quality defect discovery techniques. None of these approaches are comprehensive, what metrics can be used to discover quality defects, or describe how refactoring experiences could be reused. Neither are the effects of refactoring activities on all software qualities or code examined. Hence, in our approach we develop models to detect quality defects with metrics or rules and try to support defect correction activities with tailored experiences.

Knowledge Discovery in Databases and Information Retrieval

Knowledge discovery in databases (KDD) (Klösgen & Zytkow, 2002) is concerned with the discovery of previously unknown information from large datasets. The discovery of knowledge is a process that can be divided into the five sub-processes Selection, Preprocessing, Transformation, Mining, and Interpretation (e.g., Validation and Representation) (Fayyad, Piatetsky, & Smyth, 1996). These sub-processes underpin the importance of clean data for the

mining process (e.g., numerical without missing data) and the need for representation of clear valid knowledge (e.g., visualization of clusters). The goals of KDD can be divided into the groups *cluster discovery* (i.e., answering "Are there related elements (e.g., methods in other systems)?"), *class discovery* (i.e., answering "How to classify elements?"), *association discovery* (i.e., answering "Do causal relations exist between elements (e.g., CMS transactions)?"), *model discovery* (i.e., answering "Do valid causal models exist (e.g., for maintenance effects)?"), *trend discovery* (i.e., answering "What will happen in x days (e.g., if using refactoring Y)?"), *pattern discovery* (i.e., answering "Are there typical reoccurring structures (Are there reoccurring patterns, e.g., design patterns?"), and *correlation discovery* (i.e., answering "Do correlations between measured variables (e.g., between effort and LOC) exist?").

Today, the term *data mining* (Witten & Frank, 1999) is often used as a synonym for KDD. While Data Mining (Witten & Frank, 1999) is the detection of "nuggets" in numerical data, various forms of mining exist that examine different types of data. For example, text mining (Berry, 2004) focuses on the extraction of knowledge from collections of long texts (e.g., books), while web mining (Kosala & Blockeel, 2000) (Hsu, 2002) specializes in typically small hypertexts (e.g., web pages), clickstreams, or log data.

In this context, we classified *code mining* as a subfield of web and text mining. Since source code can be seen as hypertext due to the fact that relations between classes or methods (e.g., function calls or inheritance relations) are analog to links, techniques from KDD and especially web mining can be used in a similar way to exploit source code.

Mining in Software Repositories

Today, many application areas for KDD in SE have been established in fields like quality management, project management, risk management, software reuse, or software maintenance. Techniques like neural networks, evolutionary algorithms, or fuzzy systems are increasingly applied and adapted for specific SE problems. They are used for the discovery of defect modules to ensure software quality or to plan software testing and verification activities to minimize the effort for quality assurance (Khoshgoftaar, 2003; Lee, 2003; Pedrycz & Peters, 1998).

For example, Khoshgoftaar et al. applied and adapted classification techniques to software quality data (Khoshgoftaar, Allen, Jones, & Hudepohl, 2001). Dick researched determinism of software failures with time series analysis and clustering techniques (Dick, 2002). Cook and Wolf used the Markov approach to mine process and workflow models from activity data (Cook & Wolf, 1998). Pozewauning examined the discovery and classification of component behavior from code and test data to support the reuse of software (Pozewaunig, 2001). Michail used association rules to detect reuse patterns (i.e., typical usage of classes from libraries) (Michail, 2000). As an application of KDD in software maintenance, Shirabad developed an instrument for the extraction of relationships in software systems by inductive methods based on data from various software repositories (e.g., update records, versioning systems) to improve impact analysis in maintenance activities (Shirabad, 2003). Zimmermann and colleagues pursue the same goal using a technique similar to CBR. In order to support software maintainers with related experiences in form of association rules about changes in software systems they mined association rules from a versioning system by collecting transactions including a specific change (Zimmermann et al., 2004). Morasca and Ruhe built a hybrid approach for the prediction of defect modules in software systems with rough sets and logistic regression based on several metrics (e.g., LOC) (Morasca & Ruhe, 2000).

Future research in this field is concerned with the analysis of formal project plans for risk discovery, to acquire project information for project management, or directly mine software representations (e.g., UML, sourcecode) to detect defects and flaws early in development.

MORPHOLOGY OF OBJECT-ORIENTED SOURCE CODE

Until today, the function of a code element cannot be extracted from the syntax of arbitrary source code. Nevertheless, valuable information can be extracted from associated and internal sources. While object-oriented 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. For example, Java classes typically contain methods, attributes, comments, or documentation in JavaDoc. Additional information like introductory, readme, API, or license documents is available from files residing in the project (or systems) repository. In Figure 2, 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). We call these sets of features *feature-spaces*. For further retrieval or mining processes, these feature-spaces have to be integrated to improve the characterization of raw source code data.

