SUITE

First Intl. Workshop on **S**earch-Driven Development – **U**sers, **I**nfrastructure, **T**ools and **E**valuation

http://smallwiki.unibe.ch/suite2009/

Vancouver, Canada, **May 16, 2009**
at **31st Intl. Conference on Software Engineering**

Workshop Co-Chairs

Sushil Bajracharya	University of California, Irvine, USA
Adrian Kuhn	University of Bern, Switzerland
Yunwen Ye	SRA, Inc, Japan

Program Committee

Andrew Begel	Microsoft Research, USA
Harald Gall	University of Zurich, Switzerland
Mark Grechanik	Accenture Technology Labs, USA
Reid Holmes	University of Washington, USA
Einar Høst	Norsk Regnesentral, Norway
Toshihiro Kamiya	National Institute of Advanced Industrial Science and Technology, Japan
Andrew Ko	University of Washington, USA
Ken Krugler	Krugle, USA
Cristina Lopes	University of California, Irvine, USA
Andrian Marcus	Wayne State University, USA
Kumiyo Nakakoji	University of Tokyo & SRA, Japan
Oscar Nierstrasz	University of Bern, Switzerland
Lori Pollock	University of Delaware, USA
Romain Robbes	University of Lugano, Switzerland
Susan Sim	University of California, Irvine, USA
Janice Singer	NRC, Canada
Suresh Thummalapenta	North Carolina State University, USA
Andreas Zeller	Saarland University, Germany

Workshop Extended Abstract

WS_0824	SUITE 2009: First International Workshop on Search-Driven Development– Users, Infrastructure, Tools and Evaluation	Sushil Bajracharya Adrian Kuhn Yunwen Ye

List of Papers

SUITE_3123	Sourcerer: An Internet-Scale Software Repository	Sushil Bajracharya Joel Ossher Cristina Lopes
SUITE_3113	Characterizing Example Embedment as a Software Activity	Ohad Barzilay Orit Hazzan Amiram Yehudai
SUITE_3117	Search, Stitch, View: Easing Information Integration in an IDE	Thomas Fritz Gail C. Murphy
SUITE_3121	Do developers search for source code examples using multiple facts?	Reid Holmes
SUITE_3119	Software Component Recommendation Based on User Collaborations	Makoto Ichii Yasuhiro Hayase Reishi Yokomori Tetsuo Yamamoto Katsuro Inoue
SUITE_3115	Lowering the Barrier to Reuse through Test-Driven Search	Werner Janjic Dietmar Stoll Philipp Bostan Colin Atkinson
SUITE_3111	Programmable Queries or a New Design of Search Tools	Toshihiro Kamiya
SUITE_3110	Exploring Java Software Vocabulary: A Search and Mining Perspective	Erik Linstead Lindsey Hughes Cristina Lopes Pierre Baldi
SUITE_3118	Improving Software Quality Via Code Searching and Mining	Madhuri Marri Suresh Thummalapenta Tao Xie
SUITE_3122	Hybrid Storage for Enabling Fully-Featured Text Search and Fine-Grained Structural Search over Source Code	Oleksandr Panchenko
SUITE_3114	What to Search For	Steven Reiss
SUITE_3112	On the Evaluation of Recommender Systems with Recorded Interactions	Romain Robbes
SUITE_3116	Internet-Scale Code Search	Susan E. Sim Rosalva E. Gallardo-Valencia
SUITE_3120	Working with Search Results	Jamie Starke Chris Luce Jonathan Sillito

SUITE 2009: First International Workshop on Search-Driven Development – Users, Infrastructure, Tools and Evaluation

Sushil Bajracharya
University of California, Irvine
sbajrach@ics.uci.edu

Adrian Kuhn
University of Bern
akuhn@iam.unibe.ch

Yunwen Ye
Software Research Associates, Inc.
ye@sra.co.jp

Abstract

SUITE is a new workshop series that specifically focuses on exploring the notion of search as a fundamental activity during software development. The goal of the workshop is to bring researchers and practitioners with special interest on search technology for software developers together. Participants will have broad range of expertise in topics ranging from building software tools and infrastructure, Information Retrieval, user studies and Human-computer interaction, benchmarking and evaluation.

The first edition of SUITE is held in conjunction with the 31^{st} International Conference in Software Engineering (May 16-24, 2009. Vancouver, Canada).

1 Motivation

The workshop is motivated by the observation that software developers spend most of their times in searching pertinent information they need to solve their task at hand [8]. Past research has shown that code search is the most frequent activity software developers engage in [12]. They spend most of their time in navigation and search tools in their IDE [11]. More recently there has been some significant efforts both from academia and the industry in building specialized search engines for developers [2, 3, 1, 5, 4, 9, 6, 10, 13, 7]. Most of these leverage the huge amount of source code available in open source repositories. However, these tools are still exploring the tip of the iceberg. We know that source code is not the only artifact that developers need to search and that traditional search engine interfaces have limitations to serve as ideal tools for searching pertinent information for developers. Furthermore, along with the tools we still need a solid understanding of how developers are really using these systems.

2 Objectives

As software development is a process of both information creation and information gathering, software developers are constantly searching for the right information and person to solve their problems at hand. This workshop will focus specifically on exploring the notion of search as a fundamental activity during software development. The goal of the workshop is to bring researchers and practitioners with special interest on search technology for software developers together. Participants will have broad range of expertise in topics ranging from building software tools and infrastructure, information retrieval, user studies and HCI, benchmarking and evaluation.

The workshop will facilitate interested researchers to share their ideas and experience in understanding the search need and behavior of developers, building tools that addresses these various needs, and scientific ways to evaluate these tools.

3 Topics

The workshop addresses the problem of search as it occurs during software development. Search is related to software mining, but differs in its problems and challenges. For example, two of the important topics the workshop focuses on are: a) search-engines for public software repositories on the internet, and b) specialized search-engines for IDEs.

Areas of interests include, but are not limited to:

- Application of natural language processing on source code and related artifacts.
- Approaches, applications, and tools for software search.
- Case studies on setting up and running large software search-engines.
- Empirical studies of search and navigation in IDEs.

- How can industry and researchers collaborate?
- Information retrieval and machine learning techniques to search source code.
- Integration of specialized search engines into IDEs.
- Methods of integrating indexed data from various sources and histories.
- Query languages to search software and repositories.
- Search techniques to assist developers in finding suitable components and code fragments for reuse.
- Techniques for indexing large software repositories (and their history) efficiently.
- Static analysis and parsing of internet-scale code repositories.
- Crawling source code in the internet and code repositories.
- The use of visualizations to support software search.
- Validation of tools and software searching benchmarks (datasets).
- Ranking strategies and heuristics for code search.
- Slicing and generative techniques for code extraction and synthesis.

4 Submissions

This year's submissions to the workshop touches various themes as seen across the topics presented above. They range from tools and infrastructure to user studies and experiments. All, in one way or another, motivated by the goal of enhancing the search experience of developers during software development. The list of accepted papers is available from the workshop's website `http://smallwiki.unibe.ch/suite2009/`. Final versions of the papers appear in the ICSE proceedings.

5 Ogranizers

Sushil Bajracharya[1] is a PhD candidate in the Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California Irvine, USA.

Adrian Kuhn[2] is a PhD candidate at the Software Composition Group, University of Bern, Switzerland.

Yunwen Ye is a manager in the Technology Strategy Division in Software Research Associates, Inc. Japan.

[1] `http://www.ics.uci.edu/~sbajrach`

[2] `http://smallwiki.unibe.ch/adriankuhn`

References

[1] Google code search home page. *http://www.google.com/codesearch*.

[2] Koders web site. *http://www.koders.com*.

[3] Krugle web site. *http://www.krugle.com*.

[4] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, New York, NY, USA, 2006. ACM.

[5] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, New York, NY, USA, 2007. ACM.

[6] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM.

[7] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52, 2008.

[8] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[9] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, New York, NY, USA, 2007. ACM.

[10] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM.

[11] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Softw.*, 23(4):76–83, 2006.

[12] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[13] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2007. ACM.

Sourcerer: An Internet-Scale Software Repository

Sushil Bajracharya Joel Ossher Cristina Lopes
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{sbajrach, jossher, lopes}@ics.uci.edu

Abstract

Vast quantities of open source code are now available online, presenting a great potential resource for software developers. Yet the current generation of open source code search engines fail to take advantage of the rich structural information contained in the code they index. We have developed Sourcerer, an infrastructure for large-scale indexing and analysis of open source code. By taking full advantage of this structural information, Sourcerer provides a foundation upon which state of the art search engines and related tools easily be built. We describe the Sourcerer infrastructure, present the applications that we have built on top of it, and discuss how existing tools could benefit from using Sourcerer.

1. Introduction

The proliferation of open source software has resulted in vast quantities of source code being available online. This code is a great potential resource for software engineers, as it represents a vast store of accumulated development knowledge.

Accessing this knowledge, however, has proven to be a challenge. A number of open source code search engines have sprung up, but all are based primarily around traditional information retrieval techniques [1, 2, 3]. The rich structural information contained in the code is all but ignored. This hampers the advent of next-generation code search technologies, which seek to take into account this structural information. It also makes building software engineering tools on top of these search engines significantly more work than should be necessary. In this position paper we present Sourcerer, an infrastructure for large-scale indexing and analysis of open source code. Sourcerer provides a foundation upon which these state of the art search engines and tools can easily be built.

We describe the Sourcerer infrastructure, specifically the metamodel it uses for storing detailed structural information, how it links references across projects, and the code index. We also present the applications that we have built on top of this infrastructure, which are now available as public web services, and discuss how existing tools could benefit from using Sourcerer.

2. Sourcerer Infrastructure

Sourcerer crawls the internet looking for Java source code from a variety of locations, such as open source repositories, public web sites, and version control systems. This code is then parsed, analyzed and stored in Sourcerer in various forms: (i) *Managed Repository* keeps a versioned copy of the original contents of the project and related artifacts such as libraries; (ii) *Code Database* stores models of the parsed projects, based on the metamodel; and, (iii) *Code Index* stores keywords extracted from the code for efficient retrieval.

Figure 1 shows the general architecture of the Sourcerer infrastructure, specifically in the context of the code it indexes and the services and applications that it supports. More information on an earlier architecture in addition to some repository statistics can be found here [11].

2.1. Relational Metamodel

There were two major considerations in deciding on the exact metamodel for the structural information in Sourcerer. It had to be sufficiently expressive as to allow fine-grained search and structure-based analyses, and it had to be efficient and scalable enough to include all the code we could get our hands on.

In the end, we settled on an adapted version of Chen et al.'s [5] C++ entity-relationship-based metamodel. In particular, we agreed with their decision to focus on what they termed a *top-level declaration* granularity, as it provides a good compromise between the excessive size of finer granularities and the analysis limitations of coarser ones. The metamodel we present here is an evolution of the earlier

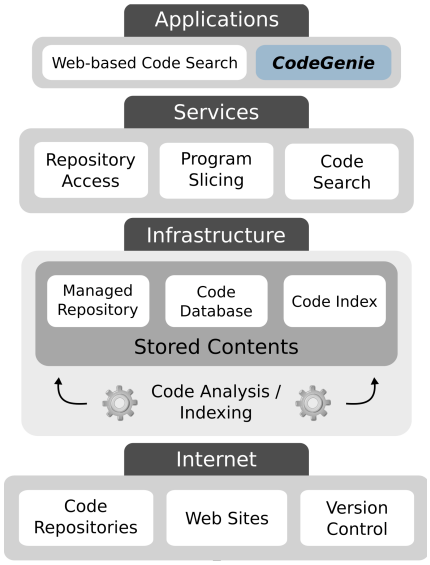


Figure 1. Sourcerer system architecture

Sourcerer metamodel [11]. The revised metamodel adds support for the latest version of Java, among other things.

Following the metamodel, a project model element exists for every project contained in the managed repository, as well as every unique jar file. A project thereby contains either a collection of Java source files or a single jar file. Both types of files are linked to the sets of entities contained within them, and to the relations that have these entities as their source.

Entities The following is a complete list of entity types: `PACKAGE`, `CLASS`, `INTERFACE`, `CONSTRUCTOR`, `METHOD`, `INITIALIZER`, `FIELD`, `ENUM`, `ENUM CONSTANT`, `ANNOTATION`, `ANNOTATION ELEMENT`, `PRIMITIVE`, `ARRAY`, `TYPE VARIABLE`, and `PARAMETRIZED TYPE`. These types all adhere their standard meaning in Java, as defined in the Java Language Specification (JLS) [6]. Each entity is uniquely identified within a project by its fully qualified name (FQN), a slightly altered version of the *binary name* described in the JLS. An entity is further annotated with the Java modifiers that referred to it, the project and file that it came from, and its location in that file.

Relations The majority of the structural information is found in the relations between these entities. All of the relations are binary, linking one entity to another. Two example relations are `INSIDE`, representing physical containment, and `USES`, a catch-all relation for any references. A relation is uniquely identified by its project, type, and the FQNs of its source and target. Any time that the same relation is generated more than once by the feature extractor, such as a method calling another method multiple times in its body, those relations are collapsed into one. Therefore, unlike entities, relations are not linked directly to their location in the source code. Their smallest containing entity is all that is known.

2.2. Cross-Project Dependencies

Every project has some external dependencies, even if only on the standard Java libraries. These dependencies are typically packaged in jar files and included along with the source code. During feature extraction, a large number of relations end up with their targets as entities contained in these jar files. Take Apache's Log4j project, for example. Many other projects use Log4j, and include `log4j.jar` in their repositories. In such a project, let's call it Loggy, one could see relations with target entities contained in `log4j.jar`.

On the one hand, these jar entities are specific to Loggy, as they are located within the copy of `log4j.jar` found in the Loggy repository. On the other hand, these jar entities could be matched to the corresponding entities in Apache's Log4j project (which also happens to be in our repository) as well as to other identical copies of `log4j.jar`. In order to gather cross-project dependency information, all such uses of entities from `log4j.jar` ought to be linked to the Log4j project. However, the original link to the jar file ought to be preserved, in case Loggy's version of `log4j.jar` is different from the version of Log4j in our repository.

In order to achieve these goals, we begin by uniquely identifying all the jar files across projects. Each file is then run through our feature extractor, and the results are placed into the database. Whenever a relation referencing a jar entity is added to the database, it is linked to the entity from that jar. Once the repository is fully populated, we then attempt to match each jar entity to a corresponding entity in a source project using a number of heuristics.

In cases where a necessary jar file might not have been included in a project repository, we try to locate that jar file based on the missing dependency information. Also, we have a number of heuristics to detect cases in which a project reuses a library by copying source code, so that such dependency information is not lost.

2.3. Fine-Grained Code Index

Sourcerer maintains a fine-grained index of the terms extracted from various parts of the code. The searchable index is constructed with fields that closely parallel the various entities in the metamodel. Table 1 presents a subset of the fields available in the Sourcerer index. The full list is described in [4]. To populate these fields, a language specific tokenizer extracts more meaningful terms from the entity

FQNs and parts of the comments. Common practices in naming, such as the CamelCase and the use of special characters (eg; "_", "-") are used to split the names into these terms.

Fields in the Sourcerer index can be categorized into five types: (i) Fields for basic retrieval that store terms coming from various parts of the name of an entity; (ii) Fields for retrieval with signatures that store terms coming from method signatures and also terms that indicate number of arguments a method has; (ii) Fields storing metadata, for example the type of the entity, so that a search could be limited to one or more types of entities; (iii) Fields that pertain to some metric computed on an entity; (iv) Fields that store ids of entities for navigational/browsing queries.

Sourcerer uses Lucene as its underlying index store. Here is a sample query that utilizes different fields:

- *"short_name: (week date) AND entity_type: METHOD AND m_ret_type_sname_contents: String AND m_sig_args_fqn_contents: Date"* meaning, find a method with terms "week" and "date" in its short name, that returns a type with short name "String" and takes in argument with a type with the term "Date" in its name.

Index Field	Description
Fields for basic retrieval	
fqn	Fully qualified name of an entity, untokenized
fqn_contents	Tokenized terms from the FQN of an entity.
comments	The collected text (untokenized) from an entity's comments
Fields for retrieval with signatures	
m_sig_args_sname	method's formal arguments short name in format arg1,arg2,arg3,...,argn
m_sig_ret_type_fqn	FQN of the method's return type
Fields Storing metadata	
entity_type	String representation of entity type. Eg; "CLASS"
Fields for navigation	
fan_in_mcall_local	entity ids of all local callers for a method from the same project

Table 1. Sample search index fields

3. Sourcerer Web Services

All of the artifacts managed and stored in Sourcerer are accessible through a set of web services. Currently three services are open to public. A detailed description of how to use these services is available online [4]. We intend to add a fourth service in the near future to provide more direct access to the code database.

1. *Code Search:* This service implements a query processing facility. Client applications (such as CodeGenie [10]) can send queries as a combination of terms and fields and the service returns a result set with detailed information on the entities that matched the queries. The query language is based on Lucene's implementation and our extended query parser supports different query forms that allow the clients to express more structural information in the queries.

2. *Repository Access:* This service provides access to the Managed Repository in Sourcerer. All the code artifacts, libraries and metadata are accessible using this service. Every entity that is stored in the Sourcerer repository has a unique identifier and thus services provides access to the source of the entity (for example a Java file) given the unique id.

3. *Slicing Service:* This service provides dependency slices of any entities stored in the repository. A dependency slice of an entity is a program which includes that entity as well as all the entities upon which it depends. Requested slices are packaged into zip files, and should immediately be compilable.

   Our algorithm for computing these dependency slices is finer-grained than the forward reachability analysis described by Chen et al. [5], and shares some similarities with the dependency slicing done by Rodrigues et al. [14] for functional languages.

4. Application to Existing Tools

This section presents existing software engineering tools from a few different areas, and describes how they could have benefited from the Sourcerer infrastructure.

Example Recommendation: Holmes et al.'s Strathcona [7] is a tool for using a developer's current structural context to recommend source code examples. Strathcona attempts to match the structural information in the current context against examples from its repository. The information stored in Strathcona's repository is sufficiently similar to that in Sourcerer's that Strathcona could be implemented on top of the Sourcerer infrastructure. This would focus Strathcona's development on the matching heuristics and client integration, while immediately providing access to a very large repository.

XSnippet [15], Prospector [12] and PARSEWeb [16] are all systems designed to provide examples of object instantiation. Although implementation on top of Sourcerer would provide some benefit to all of them, PARSEWeb would be dramatically improved. Currently PARSEWeb uses Google Code Search to find and download likely examples of object instantiation. These snippets are then analyzed to determine

if they contain appropriate invocation sequences. This analysis is complicated by the fact the code snippets are missing most of their external references. PARSEWeb is forced to utilize a variety of heuristic techniques to guess the missing types. Sourcerer is ideally suited for this sort of use, as it can provide snippets where the external references are present, eliminating the errors introduced by the fuzzy analysis.

Information Mining: Both SpotWeb [17] and CodeWeb [13] are tools for detecting API hotspots. If they were to use Sourcerer, hotspots could be detected directly simply by ordering the entities in a jar by the number of incoming relations.

Pragmatic Reuse: Holmes and Walker's approach to reuse [8] shares many similarities with our dependency slicing. While our approach is fully automated, drawing in all necessary dependencies, theirs permits a greater level of customization, allowing developers to exclude dependencies they do not want. In order to achieve this customization, however, a developer must download and import the full project into his workspace. This creates a fair amount of manual overhead, for if there are multiple candidate projects for reuse, the process must be repeated for each one. Furthermore, any unresolved dependencies in the initial project download will remain unresolved in the final result. The combination of their approach with the Sourcerer infrastructure has the potential to eliminate many of these problems. One could construct a reuse plan on a slice returned by our system, further reducing its size, without having to worry about downloading the full project or unrelated or unresolved dependencies.

Test-Driven Code Search: Tools such as CodeGenie [10] and Code Conjurer [9] take a test-driven approach to code search. Both automatically use the context provided by a test case to formulate queries. CodeGenie uses Sourcerer code search as it underlying search engine, and thus benefits from the slicer and repository access web services, as well as the cross-project dependency resolution. Code Conjuerer uses merobase (www.merobase.com), instead, which has similar capabilities to Sourcerer. While Code Conjurer's dependency resolution can pull in required files, Sourcerer's dependency slicing is at a finer granularity, which helps reduce the complexity of reused code. Nevertheless, both these tools demonstrate the benefit of an internet-scale code repository such as Sourcerer.

5. Conclusion

In this paper, we presented the Sourcerer infrastructure for the large-scale indexing and analysis of source code. We briefly outlined highlights of its functionality, and described the applications that we have built on top of it. Lastly, we discussed how existing tools could benefit from using Sourcerer.

References

[1] Google code search. http://www.google.com/codesearch.
[2] Koders. http://www.koders.com.
[3] Krugle. http://www.krugle.org.
[4] Sourcerer services. http://sourcerer.ics.uci.edu/services/.
[5] Y.-F. Chen, E. R. Gansner, and E. Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Softw. Eng.*, 24(9):682–694, 1998.
[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
[7] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM.
[8] R. Holmes and R. J. Walker. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *ICSR '08: Proceedings of the 10th international conference on Software Reuse*, pages 330–342, Berlin, Heidelberg, 2008. Springer-Verlag.
[9] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52, 2008.
[10] O. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, 2009.
[11] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*.
[12] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM.
[13] A. Michail. Code web: data mining library reuse patterns. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 827–828, Washington, DC, USA, 2001. IEEE Computer Society.
[14] N. Rodrigues and L. S. Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.
[15] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM.
[16] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2007. ACM.
[17] S. Thummalapenta and T. Xie. Spotweb: detecting framework hotspots via mining open source repositories on the web. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 109–112, New York, NY, USA, 2008. ACM.