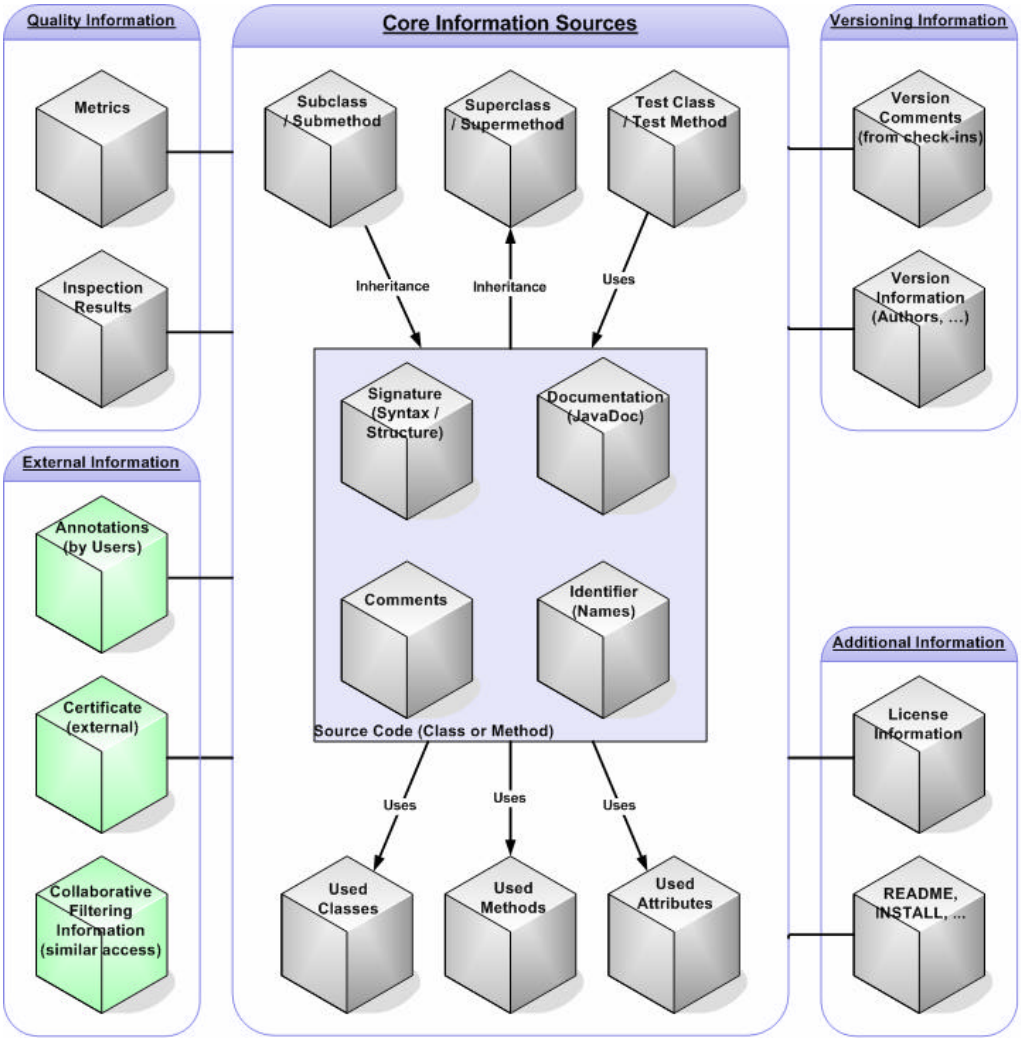

Figure 2. Structure and hierarchy of object-oriented source code

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

Object-oriented source code

Object-oriented software systems consist of objects that package data and functionality together into units that represents objects of the real world (e.g., a sensor or string). These objects can perform work, report on and change its internal state, 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 its state. Similarly, more abstract building blocks like 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 like a sensor is exchanged).

In software reuse several of these units might be of interest to a potential user. He might want to reuse a whole database (e.g., mySQL or postgeSQL) or need a solution to implement a fast sorting algorithm (e.g., quicksort). Figure 3 shows several blocks that might be of interest to the user and 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.

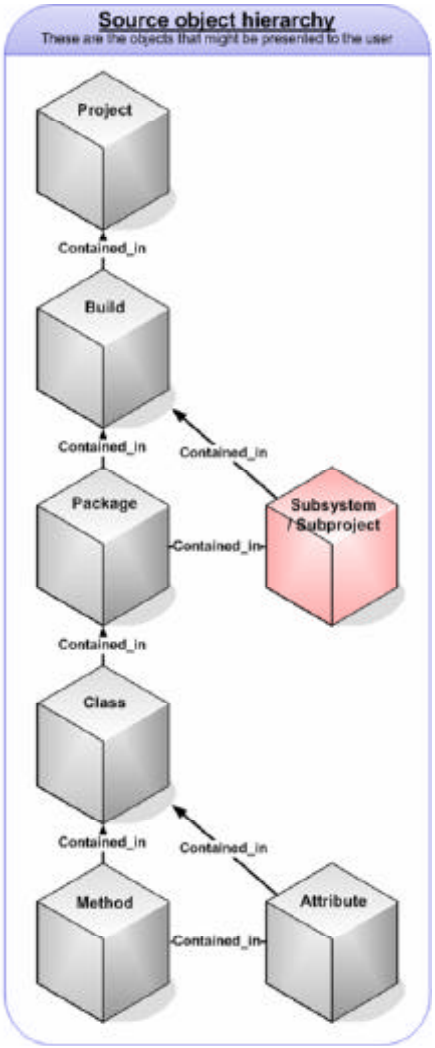

Figure 3. Source code related structure of software projects

Internal structure of source code

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 like *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).

Beside these external links source code contains several **internal information blocks**. As depicted in Figure 2 these blocks build the core information sources to describe classes in the following order:

1. **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 super-classes 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 so called *camel cases* (e.g., "`StringBuffer`") that has to been parsed and normalized (e.g., into "string" and "buffer").

2. **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 an 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).
3. **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.
4. **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 camel cases (e.g., "`MalformedMessageException`") or upper cases (e.g., "`CLASS_NAME`").

Figure 4 shows parts of the original source code of the "String" Class from the standard java libraries.

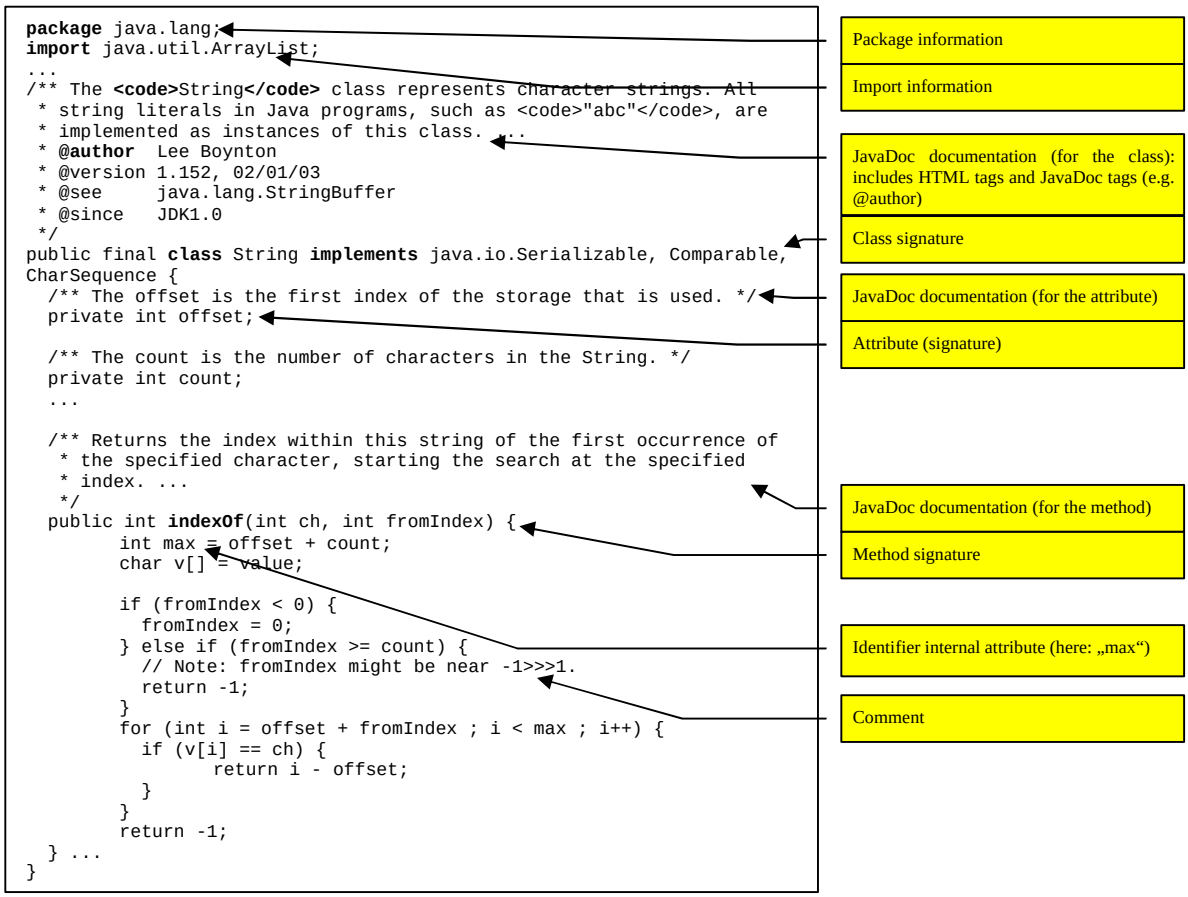

Figure 4. Java source code (String.java from the standard java libraries)

Typical information associated to source code

Beside the internal information additional information blocks can greatly contribute to the further processing of the source code. Software product *metrics* that are computed from the software system are used to describe the complexity (e.g., LOC, CC), coupling (e.g., CBO), cohesion (e.g., LCOM), or other characteristics of a method, class, package or project. Similarly, *inspection reports* are used to describe (potential) defects found during a manual inspection or audit. Furthermore, inspections can also be used to characterize the code on a subjective basis (e.g, rate the maintainability of a class on a scale from “very high” to “very low”).

License information, typically, stored in special files or in the header of class files describe the context and conditions in whithc the source code might be used or reused. In general there are three types of licences, one gives free hands in the reuse (e.g., LGPL), another requires that the authors of reused code are named and license information is passed on (e.g., SISSL), and finally one type requires that the resulting product that uses the source code has also be made open source (e.g., the GPL).

From configuration management systems like CVS additional information about the exact version or change comments can be extracted. They are helpful in hiding older versions or might include helpful keywords for retrieval processes.

PROCESSING AND INTEGRATION OF SOURCE CODE

As we have seen object-oriented source code consists of textual, categorical, and numerical data that can be exploited for further processing. But in order to use KDD or IR techniques on them different information blocks must be further pre-processed and integrated.

Based on the information inside the code warehouse that was parsed and interconnected as described by the source code we create an index that stores several information blocks for every source code artifact. For example, we store the name, type, code, comments, documentation, or signature after it is processed as described in the next subsection. Processing the code serve the identification, reduction, and cleaning of characterizing features of the source code or a information block within. The integration is used to increase the expressiveness of features, to increase the number of similar and maybe synonym features, and to enrich the representation of results given to the end-user.