Characterizing Example Embedding as a Software Activity

Ohad Barzilay
Tel Aviv University
ohadbr@tau.ac.il

Orit Hazzan
Technion - IIT
oritha@techunix.technion.ac.il

Amiram Yehudai
Tel Aviv University
amiramy@tau.ac.il

Abstract

We use an empirical qualitative software engineering research to characterize Example Embedding (EE) as a software activity - a collection of fine grained techniques which together assemble an abstract key notion in software development. This perspective lays the foundations for building an activity catalogue, forming new software practices, affecting the development process and motivating the development of new software tools.

1. Introduction

We wish to identify and characterize a variety of usages of examples in software construction as a *software activity*. Motivation for this research direction is partially derived from the already well-known software activity of *refactoring*. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [1]. Although refactoring code has been performed informally for years, William Opdyke's 1993 Ph.D. dissertation [2] is the first known resource that specifically examines refactoring.

The mere identification of refactoring as an activity, as well as its naming and defining, promoted the following important processes: First, building a catalogue of different examples of refactorings, describing how to apply them properly and discussing their subtleties. Second, it enabled the development of software tools automatically applying various refactorings. Third, it influenced the coding practices, the way in which programmers write code, using, for example, test-driven development techniques. And fourth, it affected the software lifecycle by allowing the design phase to be incorporated into the coding phase and by legitimizing time allocation for activities which improve the code without adding functionality. These advancements would not have been possible had the refactoring activity not been extracted from the various activities that constitute the coding phase.

We wish to identify another such activity, which is currently not well appreciated as a standalone technique, and thus establish the foundation for further advancements. We argue that Example Embedding (EE), the act of embedding a code segment from an example into a software system being developed, is such an activity. Although some of the aspects of EE have already been addressed in the literature from the perspectives of tool writers and machine learning, e.g. [3], we introduce a broader perspective by which one could derive additional implications and applications of the example embedding software activity.

2. Related work

The study of using examples in software engineering appears in the literature in several contexts. We highlight the similarities and differences between our approach and the approach taken by other representative researchers.

In [3] Edwards takes a tool centric approach. Example Centric Programming provides an IDE that illuminates code with examples. In order to implement a new functionality, say a method, the programmer writes some well-chosen examples that call this method, and an automated tool, called EG, writes the code of the required method interactively with the programmer. Edwards lists some interesting implications of this approach, such as example centric debugging and testing, teaching with examples and example driven development. Although our research shares some of these implications, the EE software activity described here relies on already existing examples rather than trade code writing with example writing.

In [4], Hummel *et al* use a slightly different approach, They present a tool that automatically finds and presents suitable reusable software components to developers, where the programmer specifies the functionality needed using an interface-like syntax. Our approach, on the other hand, is not tool driven, but rather a product of an empirical study, formulated through a field research. We would seek tool support for an already existing software activity rather than engineering a software activity to conform to existing useful tool.

Another context in which examples are studied is learning, such as machine learning and program comprehension. This is out of the scope of our work.

3. Methodology

The software engineering research field described in the paper employs a qualitative research methodology to build a field-grounded theory [5]. We investigate industrial software development activities and patterns by observing professional software engineers at work in major software companies, aiming to identify development activities that are currently assimilated in the coding phase. Data gathering tools include observations, interviews, reflective practitioner techniques, and questionnaires.

The research is built bottom-up: we start from fine-grained activities observed in the field and analyze the data using standard qualitative techniques [5], looking for recurring activities and hidden patterns. We then refine our research questions and repeat the process (observation-analysis-refinement), aiming to build a field-grounded theory, identifying and characterizing an already existing software activity that have not been investigated yet. The first phase of this 3-phases research has already been completed and involved the investigation of software development at the development sites of two major, worldwide software companies.

4. Software activity

We define a *software activity* as a collection of fine grained techniques, which together assemble an abstract key notion in software development. We argue that some of the already acknowledged software activities, such as *coding* or *debugging*, are composed of other finer software activities. For example, *refactoring* is one of the software activities that constitute the *coding* software activity.

Distinguishing a new software activity from an already acknowledged activity is an important process that affects software engineering in various dimensions as follows.

4.1. Abstraction

The mere naming and definition of a new software activity increase the level of abstraction in the software engineering discourse. Due to the new name, some micro activities could be interpreted in a wider context of a more abstract notion. This recognition may affect the development of tools, catalogues and practices as described in the next sub-sections.

Moreover, the definition raises the awareness level of the programmers to cognitive aspects inherent in the particular software activity, by highlighting the fact that some of the tasks involved in software construction differ from others.

4.2. Recipe catalogue

We use the term *recipe catalogue* to describe a detailed and comprehensive description of all known instances of some software activity. The GoF book [6] is a good example of a catalogue (for design patterns). As with cook books, where different dishes require different attention, so do software activities.

4.3. Practices

We define a *software practice* as a set of behaviors, applied systematically following some mental model [7] to perform a software related task. In contrast with a software activity, which is declarative in nature, a software practice is a meaningful imperative unit at the abstraction level of the development process. A software activity by itself is meaningless unless applied in a systematic, disciplined methodological way. For example, *test driven development* [8] is a programming practice enabled by the refactoring and testing activities.

4.4. Development process

An effective programming practice that encompasses multiple activities challenges software lifecycle models such as the waterfall model and the spiral model, which address only one activity in each of their phases, such as: design, coding and testing. Agile methods, on the other hand, [9], break the phase paradigm, and compensate the lack of a designated design and testing phases by using a test driven development software practice. This practice incorporates design and testing together into the coding activity.

4.5. Tools

Software tools are useful for activity implementation in at least two ways. First, after one characterizes a key activity, it is easier to identify repetitive tasks. In many cases, those tasks could be automated or at least supported by an automatic software tool that would perform them easily, and in some cases could ensure and enforce their correctness consistently. A second use of a software tool is in providing a framework that captures the sub elements

of this software activity and assists in streamlining them. Working with such a framework guides the developer along the major steps.

5. Example embedding

During the first observation phase of the research, we tracked nine experienced developers in two Israeli sites of major worldwide software companies. The analysis of the observation reports discovered that many developers (among other activities) use examples in ways that differ from each other, for example in the granularity and type of the example, in the way the example is found and where, in how the example is chosen and in how it is embedded in the code.

Aiming to generalize and characterize this software activity, we define *example embedding* (EE) to be the use of some already written code in the process of writing some new code. In what follows, we present the various dimensions of software engineering mentioned above with respect to EE and introduce preliminary classification criteria that could serve as a starting point for a future catalogue, and motivate the development of new software tools and programming practices.

5.1. Abstraction

The definition and characterization of EE raises the programmers' level of awareness to *code reuse*; it is not only a programming language feature but should also be assimilated in a common practice. We argue that reuse should not be limited to the use of code as is, but rather, should be applied more freely, as long as the resulting unit is tested thoroughly. The difference between our approach to reuse and the conventional black box reuse is analogous to the difference between refactoring and thorough design. Specifically, as refactoring liberated us from the need to go through an elaborate design phase, EE aims at liberating us from the belief that we need to reuse the code as is. In several situations even code duplication seems to be a reasonable or even beneficial design option [10].

5.2 Recipe catalogue

As this research is still in progress, we present only preliminary results with respect to the categorization task. Following are several possible categories of EE; each of them highlights its variability:

- Example for what. Developers use examples in order to perform various tasks: adding functionality, fixing a syntax error, applying a design pattern or bootstrapping with a new environment ("hello world"). Examples are used for programming languages, scripting languages, libraries and APIs and configurations.
- Example size. Examples are different in size, scope and complexity: from several characters demonstrating a language operator, through function calls, and complex operations requiring a sequence of several method invocations involving several types.
- Examples source. Examples are taken from the organization code base, documentation, example set which is provided by the company, web tutorials, blogs, emails and more.
- Searching for example. Developers search for examples using Google search, code search, code browsing, asking people, and documentation search; in several occasions, they know upfront where to look.
- Using the example. Examples are used in various ways: copy and paste, retyping the example, refactoring of the example code and then call it.

In some cases one could think of additional variations that have not been observed yet in the field. For example, we mention that examples are taken from the organization code base but we do not mention using *open source code* because at this phase our research is empirically driven and during our observations so far we have not yet found evidence for that. At later phases of our research we will focus on additional EE techniques and look for some missing variations.

5.3. Practices

Example driven development is primarily a state of mind, a mental model [7]. Accordingly, we argue that many programming tasks could be reduced to an example driven development cycle: find–alter–embed. In this spirit, one of the important roles of the categorization presented above is to raise the level of awareness to the wide spectrum of scenarios in which examples exist.

5.4. Development process

A software organization that appreciates the contribution of EE to the development process could take actions to improve its effectiveness.

One key component is *writing examples*. We argue that each programmer should provide examples for using the code he or she writes. This should be anchored in the development process the same way that testing the code is. The analogy with test coverage

is subtle. In *test coverage* every unit in the system is tested for some behavior. *Example coverage*, on the other hand, aspires that every snippet of the system code could be easily used to perform meaningful standalone tasks not necessarily in its original context.

Making example writing a habit, would pave the way for public and general purpose example repositories, integrated with documentation ones, with similar structure and wide community support.

5.5. Tools

We identify three major building blocks in the EE activity: find, alter, and embed. Tool support should address each of the activities by itself as well as their integration.

IDE-browser integration. When a programmer conducts a web search using a general purpose web search engine, he or she needs to switch from the IDE to another application (the browser). The search might yield several results, which s/he opens in different tabs, and for examination purposes should copy and paste each of them back to the IDE. We propose an integrated solution in which the web search would be possible from within the IDE. The various possible solutions could be presented side by side, along with the code in which they are to be embedded. This would preserve the programming context along the process of finding, evaluating and embedding the example.

Extract example IDE support. Modern IDEs support a large number of automatic refactorings; we wish to have something similar for EE. When a programmer browses production code and spots a snippet that fits his or her needs, the IDE should offer assistance in this task as described in [11]. However, we suggest that copy and paste is not the preferred consumption of an example (to avoid code duplication) but rather, we propose to enable to extract the example snippet into method and to call it.

IDE-example repositories integration. Such integration was already suggested by [12] and [13]. Recent advancements in IDE technology and useful heuristics such as [14] could overcome the challenges of searching and browsing these repositories.