Pre-processing source code for retrieval processes

In order to use retrieval or mining techniques on source code, the textual documents have to be further analyzed and processed. Source code, as many other semi-structured data sources, is enriched with special constructs and features with varying relevance.

Preprocessing of identifiers in source code

In programming languages like Java names and identifier of classes or methods indicate their functionality and are written in so-called “camelcase” (e.g., “QueryResultWrapper”). To include the information enclosed in these special word constructs, filters have to be use to extract additional words (i.e., features) to further characterize the document.

The preprocessing of source code in our code warehouse is partitioned in 9 phases as shown in Figure 5. 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 will split *camel-cased* tokens (“ResultSet”) into the subtokens “Result” and “Set” as well as “IOException” into “IO” and “Exception”. It also pays attention that upper-case abbreviations like “URL” are not splitted, title-case 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 camel-case filter we change all upper-case characters in a token to *lower-case* (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 title-case). Then we use the *cleaning filter* to remove unnecessary punctuation characters like “.”, “;”, “_”, etc. at the start or end of the token that might have been inserted at formulas or (e.g., “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 don’t 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", "do", "if", "package", "synchronized", "boolean", "double", "implements", "private", "this", "break", "else", "import", "protected", "throw", "byte", "extends", "instanceof", "public", "throws", "case", "false", "int", "return", "transient", "catch", "final", "interface", "short", "true", "char", "finally", "long", "static", "try", "class", "float", "native", "strictfp", "void", "const", "for", "new", "super", "volatile", "continue", "goto", "null", "switch", "while", "default", "assert", "get", "set", or "java". 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., "a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if", "in", "into", "is", "it", "no", "not", "of", "on", "or", "s", "such", "t", "that", "the", "their", "then", "there", "these", "they", "this", "to", "was", "will", "with") as source code is almost exclusively written with English acronyms and comments. Finally, we use the standard *Porter stemmer* 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.

```
public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(reader);
    result = new JavaIdentifierFilter(result);
    result = new JavaCamelCaseFilter(result);
    result = new LowerCaseFilter(result);
    result = new JavaCleaningFilter(result);
    result = new JavaNumberFilter(result);
    result = new JavaStopwordFilter(result);
    result = new StopFilter(result, stopWords);
    result = new PorterStemFilter(result);
    return result;
}
```

Figure 5. The processing sequence for java source code

Stemming and Anti-Stemming for source code

Beside the reduction of characterizing features from the source code we also extract special features from the token stream during the processing phases.

Identifiers that typically name classes or methods are used in import statements to help the compiler in identifying the exact element that should be used in the program. In this context, *Code Stemming* is the reduction of identifiers to their stem similar to normal language stemming (e.g., the porter stemmer) and is used to extend the query focus. An identifier that specifies the path to the exact class in a hierarchy of packages can be used to integrate the information from classes in higher package. For example, we exploit the identifier “java.util.Arrays” that points to the class “Arrays” by looking into classes and subpackages of the package “java.util”.

Code “*Anti-Stemming*” we understand the inflation of abbreviated words to the nearest full word in order to improve query results. For example, if we have a token “calTemp” and split it into the subtokens “cal” and “temp” that do not represent valid stems of natural words we use a

dictionary to find words that might be abbreviations for the subtokens (e.g., “calculate” and “calibrate”). After this inflation we additionally use a stemmer in order to reduce the now larger set of features.

Integration of distributed information

In order to exploit information from several sources similar to the pagerank algorithm (Brin & Page, 1998) used in google the relations to superclasses, test classes, or used classes are preserved to improve retrieval and mining results.

As depicted in Figure 2 and Figure 3 we use the signature, documentation, comments, and identifiers to describe a source code artifact like a class or method. At first we used this information to enable the retrieval by matching keywords from a search to the words in these four information blocks. But as keywords and natural text is typically very rare in source code we thought about how we could improve the retrieval. So we thought about using the ideas behind the PageRank algorithm to integrate information from associated source code artifact. In object-oriented programming language we have several typed relations like “inheritance”, “uses”, or “calls”. So it was straightforward to integrate these additional information as they seem to carry many information, sometimes about the context sometimes about the functionality. For example, a method that uses the class “java.awt.Graphics” and calls its method “drawOval” might be helpful if you search for a way to quickly learn how to draw something (esp. circles) in Java – even if the variable is named “g”.

Now on one hand we have method like “drawOval” that are called by many other methods that draw circles or ovals and include words that hint to their functionality (e.g., to draw something). Therefore, this method would increase the bond to the words defined in the calling methods. On the other hand we have methods that call many other methods (i.e., subfunctions) and aggregate their functionality into a higher functionality. Here the method not only is defined by its own keywords but also by the keywords described in their subfunctions.

For retrieval we use the SourceRank (SR) and SourceScore (SS) algorithms as shown in Formula 1 and Formula 2. The SourceScore algorithm is basically similar to the PageRank algorithm but does not calculates the weight from other SourceRank weight but from the SourceScore weight. In calculating the rank of an element (E_0, e.g., an class or method) it includes the sum of the SourceScore from all incoming (i.e., users of this element) elements (ISC) divided by the number of their outgoing connections to other source elements (OSC).

$$SR(E_0) = (1-d) + d * (\sum_{i \in ISC(E_0)} \frac{SS(E_i)}{OSC(E_i)})$$

SR: SourceRank
SS: SourceScore
ISC: Incoming Source Connections
OSC: Outgoing Source Connections
D: Dampening factor

Formula 1. The Source Rank Algorithm

$$SS(E_0) = \frac{SK(E_0)}{sw} + \frac{DK(E_0)}{dw} + \frac{CK(E_0)}{cw} + \frac{IK(E_0)}{iw}$$

SK: no. of Signature Keywords
DK: no. of Documentation Keywords
CK: no. of Comment Keywords
IK: no. of Identifier Keywords

sw, dw, cw, iw: weights for the signature, documentation, comments, and identifier

Formula 2. The Source Score Algorithm

Other information like license or versioning information is only associated to the software artifact and integrated into the representation (e.g., the result list of a retrieval process).

Furthermore, information about the quality of a software artifact like metrics or inspection reports are currently not integrated into the presentation or retrieval technique. We plan to modify the score or weight of an artifact based on its quality so that artifacts with higher quality are more dominant in the results.

DECISION SUPPORT FOR REFACTORING AND REUSE

In order to support decisions about what to reuse or refactor in a software system we developed several methods and techniques. Based on the previously described data structures, their relationship, and integration strategies we describe two environments to support decisions in software engineering.

Decision Support in Software Refactoring and Reengineering

Our approach encompasses several methods in order to support the decision where, when and in what sequence to refactor a software system as depicted in Figure 6. Beginning from the left upper corner counterclockwise knowledge about quality defects from defect discovery processes is used to retrieve experiences associated to similar defects from previous refactorings. These experiences are used to handle the quality defect in the defect removal phase. Additionally, suitable experiences are augmented by so called micro-didactical arrangements (MDA) that initiate learning processes and hence improve the experiences' understandability, applicability and their adaptability to the reuse context.

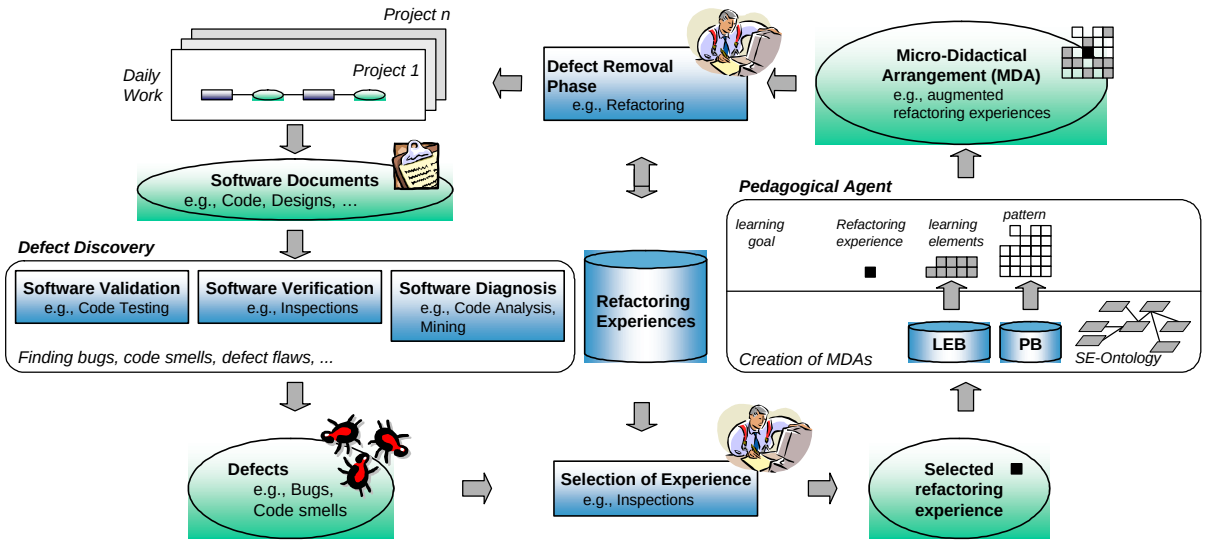

Figure 6. Experience-based Semi-Automatic Reuse of Refactoring Experiences

As shown in Figure 7, we define six phases, based on QIP (V. R. Basili, G. Caldiera, & D. Rombach, 1994), for the continuous automatic discovery of quality defects. First, we start with the definition of qualities that should be monitored and improved. For example, this may result in different goals (i.e., quality aspects) as, reusability demands more flexibility or "openness" while maintainability requires more simplicity. Phase two represents the application area for Knowledge Discovery in Databases (KDD). It is concerned with the measurement and

preprocessing of the source code to build a basis for quality defect discovery. Results from the discovery process (i.e., quality defects) are represented and priorized to plan the refactoring in phase three. Here, the responsible person has to decide which refactorings have to be executed in what configuration and sequence in order to minimize work (e.g., change conflicts) and maximize the effect on a specific quality. In phase four the refactoring itself is applied to the software system by the developer that results in an improved product. Phase five compares the improved with the original product to detect changes and their impact on the remaining system. Finally, in the sixth phase we report the experiences and data about the refactoring tasks, changes to the software system, and other effects to learn from our work and continuously improve the model of relationship between quality, refactorings, and quality defects.