6. Summary and future work

In this paper we described an ongoing empirical software engineering research aimed at characterizing EE as a software activity. From a methodological perspective, we showed the benefits of conducting an empirical SE research on the granularity and abstraction level of software activities and highlighted its affect and implications. We also presented preliminary results of our prototypical research findings regarding EE and its contribution to programming practices, the development process and the creation of software tools.

A lot of work is still left to be done identifying additional software activities and characterizing the already existing ones. This is a long, iterative and interactive process involving SE empirical researchers, computer scientists, software engineers and other experts seeking to know more about this multi-faceted domain of software engineering.

7. References

[1] M. Fowler, "Refactoring Home Page," http://www.refactoring.com.

[2] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1993.

[3] J. Edwards, "Example centric programming," *SIGPLAN Not.*, vol. 39, no. 12, pp. 84–91, 2004.

[4] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[5] B. G. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago: Aldine Publishing Company, 1967.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[7] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[8] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[9] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[10] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, 2006, pp. 19–28.

[11] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*. IEEE Computer Society, 2004, pp. 83–92.

[12] L. R. Neal, "A system for example-based programming," *SIGCHI Bull.*, vol. 20, no. SI, pp. 63–68.

[13] F. Gerhard, H. Scott, and R. David, "Cognitive tools for locating and comprehending software objects for reuse," in *ICSE '91: Proceedings of the 13th international conference on Software engineering*. IEEE Computer Society, 1991, pp. 318–328.

[14] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 117–125.

Search, Stitch, View: Easing Information Integration in an IDE

Thomas Fritz and Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
{fritz,murphy}@cs.ubc.ca

Abstract

When building a software system, software developers each contribute a flow of information that together forms the system. As they work, developers continuously consult various fragments of information to answer their questions about the system. In today's programming environments, information is kept in disparate silos, such as program code, bugs and change sets. However, to answer the variety of questions a developer faces, the interleaving of information from multiple sources is typically needed. We have implemented a prototype that allows for the composition of information fragments from different silos. We also interviewed three experienced developers to find out about cases when they need to interleave information.

1 Motivation

When building a software system, software developers each contribute a flow of information that together forms the system. This information comes from many different domains; for instance, source statements, bug descriptions and meta-information about the creator as well as the creation time of a change to the system. As a developer is working on the system he has to answer a variety of questions, such as "Why and when was this file introduced and how did it evolve over time?", "Who is working on what in my team?" and "Which changes are affecting me [my code]?". The questions originate from interviews with practicing industrial developers and are equivalent to some found by Ko et al. [3]. These kind of questions span across multiple domains of information, such as changes, bugs and source code. Current integrated development environments (IDEs) that help support developers in their work, mostly focus on providing information about one domain at a time. This focus makes it hard for developers to answer these kinds of questions. Through talking to experienced software developers, we have learned about three particular scenarios in which these problems occur.

Evolution of a file. One developer told us about his development team's process for tracking the evolution of a file. This team uses a tool external to the IDE that requires each developer to enter an identifier of the bug a commit fixes in the commit comment. The developers in the team can then use this tool to find to find the changes made to the file, look at the actual changes and use the identifier from the comment to find the corresponding bug to understand why the change was implemented. While this approach helps solve a development problem, the solution is outside of the IDE so the developers lose the benefits of integration with other tools to be able to achieve integration of source code and bug information.

Who is working on what in my team. New team members and team managers need to find out who is working on what in their team. Jazz, a team collaboration platform on top of the Eclipse integrated development environment, should be well-suited to answer this kind of question. It brings together several different domains of information, such as source code, bugs, and teams. However, as many views in Jazz focus on a single domain of information, a developer typically has to manually and cognitively map the information from different views to answer such questions as who is working on what.

Which changes affect me. To ensure that developers are made aware of changes that might affect them, one team we spoke with implemented a strategy that generates an email to each team member for each change committed to the team's repository and sends it out to the whole team. A downside of this approach is the large number of emails generated that must be dealt with by each team member. Another developer stated that his team has a "general communication problem" as the changes do not "trickle down"

to the people. The problem in this case is that developers might miss important information because its not being distributed.

The approaches taken by the developers we talked to for each of these scenarios, either result in developers cognitively mapping together information from different tools and views, result in developers having to wade through a lot of information to find just those subsets that are relevant, or result in developers missing important information. We want to reduce these problems by providing a common model in the developer's IDE that allows for the integration and presentation of information fragments from multiple domains to answer the questions at hand.

A more detailed description of the underlying approach as well as related work is presented in [1]. In this position paper, we focus on the initial prototype implementation as well as initial feedback by three experienced developers.

2 Working with Information Fragments

We have implemented a prototype that allows for the creation, composition and presentation of information fragments on top of Jazz.

Information Fragments. Underlying the prototype is the idea that a developer working on a system never thinks about all of the information forming the system at a time. He only takes into account a small part of the complete information. These subsets of information are what we refer to as *information fragments*. In our prototype, an information fragment is represented by a graph. The nodes represent uniquely identifiable items of information; each is attributed to a domain. Currently, we are supporting Java nodes (classes, methods and fields), work items, change sets and team elements (teams and team members). Edges represent relationships between nodes that can be explicit, such as method calls, or implicit, such as a relationship between a change set and a work item that can be inferred from nodes. Both, nodes and edges contain certain properties that describe them. To create an information fragment, we enhanced existing views in Jazz to create fragments by simply selecting elements of interest and clicking the button highlighted in Figure 1(a). We call these fragments *base fragments* as they only contain nodes of one domain. Once the fragment is created, it will be added to the list of base fragments as shown in Figure 1(b).

Composition. To answer questions like "Who is working on what in my team?", developers have to combine information from different domains. To support the variety of ways in which information fragments may be integrated, our approach provides composition operators. A composition operator combines two fragments by creating new edges between nodes from different fragments based on node properties. Currently, our prototype provides only one operator that matches identifiers. In the upper part of the Fragment Explorer view shown in Figure 1(c) the *composed fragment* contains a work item node and change set nodes. To integrate the afore added Java fragment with the composed fragment, the developer selects the fragment in the lower part of the view and clicks the composition button highlighted in Figure 1(c). The composition operator will go through all properties of the work item nodes as well as change set nodes and find identifiers that refer to Java nodes. If there are any it will create edges between the nodes. Afterwards it will look through the properties of Java nodes and check wether there are references to the work item or change set nodes. In our example, change sets know about the Java elements they are affecting and therefore, the resulting view shown in Figure 1(d) has a three level hierarchy, with work items on top, the corresponding change sets below and the Java nodes that are affected by those change sets and that area also in the Java fragment selected beforehand on the third level.

Presentation. As each developer has several questions and his own way of answering the questions and interpreting the result, our prototype allows for the view to be configurable. In the upper right corner of the view the order of the hierarchy is shown in a bar. The developer can change the order of the hierarchy by using drag and drop and changing the order of the icons in the bar. For instance, changing the order so that the Java elements are the top nodes as shown in Figure 1(e) allows the developer to answer the question "Which changes are affecting me?". For this question, he just has to choose the Java fragment to include all the elements he is interested in, as well as the interesting change set and work item fragment. By doing so, he can navigate from the Java element to the change set and then even to the work item in one single view without having to map anything manually or cognitively. To answer "Who is working on what in my team?", the developer can add a team fragment with the developers from his team and then change the order so that he sees the work segmented by people and can even go down all the way to the actual classes people were changing as shown in Figure 1(f).

3 Discussion

It is often said that "information is power". Most developers today are deluged with information. At the touch of a few buttons, thousands of lines of source code, the interrelationships between the code and documents about how to use the code can be accessed. All too often, this information does not align with simple questions that the developers need to answer, such as "What is my team working on this week?".

As we have presented in this paper, we believe a possible answer lies in making it easy for a developer to search for

particular kinds of information of interest and then to easily integrate the results of those searches. A different approach would be to pre-code more views into the integrated development environment to support the direct answer of some of a developer's questions. This route seems doomed for two reasons. First, we can't possible pre-code all information integrations of interest. Second, even if the views exist in the environment, it is becoming impossible for developers to find the view of interest when it is needed.

An interesting question to consider is how much of an existing development environment's views might be replaced with a query/integration mechanism such as we suggest. Interestingly, it seems that very few could currently be replaced. Instead, developers would use our approach for answering questions that are currently not being handled in one view but that require work to put the answer together. For instance, one developer, while telling us about the process of answering "who is working on what", pointed out that the view he has to use to answer the question presents only "static" information. Therefore, he has to follow links to several other views to accumulate the required information. Having the information all in one view would save him a lot of time and he was even thinking about implementing a new view himself.

With other query mechanisms (e.g., relational views [4] or JQuery [2]), a developer might be able to form a query to answer some questions of interest, but forming the appropriate query can be almost intractable, requiring a deep knowledge of the query language. We believe that many of the questions we are targeting come in sufficiently different flavors at sufficiently different times that developers are not willing to put a significant time investment into determining the correct formation of a complicated query because the query may need to be different tomorrow. Instead, we have been investigating offering general composition operators between different types of information. This raises two interesting questions: "Will the developer be able to understand what the composition does?", and "Will we be able to cover a variety of a developer's questions with a limited set of operators?". As we develop our approach we need to answer these questions. So far, we have determined that one composition operator together with the configurable presentation accounts for the three scenarios we introduced at the start of the paper; recall, that the original approaches taken by developers required different views of the IDE, external tools and manual mapping. We believe that with a small set of operators we will be able to cover a variety of questions, but future work on operators as well as on scenarios is needed.

4 Summary

Developers have a variety of questions over multiple domains of information. Our approach is intended to support the interleaving of information from multiple domains and thereby alleviating the need to manually and cognitively map the information together. We have presented an initial prototype that allows the composition of information fragments and provides a configurable view to tailor the presented information to the developer's needs. We believe that the combination of the model and the presentation allows the developer to interleave the information in a way that allows him to answer his questions.

5 Acknowledgments

This work was supported by IBM and the IBM Ottawa Center for Advanced Studies. We thank all developers for their time and feedback.

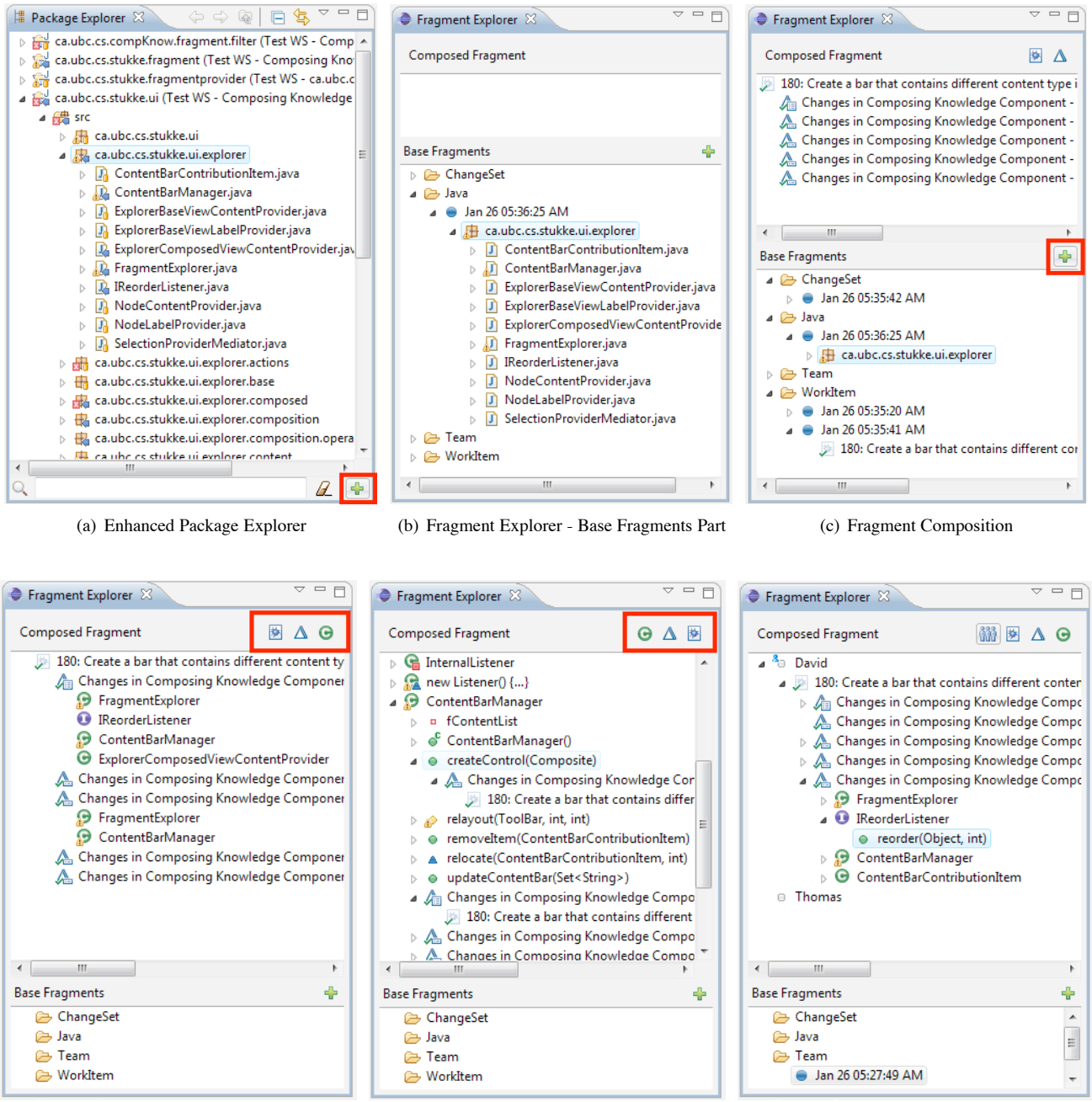
References

[1] T. Fritz. Composing knowledge fragments: a next generation ide. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 999–1002, New York, NY, USA, 2008. ACM.

[2] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *AOSD'03: Proc. of the 2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.

[3] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 344–353, 2007.

[4] M. A. Linton. Implementing relational views of programs. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, New York, NY, USA, 1984. ACM.



(a) Enhanced Package Explorer (b) Fragment Explorer - Base Fragments Part (c) Fragment Composition

(d) Fragment Presentation (e) Which changes are affecting me. (f) Who is working on what.

Figure 1. Fragment Creation, Composition & Presentation.

Do developers search for source code examples using multiple facts?

Reid Holmes
Department of Computer Science & Engineering
University of Washington
Seattle, WA, USA
rtholmes@cs.washington.edu

Abstract

In this paper we examine the search behaviours of developers using the Strathcona source code example recommendation system over the period of three years. In particular, we investigate the number of query facts software engineers included in their queries as they searched for source code examples. We found that in practice developers predominantly searched with multiple search facts and tended to constrain their queries by iteratively adding more facts as needed. Our experience with this data suggest that example search tools should both support searching with multiple facts as well and facilitate the construction of multi-fact queries.

1. Introduction

Several research tools have been created to help developers locate relevant source code examples [7, 4, 1, 3, 6, 5]. One dimension in which these systems vary (see Section 2 for additional detail) is whether they allow a single search fact (such as `getStatusLine()`) or multiple search facts. Although the precision of searches can be improved by providing multiple search facts, the question of whether developers know enough information to create multiple search facts, and whether they are willing to enter them, remains open. By analyzing log files from hundreds of query sessions by nearly one hundred users of Strathcona, a system that allows developers to query for source code examples using multiple search facts, we present preliminary evidence that developers have the knowledge to formulate queries with several search facts in their search for source code examples and do so in practice.

Strathcona is a client–server example recommendation system that enables developers to quickly select a block of code from which a query is automatically generated. The server then returns source code examples that best match the code the developer has selected [1, 2]. We are able to analyze Strathcona usage patterns as the server component saves to disk every query made by developers as well as their corresponding responses.

The primary contribution of this paper is evidence that developers searching for source code examples usually provide multiple search facts in practice. We have observed that 92% of queries contain two or more facts while 36% of queries contain five or more facts. Investigating individual query sessions we found that developers queried on average 2.5 times per query session and often augmented their previous queries with new facts learned from prior results.

Background details on related search approaches is given in Section 2. Section 3 provides a brief overview of the Strathcona tool. The data we analyzed and some quantitative results are presented in Section 4. The paper ends with some suggestions for future code search tools (Section 5) and conclusion (Section 6).

2. Related Work

Several research tools have been developed that can help developers locate relevant source code examples. CodeBroker is an adaptive system that automatically queries an example repository using the comment and method signature of the method the developer's cursor is currently in [7]. Prospector locates examples given a start and end types; the tool then computes possible paths that would enable a developer to get a reference to the end type given their starting type by statically mining example source code [3]; PARSEWeb [5] uses the same input and locates examples using existing code search engines. CodeWeb [4] and MAPO [6] take a simple input and locate examples using generalized association rules.

Each of these systems constrains the number of facts that can be queried on by the developer; typically at most two query facts can be specified although often one of these is reserved. CodeBroker uses one fact for the method signature and the other for the method comment; these cannot be changed by the developer except by moving the cursor to another method. Prospector and PARSEWeb specify that one fact is related to the origin of the query and the other as

the destination. Both CodeWeb and MAPO generate relevant examples from a single query fact.

In contrast to these approaches, Strathcona allows the developer to select any contiguous block of code; all of the statically derivable facts that can be extracted from this block are automatically collected sent to the server in the query. The developer can adapt the query by modifying their selection, but cannot modify the the query otherwise.

3. Strathcona

The Strathcona example recommendation system is an Eclipse plug-in that helps developers search for source code examples. Strathcona is unique in its mechanism for automatically constructing queries for the developer based on their development context. The extracted facts are sent to a remote server that contains a repository of source code; using a series of heuristics [2] the server identifies examples that best match the developer's query.

Strathcona returns at most 10 matches, regardless of the number of examples that are located. The developer can view an abstract representation of each example using a UML-like view, requesting to see the source code only if the example seems relevant to their task. As Strathcona queries are constructed automatically, we envisioned that developers using the tool would query on many structural facts. While we have shown that the heuristics used by the server to match the examples are most effective when two or more facts are included in the query [2], due to limited information at the time we were unable to confirm that this is how developers would use the tool; this paper demonstrates that our assumption of large queries was valid.

4. Quantitative findings

By analyzing all of the saved interactions between the client and the Strathcona server, we were able to gain insight into how the developer used Strathcona during their query session. We analyzed three main types of data recorded by the Strathcona server.

Context queries. Context queries documents were sent from the client to the server whenever a developer selected some fragment of code and queried Strathcona. These documents contain a list of all of the structural facts comprising the query. These facts identify statically derivable method calls, field references, inheritance relationships, and type usages within the block of code the developer has selected.

Returned examples. Strathcona answers each context query with a set of structurally-relevant examples. Each example includes the same structural facts present in the context query so the Strathcona client can build a rationale explaining why the example is relevant for the developer's query and to build the UML representation of the example.

Source requests. If the developer deems an example interesting, they can request its source code; this document simply provides an identifier for the example the developer wishes to see the source code for.

The server records a timestamp for each of these documents, as well as a unique identifier for each host making the query. Unfortunately, we cannot tell from the server logs if the developer found an example useful or not; the only indicators we can use are whether the developer asked for the source code for an example. In this case, we infer that the developer felt the example could be relevant given its UML representation and assume the example was helpful in some way. We facted sessions as *successful* if they ended with a developer making a query and looking at the source code of at least one example (in contrast to ending with a query itself).

4.1. Session overview

Over the thirty-five month period of our Strathcona logs, 239 search sessions were initiated by 94 software developers (from at least 5 countries) encompassing 783 queries.[1] Figure 1 provides an overview of the number of context query and source requests made in each session. Developers averaged 2.4 context queries per session, although the median was only 1. They also requested 4.3 source examples on average, with a median of 2. 49% of sessions involved more than one context query, while 54% of sessions involved multiple source code requests. Figure 2 provides a breakdown of the 3652 query facts provided by developers.