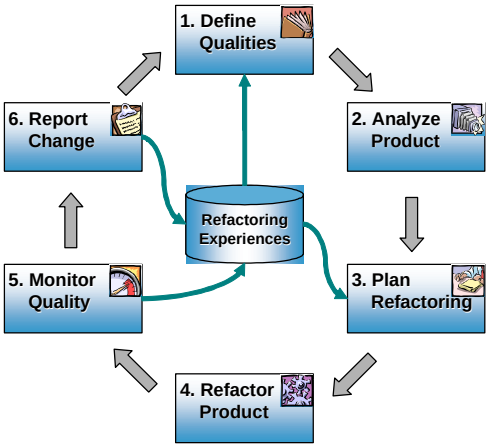

Figure 7. Quality-driven metrics-based refactoring

As indicated previously, the KDD sub-processes are grouped in phase two. We select source code from a specific build, preprocess the code and store the results in the code warehouse, analyze the data to discover quality defects, discover deviations from average behavior, cluster code blocks with severe or multiple quality defects, and represent discovered and priority sorted quality defects to the user.

For example, lets say we detect a method in an object-oriented software system that has a length of 300 LOC. As described in (Fowler, 1999), this is a code smell called “Long Method”. A long method is a problem esp. in maintenance phases as the responsible maintainer will have a hard time to understand the function of this method.

One suitable refactoring for the mentioned code smell might be the refactoring simply called “Extract Method”: the source code of the long method is reviewed to detect blocks that can be encapsulated into new (sub-)methods. Experiences about the “Extract Method” refactoring are used to support the decision where, when, how, and if the refactoring has to been implemented. For example, the developer might remark that every block of code that has a common meaning, and could be respectively commented, could also be extracted into several small methods. Furthermore, he might note that the extraction of (sub-) methods, from methods implementing complex algorithms, can effect performance requirements of the software system and therefore might not be applicable.

Additionally, the generation of new methods might create another smell called “Large Class” (i.e., the presence of too many methods in a class) which might complicate the case even more. Finally, the new experiences are annotated by the developer and stored in the refactoring experience base.

While this example only touches a simple quality defect and refactoring, more complex refactorings influence inheritance relations or introduce design patterns (Fowler, 1999).

Decision support in reusing software artifacts

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 like 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 like “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 the 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 he has to include 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 Figure 8 our system presents search results similar to google and includes information like 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 are 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.

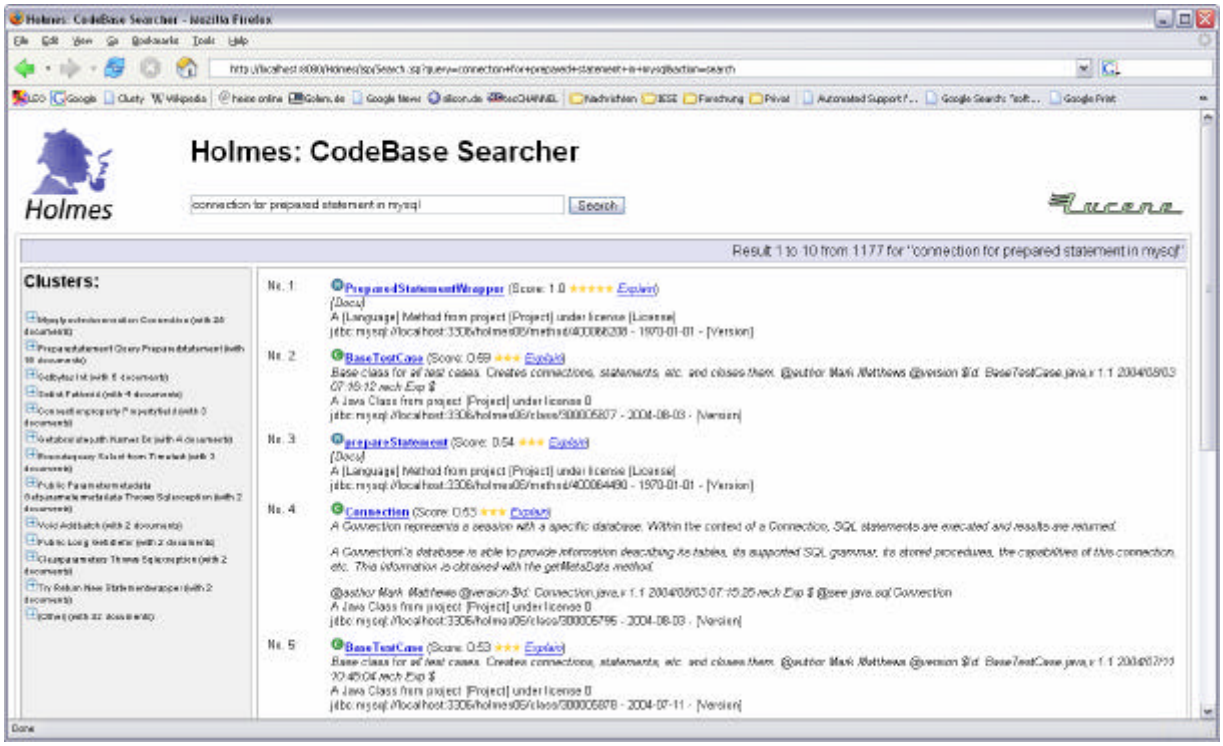

Figure 8. Screenshots from the Code Warehouse

CONCLUSION

Recapitulating, we described the morphology and complexity of object-oriented source code that we used in our approach in software engineering. Based on the definition of our data and its structure we described the processes how to integrate and (pre-) process this data from different sources. Finally, we gave two examples how we used this base of information for decision support in quality defect discovery and software reuse.