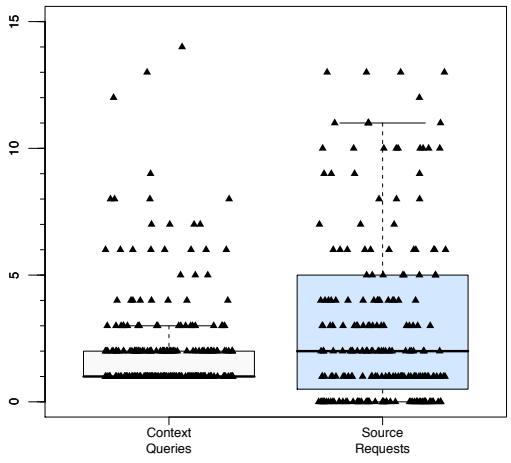
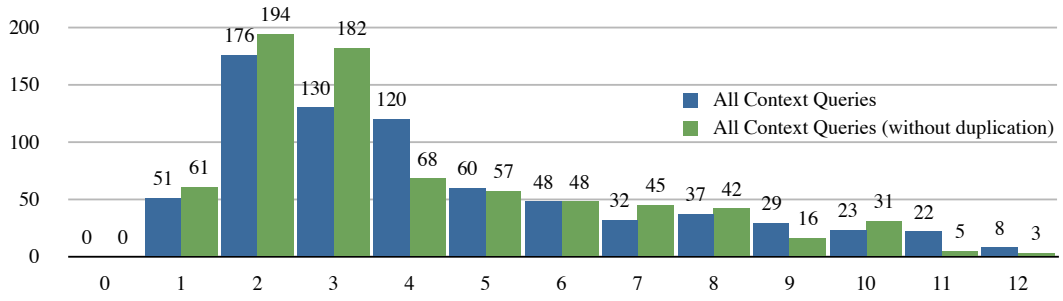
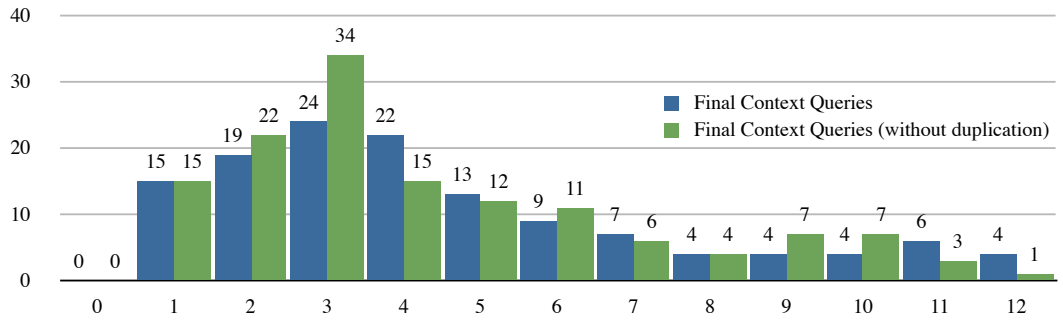


Figure 1. Number of context queries and source requests per Strathcona session.

[1] Before analyzing any of the data, we removed all of the sessions associated with our own usage of the Strathcona tool.



(a) Number of facts provided considering all Strathcona queries.



(b) Number of facts provided considering only the final query in a successful Strathcona session.

Figure 3. Each graph depicts number of structural facts included in a query (x axis) by the frequency queries of each size occurred (y axis). The top set of graphs consider all queries while the bottom pair consider only the final query of each session.

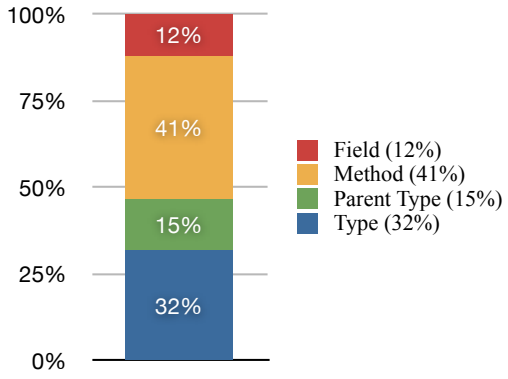


Figure 2. Proportion of query facts.

4.2. Queried facts

For this analysis, we combined all of the different kinds of search facts and treated them equivalently. Figure 3 provides a graphical representation of the queries; the x-axis represents the number of search facts while the y-axis represents the number of queries for each quantity of facts.

Strathcona considers a query fact representing a reference to `Status.OK` as two facts, the reference to the `Status.OK` field and a use of the `Status` class. To account for this, we provide both the total count of the facts as Strathcona interprets them as well as a version that does not duplicate any counts; we include both as different search approaches can chose to use one representation or the other. Figure 3(a) shows the number of facts for all of the context queries while Figure 3(b) shows the number of facts for only the final query of successful sessions.

In Figure 3(a) we can see that that while the median number of facts was two, developers provided three or more structural facts for 67% of their queries. For their final query (Figure 3(b)) the median number of facts has increased to three, with developers providing three or more structural facts for 74% of their queries. This clearly demonstrates that developers using Strathcona are formulating queries with multiple search facts and that they are adding facts to these queries as they progress through their search session.

While examining several sessions qualitatively, we found that while iterating on their query sessions developers were adding new facts to subsequent queries based on information present in example source code they viewed that was returned during prior queries.

5. Discussion

The internal validity of this study is hampered by the fact that Strathcona makes including additional search facts in a query trivial. While this is true, the interesting finding in

this paper is that the developers knew the facts to include in the first place. The external validity of our findings is limited from our lack of knowledge about the 94 developers who used Strathcona, if they were actually successful in finding the information they were looking for, and the obvious limitation of only having 239 search sessions to draw data from.

A key assumption of the Strathcona system was that developers would search for source code examples using multiple search facts; the more facts included in a query, the more effective Strathcona's heuristics tended to be [2]. This paper demonstrates that the assumption upon which Strathcona was created was valid and suggests that evaluations comparing Strathcona to other example recommendation systems should conduct their comparisons using several search facts in order to achieve a fair comparison of relative effectiveness.

Our analysis of the Strathcona usage data have given us several insights into how developers search for source code examples; these observations should be considered by researchers and practitioners creating source code search tools and services.

Developers search with multiple facts. Developers are able to elucidate multiple search facts when searching for context-relevant source code examples. This suggests that code search approaches should support and encourage searching using multiple terms; this can both help the developer to fully express their current knowledge and to constrain the result space to identify the most relevant examples possible.

Query sessions are iterative. Developers modify their queries over the course of a query session to specialize them as they identify new facts they deem relevant to their investigation; search tools should encourage iterative query refinement by including facilities that encourage developers to modify their queries and view their results in a lightweight manner. Tool designs that minimize the effort required to reformulate and specialize queries and reduce the effort required for the developer to glean useful facts from returned examples can help support iterative investigation.

Queries are composed of heterogeneous facts. While method calls were the most common kind of query facts, other types of facts were often included in queries. Facts relating to specific types made up 47% of queries (15% of these type facts related to parent classes and interfaces). Code search systems should enable developers to supply any kind of fact they are able to discern rather than forcing developers to only supply a single constrained kind of fact.

6. Conclusion

By analyzing 35 months worth queries sent to the Strathcona example recommendation system, we have found that developers predominantly queried Strathcona for source code examples using three or more search facts. We also found that as developers iterate on their searches, they tend to constrain their queries by adding more facts, as opposed to widening them by removing facts. These findings demonstrate that developers query for source code examples using multiple search facts in practice; this suggests that example recommendation tools should both allow developers to include multiple facts and make it easy for them to do so, enabling them to fully express the knowledge they have about the examples they are looking for.

Acknowledgements

I would like to thank David Notkin and Rylan Cottrell for their insight and assistance with this paper as well as reviewers for their comments and suggestions. This work has been funded in part through a NSERC Postdoctoral Fellowship.

References

[1] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[2] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[3] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005.

[4] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–176, 2000.

[5] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 204–213, 2007.

[6] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 54–57, 2006.

[7] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. *SIGSOFT Software Engineering Notes*, 25(6):60–68, 2000.

Software Component Recommendation Using Collaborative Filtering

Makoto Ichii[†] Yasuhiro Hayase[†] Reishi Yokomori[‡] Tetsuo Yamamoto[*] Katsuro Inoue[†]

[†]Osaka University
{m-itii,y-hayase,inoue}@ist.osaka-u.ac.jp

[‡]Nanzan University
yokomori@it.nanzan-u.ac.jp

[*]Ritsumeikan University
tetsuo@cs.ritsumei.ac.jp

Abstract

Software component retrieval systems are widely used to retrieve reusable software components. This paper proposes recommendation system integrated into software component retrieval system based on collaborative filtering. Our system uses browsing history to recommend relevant components to users. We also conducted a case study using programming tasks and found that our system enables users to efficiently retrieve reusable components.

1 Introduction

A software component retrieval system (retrieval system), or a source code search engine, is widely accepted as a tool for finding reusable software components or helpful code examples. For example, Google code search [1] and Krugle[2] are publicly available through WWW. There are academically developed system such as SPARS-J [6] and Sourcerer [1]. Hummel et al., reports the growth of the database of publicly-available code search engines [5], on which needs for easily-accessible component are reflected.

However, developers, especially who are not familiar with search engines, often have trouble to retrieve reusable components with search engines. One reason for this is constructing "good" queries requires experience to the search engines. For finding components suitable for his/her requirement from large component repository, refinement of search queries based on try-and-error is necessary. In other case, a component found by a retrieval system requires further components that are directly or indirectly related to the component. In addition, code examples are required to reuse components.

In this paper, we propose software component recommendation approach based on collaborative filtering (CF) technique. CF is a technique to recommend items to a target user by feeding back reputation of other users who have similar preferences/interests with the target user. Various e-commerce systems use CF to recommend items to customers.

We have developed component recommendation system integrated into SPARS-J. Our system automatically recommends potentially-beneficial components for users based on their browsing history. Our approach assumes that the developers who have similar browsing history require similar components. Our approach has potential to help users finding a set of reusable components including dependency and example code as they are guided by experienced antecessors without labored try-and-error.

We have conducted a case study in order to evaluate how our system can help developers; eight participants have tried coding tasks with/without our system.

This paper is constructed as follows: Section 2 describes about the recommendation systems as the background. We explain our recommendation method in Section 3 and its implementation in Section 4. Section 5 describes about the case study. We conclude this paper in Section 6.

2 Recommendation System

Various works have tried to support developers who have trouble to understanding a software system based on the *content-based* approach, i.e., analysis of the software system itself. Find-Concept [8] helps developers to finding concerns, or implementation which they are interested in, based on a hybrid technique of structural program analysis and natural language processing.

Meanwhile, *collaborative filtering* (CF) is a technique to recommend items to a user that matches to preferences of the user from vast amount of items by sharing reputations to items among users. [4]. GroupLens [7] is an automated collaborative filtering system that recommends articles of Usenet. A user of GroupLens *votes ratings* of articles they read in five-point scale. Then GroupLens recommends articles that are highly-rated by users whose ratings are similar to the target user.

CF can enhance content-based filtering tools: providing filtering of items whose content is hard to analyze; recommendations based on quality and taste; and serendipitous

[1] http://www.google.com/codesearch
[2] http://www.krugle.org

recommendations [4]. DeLine et al. proposes a system that helps program comprehension by filtering and recommending the resources in the project of a developer based on navigation data shared among the project team [3]. We believe that CF approach can improve retrieval systems by achieving filtering that is based on implicit relevance between components and personalized to target user, i.e. suitable for requirements and restrictions of the user's task. In addition, a user may get suggestion of some "better" components than ones that a user considers to reuse.

3 A method for component recommendation

This section introduces a method for recommending software components using CF on search history. The method extends the GroupLens algorithm, which uses correlation between users, for component recommendation. The method consists of following steps.

Gather Ratings for Components: Store browsing history of each user as ratings for components on a database.

Compute Correlation: Obtain correlations between users based on ratings in the database.

Compute Recommendation Score: Obtain scores of all components using ratings made by a user and its correlation to other users.

Recommend Components: Present components for the user based on the scores of the components.

Details of these steps are described below.