Although we can integrate and use this complex information for several applications we currently do not know if the system scales to large amounts of source code and how we integrate additional information like annotations, experiences, or inspection reports. Yet another obstacle is the maintenance of source code in our 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 "normal" 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.

We expect to further assist software engineers and managers in their work and in decision making. One burning problem is currently the feature selection and reduction in order to improve speed and preciseness of the techniques in use (esp. the SOMs). One approach to enlargen the base of stopwords will be the extensive analysis of source code from sourceforge to identify typical words used in most code artifacts (e.g., variables like "temp"). Planned extension to the

techniques will be *trend discovery* based on system changes to detect problems in the software system as early as possible and to alert the quality and project manager about deviations from the plan. *Pattern discovery* based on the structural information of object-oriented source code will be used to detect typical patterns in the organization of source code (and maybe even new "design pattern"). Furthermore, we plan to use the retrieval part 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

Allen, E. (2002). *Bug patterns in Java*. Berkeley: Apress, USA, New York, NY.

Althoff, K.-D., Birk, A., Hartkopf, S., Muller, W., Nick, M., Surmann, D., et al. (1999). Managing software engineering experience for comprehensive reuse. *Branden Univ.; Fraunhofer Inst,* Knowldege Syst. Inst.; Univ. Kaiserslautern; et al. - In Proceedings. SEKE'99. Eleventh International Conference on Software Engineering and Knowledge Engineering. - Skokie, IL, USA, USA Knowledge Syst. Inst, 1999, ix+1408 1910-1919, 1952 Refs.

Basili, V., Costa, P., Lindvall, M., Mendonca, M., Seaman, C., Tesoriero, R., et al. (2002). An experience management system for a software engineering research organization. *IEEE Computer*(NASA Goddard Space Flight Center), 29-35.

Basili, V. R., Caldiera, G., & Cantone, G. (1992). A reference architecture for the component factory. *ACM Transactions on Software Engineering and Methodology, 1*(1), 53-80.

Basili, V. R., Caldiera, G., & Rombach, D. (1994). The Goal Question Metric Approach. In *Encyclopedia of Software Engineering* (1st Edition ed., pp. 528-532). New York: John Wiley & Son.

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). Experience Factory. In J. J. Marciniak (Ed.), *Encyclopedia of Software Engineering* (Vol. 1, pp. 469-476). New York: John Wiley & Sons.

Basili, V. R., & Rombach, H. D. (1991). Support for comprehensive reuse. *Software Engineering Journal, 6*(5), 303-316.

Beck, K. (1999). *eXtreme Programming eXplained: Embrace Change*. Reading.

Bennett, K. H., & Rajlich, V. T. (2000, 2000). *Software Maintenance and Evolution: A Roadmap.* Paper presented at the Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland.

Berry, M. W. (Ed.). (2004). *Survey on Text Mining: Clustering, Classification, and retrieval*. New York: Springer.

Brin, S., & Page, L. (1998, 14-18 April 1998). *The anatomy of a large-scale hypertextual Web search engine.* Paper presented at the 7th International World Wide Web Conference, Brisbane, Australia.

Brown, W. J. (1998). *AntiPatterns : refactoring software, architectures, and projects in crisis*: Wiley.

Browne, S., Dongarra, J., Horner, J., McMahan, P., & Wells, S. (1998). 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.*

Carro, M. (2002, 19. - 20. September 2002). *The AMOS Project: An Approach To Reusing Open Source Code.* Paper presented at the First CologNet Workshop on Component-Based Software Development and Implementation Technology for Computational Logic Systems, Madrid, Spain.

Cinneide, M. O. (2000). *Automated Application of Design Patterns: A Refactoring Approach.* Ph.D. Thesis, Trinity College, Dublin.

Cinneide, M. O., Kushmerick, N., & Veale, T. (2004, July 2004). Automated Support for agile software reuse. *ERCIM News*.

Cook, J. E., & Wolf, A. L. (1998). Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology, 7*(3), 215-249.

Dai, H., Dai, W. E. I., & Li, G. (2004). Software Warehouse. *International Journal of Software Engineering and Knowledge Engineering 14, 04*, 395-406.

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2000). Finding refactorings via change metrics. *ACM. In: SIGPLAN Not. (USA), 35*(10), 166- 177.

Dick, S. H. (2002). *Computational Intelligence in Software Quality Assurance.* PhD Thesis, College of Engineering, University of South Florida.

Fayyad, U., Piatetsky, S. G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine, 17*(3), 37-54.