3.1 Gather Ratings for Components

User ratings for components are essential for CF. Therefore, recommendation system has to gather ratings voted by users. However, explicit ratings for components are undesirable since effort for the inputting is not negligible. Our method uses *implicit rating* [4] for avoiding the effort. More specifically, a component is rated 1 (i.e. good) by a user *iff* the user browses the source code of the component. Our approach allows users to *cancel* their ratings, i.e., users can improve the recommendation by deleting their browsing history.

CF is based on user correlation and assumes that preferences of users are stable. This assumption is problematic for search systems because user preference varies according to change of search purpose.

When using a search system, rating preference is stable while search purpose is same. Therefore, we employed a search purpose as a *user* in context of CF, and then used similarity between search purposes for component recommendation. Accordingly, the system has to detect changes of search purpose and split browsing log at the changes. But detecting the change from user action is difficult generally. We treat an end of using a search system as a change of search purpose, and ratings from beginning to end of use (i.e. session) as *user* rating. A user and a browsing session will not be distinguished hereinafter except for necessary contexts.

We have to consider about same or similar components, which are copied for reusing, in a search target. Users of search system can arbitrarily choose one component from similar components. However, this choice affects harmful for CF because votes for the similar components are scattered for each components. We adopted a set of similar components as a basic unit of recommendation for alleviating this problem.

3.2 Compute Correlation

CF requires correlation between users for the purpose of speculating users similar to a user to whom the system recommends items.

GroupLens defined a correlation between two users as similarity of ratings for components that are voted *both* of the two users. However, this definition is inappropriate for our method. In our method, vote is done by browsing and ratings for browsed components are always 1. Therefore similarity between sessions is 1 if common component are browsed in the sessions, otherwise similarity is 0. Furthermore, sessions are generally short and contain few records of browsing, sessions rarely share browsed components. So we employed Breese's recommendation algorithm[2] which defines the correlation between two users as similarity of ratings for components that is voted *either or neither* of the two users. Rating for a browsed component is 1, and for not browsed component is 0.

Correlation $c(a,i)$ between user a and i is defined as follows. I_i means the components that is voted by user i, and I means all components that is registered in recommendation system.

$$v_{i,j} = \begin{cases} 1 & \text{if } j \in I_i \\ 0 & \text{if } j \notin I_i \end{cases}, \ \bar{v}_i = \frac{1}{|I|} \sum_{j \in I} v_{i,j}$$

$$c(a,i) = \frac{\sum_{j \in I} (v_{a,j} - \bar{v}_a)(v_{i,j} - \bar{v}_i)}{\sqrt{\sum_{j \in I} (v_{a,j} - \bar{v}_a)^2 \sum_{j \in I} (v_{i,j} - \bar{v}_i)^2}}$$

3.3 Compute Recommendation Score

CF computes a recommendation score for each component based on ratings made by the users whose behavior is similar to a receiver of the recommendation. In particular, recommendation score is an estimation value for the rating that the receiver will vote. An estimation score is a weighted average of scores made by other users; the weight is a similarity for the receiver.

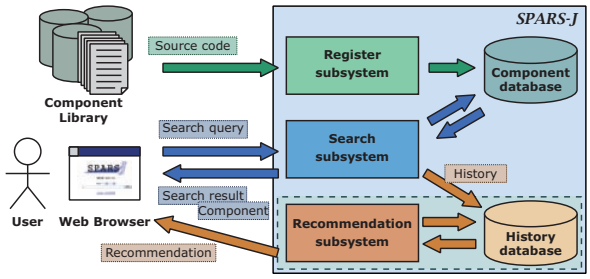


Figure 1. SPARS-J with recommendation subsystem

In our method, a set of the similar users are preliminary computed for speed of recommendation. The set consists of the users who satisfy the following conditions: 1) he/she shares browsed components with a receiver, 2) His/her correlation to the receiver is positive, and 3) he/she browsed two or more components. The weight for the averaging is square of the correlation according to [2].

Consider set of similar user $U = \{i | I_a \cap I_i \neq \phi \wedge c(a,i) > 0 \wedge |I_i| > 1\}$. Recommendation score of component k for user a is defined as follows:

$$p_{a,k} = \frac{\sum_{i \in U} c(a,i)^2 v_{i,k}}{\sum_{i \in U} |c(a,i)^2|}$$

3.4 Recommend Components

The system recommends components in a descending order of the scores. With the exception of recommendation, the components that the receiver has already seen are eliminated from recommendation, since these components are well-known for the receiver.

4 Implementation on SPARS-J

We implemented our recommendation method on SPARS-J, which is a retrieval system for Java components. Our method is built into SPARS-J as recommendation subsystem. Figure 1 is the architecture diagram of SPARS-J with the recommendation subsystem.

4.1 Record browsing history

Our system distinguishes a *session* using a session cookie, which is generated when a searcher accesses to SPARS-J using a web browser and is expired when he/she closes the browser window.

When he or she opens a source code of a component, a vote for the component is stored in the history database with the session identifier that is assigned to the session cookie.

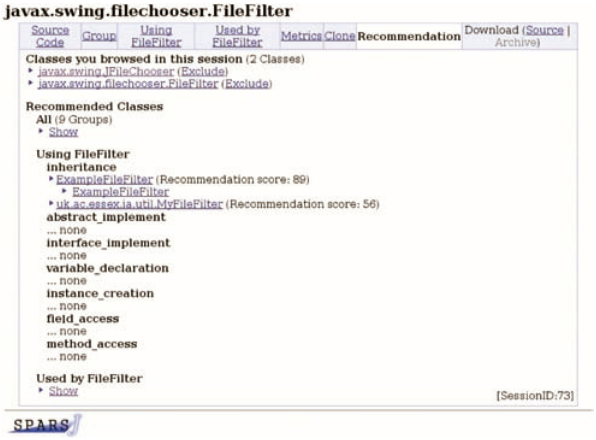


Figure 2. Recommendation page

4.2 Display recommended components

Recommendation page displays a list of recommended components whose recommendation score is higher than certain threshold.

Figure 2 is a sample of recommendation page. The list contains hyper links to details pages for each recommended components. Recommendation score is also shown in the list; the user can refer the score for selecting components.

There are two way to receive recommendation: flat view and relation view. The flat view shows all recommended component in descending order of recommendation score. The relation view displays components classified by those relations to a certain component.

5 Case study

We conducted a case study to evaluate whether our recommendation method can help users to retrieve reusable components in coding tasks. Eight participants (A1 $\sim$ A8) tried four Java coding tasks (P1 $\sim$ P4) with/without the recommendation subsystem. The all participants have enough Java experiments to solve the tasks; they are members of a software engineering laboratory and they have used Java in their research.

In each task, they were asked to implement some feature into skeleton code using only SPARS-J and the documentation of Java API. The resources for running the programs are provided so that the participants can test their programs quickly when they finished to code. The subjects of the tasks are image file conversion (P1), file transfer using FTP (P2), GUI (P3) and database access (P4). We suppose that the number of reused components to finish the each task range from 2 to 5; and the lines of code range from 10 to 50.

Table 1. Result of the case study

	Search time (min)				Precision			
	P1	P2	P3	P4	P1	P2	P3	P4
A1	28	25	**14**	**3**	0.42	0.06	**1.00**	**0.67**
A2	47	19	**60**	**38**	0.53	0.38	**0.50**	**0.64**
A3	14	2	**3**	**7**	0.23	1.00	**1.00**	**0.83**
A4	50	28	**9**	**12**	0.26	0.15	**1.00**	**1.00**
G1 Ave.	34.8	18.5	**21.5**	**15.0**	0.36	0.18	**0.88**	**0.79**
A5	**4**	**2**	9	23	**1.00**	**1.00**	1.00	0.55
A6	**3**	**4**	18	45	**1.00**	**1.00**	1.00	0.63
A7	**28**	**5**	44	13	**0.78**	**0.40**	0.55	0.73
A8	**15**	**2**	26	23	**0.75**	**1.00**	0.37	0.73
G2 Ave.	**12.5**	**3.2**	24.2	26.0	**0.89**	**0.73**	0.73	0.66

We set up SPARS-J for the case study with the database comprising Java API and demos, open source software packages and samples retrieved from WWW. The database contains enough reusable components and their sample codes to accomplish the tasks. Total number of components is about 35,000. The case study begins with an empty history database

5.1 Procedure

Before the case study, every participant takes one-hour lecture about the system usage including a training task. We grouped the participants into two groups (G1 and G2) so that the groups have similar programming ability based on the memers' Java experiments and the time spent to accomplish the training task.

At first, the members of G1 (A1 $\sim$ A4) tries P1 and P2; and the members of G2 (A5 $\sim$ A8) tries P3 and P4 without recommendation. Then, G1 tries P3 and P4; and G2 tries P1 and P2 with recommendation.

5.2 Result and discussion

Table 1 presents the search time and the precision of the retrieved components. The search time indicates (*total time to accomplish the task*) - (*coding time*); and the precision is defined as $|$*the components used for implementation*$| \div |$*the components displayed in SPARS-J*$|$. The participant/task combinations using the recommendation are presented with the boldface. We can see that the recommendation successfully helps users by reducing searching time and improves precision.

The result shows that difference of P3 is smaller than the other tasks. This is likely because A5 and A6, who work on P3 without recommendation, have knowledge on the subject area of P3 enough to navigate themselves to the components available for the task.

We also found that A2 and A7 did not improve the efficiency with recommendation. Their common behavior is that they briefly read a lot of components at the beginning of their activity. As the result, the available components were recorded into the history without recognized as "available" and never recommended. An idea to support such users is changing the policy to gather the rating to ignore components that had been viewed only in a minute.

6 Conclusion

This paper proposed software component recommendation based on collaborative filtering (CF). We also implemented the recommendation system that is integrated into SPARS-J. The result of the case study shows that CF is effective to improve search efficiency of reusable components. Our future work is improving recommendation by sophisticating logic to gather ratings from browsing history.

Acknowledgements This work was supported in part by "Global COE (Centers of Excellence) Program" of the Ministry of Education, Culture, Sports, Science and Technology, Japan; and was conducted in part as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Proc. OOPSLA '06*, pages 25–26, Oct. 2006.

[2] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proc. UAI '98*, pages 43–52, 1998.

[3] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proc. VL/HCC '05*, pages 241–248, 2005.

[4] J. L. Herlocker, J. A. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proc. SIGIR '99*, pages 230–237, 1999.

[5] O. Hummel and C. Atkinson. Using the web as a reuse repository. In *Proc. ICSR '06*, pages 217–230, July 2006.

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

[7] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: An open architecture for collaborative filtering of netnews. In *Proc. CSCW '94*, pages 175–186, 1994.

[8] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. AOSD '07*, pages 212–224, 2007.