Feldmann, R. L. (1999). On developing a repository structure tailored for reuse with improvement. *Learning Software Organizations. Methodology and Applications. 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99. Proceedings, Kaiserslautern, Germany, 16 19 June 1999 * Berlin, Germany: Springer Verlag, 2000, p 51 71*.
Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code* (1st ed.): Addison-Wesley.
Frühauf, K., & Zeller, A. (1999). Software configuration management: state of the art, state of the practice. *System Configuration Management. 9th International Symposium, SCM 9. Proceedings (Lecture Notes in Computer Science Vol.1675) / Estublier, J.. Berlin, Germany, Germany: Springer Velag, 1999, viii+254 p.217 27, 37 Refs.*
Grant, S., & Cordy, J. R. (2003). *Automated Code Smell Detection and Refactoring by Source Transformation.* Paper presented at the International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), Victoria, Canada.
Griswold, W. G. (1991). *Program Restructuring as an Aid to Software Maintenance.* PhD Thesis, University of Washington, Washington.
Hsu, J. (2002). Web mining: A survey of World Wide Web data mining research and applications. *33rd Annual Meeting of the Decision Sciences Institute. San Diego, CA, USA, USA: Decison Sci. Inst, 2002, CD ROM p.6 pp., 50 Refs.*
IBM. (1994). *Software Reuse: Overview and ReDiscovery* (No. ISBN 0738405760): IBM.
Inmon, W. H. (1996). The Data Warehouse and Data Mining. *Communications of the ACM, 39*(11), 49-50.
Jedlitschka, A., Althoff, K.-D., Decker, B., Hartkopf, S., Nick, M., & Rech, J. (2002). The Fraunhofer IESE Experience Management System. *KI, 16*(1), 70-73.
Khoshgoftaar, T. M. (2003). *Software engineering with computational intelligence* (Vol. 731). Boston: Kluwer Academic Publishers.
Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., & Hudepohl, J. P. (2001). Data mining of software development databases. *Software Quality Journal, 9*(3), 161-176.
Klösgen, W., & Zytkow, J. M. (2002). *Handbook of data mining and knowledge discovery*. Oxford: New York.
Kosala, R., & Blockeel, H. (2000). Web Mining Research: A Survey. *SIGKDD Explorations, 2*(1), 1-15.
Lee, J. (2003). *Software engineering with computational intelligence*. Berlin: New York.
McIllroy, M. D. (1968, 7th to 11th October 1968). *Mass-produced software components.* Paper presented at the NATO Conference in Software Engineering, Garmisch, Germany.
Mens, T., Demeyer, S., Du Bois, B., Stenten, H., & Van Gorp, P. (2003). Refactoring: Current Research and Future Trends. *Electronic Notes in Theoretical Computer Science, 82*(3), 17.
Michail, A. (2000, Soc. Tech. Council on Software Eng.). *Data mining library reuse patterns using generalized association rules.* Paper presented at the Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000), New York.
Moonen, L. (2002). *Exploring Software Systems.* Ph.D Thesis, University of Amsterdam, Amsterdam, Netherlands.
Morasca, S., & Ruhe, G. (2000). A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. *Journal of Systems and Software, 53*(3), 225-237.
Opdyke, W. F. (1992). *Refactoring object-oriented frameworks.* Ph.D. Thesis, University Illinois at Urbana-Champaign, Urbana, Illinois.
Pedrycz, W., & Peters, J. F. (1998). *Computational intelligence in software engineering*. Singapore: River Edge N.J.
Pozewaunig, H. (2001). *Mining Component Behavior to Support Software Retrieval.* PhD Thesis, Universität Klagenfurt, Klagenfurt.
Rech, J. (2004). *Towards Knowledge Discovery in Software Repositories to Support Refactoring.* Paper presented at the Workshop on Knowledge Oriented Maintenance (KOM) at SEKE 2004, Banff, Canada.
Rech, J., & Althoff, K.-D. (2004). Artificial Intelligence and Software Engineering: Status and Future Trends. *Künstliche Intelligenz, 18*(3), 5-11.
Rech, J., Decker, B., & Althoff, K.-D. (2001). *Using Knowledge Discovery Technology in Experience Management Systems.* Paper presented at the Workshop "Maschinelles Lernen" (FGML), Universität Dortmund.
Rech, J., & Ras, E. (2004). *Experience-Based Refactoring for Goal-Oriented Software Quality Improvement.* Paper presented at the First International Workshop on Software Quality (SOQUA 2004), Erfurt, Germany.
Riel, A. J. (1996). *Object-oriented design heuristics*. Reading, Mass.: Addison-Wesley Pub. Co.
Roberts, D. B. (1999). *Practical Analysis for refactoring.* Ph.D. Thesis, University of Illinois at Urbana-Champaign.
Ruhe, G. (2003). Software engineering decision support - a new paradigm for learning software organizations. *Advances in Learning Software Organizations*, Maurer, F.. - Berlin, Germany, Germany Springer-Verlag, 2003, vi+2113 2104-2013, 2024 Refs.

Shirabad, J. S. (2003). *Supporting Software Maintenance by Mining Software Update Records.* PhD Thesis, University of Ottawa, Ottawa, Ontario, Canada.
Shklar, L., Thattle, S., Marcus, H., & Sheth, A. (1994, October 17-20). *The InfoHarness Information Integration Platform.* Paper presented at the The Second International WWW Conference `94, Chicago, USA.
Simon, F. (2001). *Meßwertbasierte Qualitätssicherung: Ein generisches Distanzmaß zur Erweiterung bisheriger Softwareproduktmaße (in German).* Ph.D. Thesis, Brandenburgische TU Cottbus, Cottbus.
Simons, C. L., Parmee, I. C., & Coward, P. D. (2003). 35 years on: to what extent has software engineering design achieved its goals? *IEE Proceedings Software*, *150*(6), 337-350.
Whitmire, S. A. (1997). *Object-oriented design measurement*
Witten, I. H., & Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations* (1st edition (October 11, 1999) ed.): Morgan Kaufmann.
Zimmermann, T., Weißgerber, P., Diehl, S., & Zeller, A. (2004, September 2003). *Mining Version Histories to Guide Software Changes.* Paper presented at the 26th International Conference on Software Engineering (ICSE), Edinburgh, UK.
Zündorf, B., Schulz, H., & Mayr, D. K. (2001). *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