Lowering the Barrier to Reuse through Test-Driven Search

Werner Janjic, Dietmar Stoll, Philipp Bostan, Colin Atkinson
Chair for Software Engineering
University of Mannheim, Germany
{janjic, stoll, bostan, atkinson}@informatik.uni-mannheim.de

Abstract

Dedicated software search engines that index open source software repositories or in-house software assets significantly enhance the chance of finding software components suitable for reuse. However, they still leave the work of evaluating and testing components to the developer. To significantly change the risk/cost/benefit tradeoff involved in software reuse, search engines need to be supported by user friendly environments that deliver code search functionality, non-intrusively, right to developers fingertips during key software development activities and significantly raise the quality of search results. In this position paper we describe our attempt to realize this vision through an Eclipse plug-in, Code Conjurer, in tandem with the code search engine, merobase.

1. Introduction

The vision of systematically assembling software applications from prefabricated parts is as old as software engineering itself (McIlroy presented a paper on Mass-Produced Software Components at the NATO conference that coined the terms software engineering and software crisis [10]). However, despite significant research effort into reuse in the 1980s and 1990s, McIlroys vision has remained stubbornly elusive. Over the years there were many reasons why fine-grained component reuse failed to take off, but generally speaking there have been three main barriers [3] –

1. there simply were not enough good components around to make reuse worthwhile,
2. the recall and precision of the retrieval technologies used to find suitable components was not sufficient,
3. the overall risk and effort involved in finding and evaluating components for reuse was too high compared to the risk and effort involved in building them from scratch.

Over the last few years there have been dramatic improvements with respect to the first two of these. The rapid growth in freely available, open source software repositories such as SourceForge and Google Code as well as the emergence of dedicated search engines that index them (such as Google Code Search, Krugle and merobase) now provide developers with easy access to vast swathes of reusable software. However, these advances have only partially alleviated the third problem. They are necessary but not sufficient. Although the precision of some of the new generation of code search engines is much higher than before [5], the ratio of suitable to non-suitable components in search results is still relatively low and developers have to evaluate them all by hand. The costs and risks involved in manual reuse by directly interacting with code search engines therefore still typically outweigh the benefits.

To fundamentally change the risk/cost/benefit balance and make fine-grained component reuse the rule rather than the exception Garcia et al. [1] argue that component search facilities need to be integrated into a fully fledged software reuse environment. Such an environment should (a) allow reuse recommendations to be driven by a background agent that monitors the work of the developer and triggers searches proactively (b) provide automatic assistance for query formulation to bridge the gap between the described functionality of a component and the described needs of the developer and (c) make reuse as non-intrusive as possible so that the developer is barely disturbed from his normal work. We believe that integrating search functionality seamlessly and unobtrusively into standard development environments is only one half of the solution, however. We also believe it is important to substantially raise the quality of research results. Even if component search functionality is offered in a highly unobtrusive way, it will still not be used unless there is a reasonable likelihood that the effort and risk involved in evaluating a component will be worthwhile. We believe the best way to enhance the quality of the results is to exploit the fact that code, unlike most other documents indexed by text-driven search engines, is executable. This means that a components fitness for purpose can be estab-

lished by testing it. This presupposes the existence of test cases that can be used to test components, but fortunately the trend in modern development approaches such as agile development is to develop test cases before writing code.
In this paper we present a tool, Code Conjurer, (www.code-conjurer.org), that implements these features using the merobase software component search engine (www.merobase.com).

2. Proactive Reuse Recommendation

Several prototype tools have been developed to provide assistance to developers based on information garnered from code repostories. The chief examples include Rascal [9], Prospector [8], ParseWeb [12] and Strathcona [2]. These help developers to work out what methods to call in what sequences or provide examples of previous ways in which a component has been reused. However, none is directly focused on finding reusable components and none provides support for proactive recommendations.
The first tool to offer proactive help to users based on information garnered from a code repository was CodeBroker [13]. This tool was focused on reusing good design and coding practices rather than fully blown components per se, but it pioneered the notion of proactive recommendation. The main weakness of CodeBroker is that it requires components to be annotated by developers and is unable to handle normal software modules. Finally, CodeGenie [7] is an Eclipse plug-in that focuses on finding reusable components. However, it is not proactive and requires developers to manually test all reuse candidates locally in their development environments. At the University of Mannheim we have been working on a plug-in known as Code Conjurer [6] that uses the merobase code search features to realize a software reuse environment of the kind envisaged by Garcia et al. [1] within the Eclipse framework. When plugged into the standard Eclipse Java environment, it allows searches to be initiated from various kinds of Java code fragments at the click of a button. When set into proactive mode the plug-in also provides proactive reuse recommendations. It includes an agent that monitors the component under development (CUD) and autonomously recommends potentially interesting candidates for reuse in a non-intrusive way.
When the background agent discovers a significant change in the CUDs interface-defining part (e.g. a method has been added, changed or removed) it triggers a search via the Merobase API. The component is analyzed, its interface is extracted, an MQL (Merobase query language) query is created and user-defined constraints are added (e.g. duplicate filtering or exclusion of interfaces). The resulting list of components is presented to the developer in an Eclipse view, as shown in the bottom left of figure 1. He can then study the components in more detail, review the implemented methods and compare components using different metrics.
If the developer decides that a component is worth reusing, by a simple double-click he can either weave it into the current project, thus overwriting his own code, or can put it into a new project. When the component is inserted into a new project, Code Conjurer automatically detects unresolved dependencies and tries to automatically resolve them (provided that this functionality is activated in the preferences). During development it may also happen that the component under development needs a new kind of object (e.g. when writing an address book a Person object might be necessary). In this case the developer can simply specify the object (e.g. with `Person p = new Person();`) which will lead to an error message displayed by Eclipse indicating that the type cannot be resolved. Using the QuickFix feature of Eclipse, the developer can easily get Code Conjurer to search for a Person component and afterwards directly add it to his project thereby avoiding self-development.
Even if the developer does not wish to use one of the recommended components, Code Conjurer provides potentially interesting information about the typical or average form of the discovered components. Using various clustering techniques, the recommended components are analyzed and a characteristic group picture is created. This information indicates the typical set of methods offered by components matching the developers partially defined interface. For example, suppose the developer is working on a class Polynomial, Code Conjurer can indicate that classes of this name typically offer the following methods:

```
public class Polynomial {
   Polynomial add(Polynomial arg1) {}
   String toString() {}
   int getDegree() {}
}
```

In contrast to the software reuse environment envisaged by Garcia et al. [1] which only foresees automatic help in query formulation, Code Conjurer extracts all necessary information automatically from the CUD and creates the search queries itself without user involvement. Moreover, the developer does not have to write code according to any particular standard or worry about interacting with the search engine but can fully concentrate on developing his application.

3. Test-Driven Software Reuse

Code Conjurer seamlessly and unobtrusively integrates search functionality into the Eclipse development environment, but this still does not ensure that it will be reused in practice. As mentioned above, the value of a software reuse

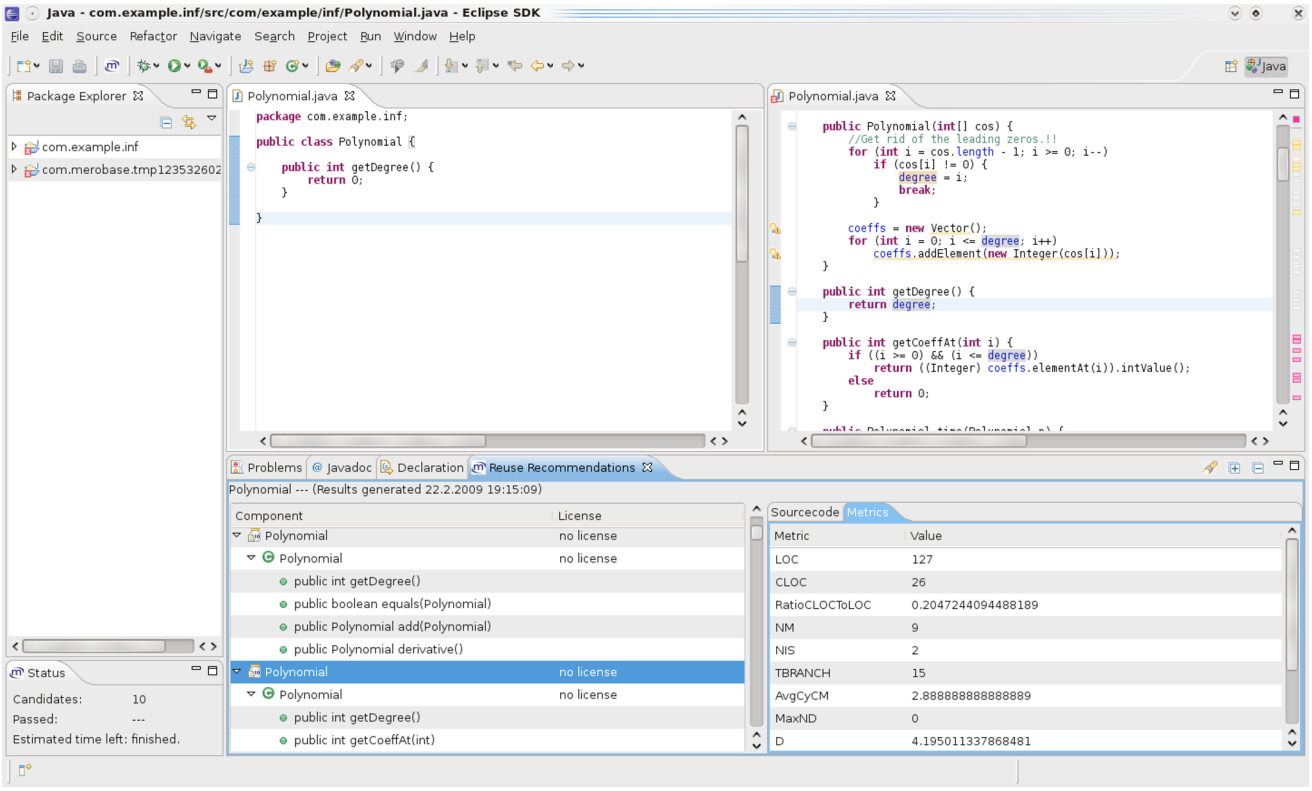


Figure 1. Proactive reuse

environment is significantly diminished if the quality of the search results is low and the developer has to spend a lot of time manually studying and evaluating components. As Mili and Mittermeir pointed out in 1998 [11], software artifacts are not textual documents but are executable modules with observable behavior. Thus, it is possible to test their fitness for purpose by checking whether they pass one or more tests. The merobase search engine has therefore been enhanced to support test-driven as well as standard search mechanisms [6]. As soon as an executable test has been defined by the developer and a search is initiated, Code Conjurer sends a request to the merobase server to find matching components that pass the test. Merobase then initiates the test-driven searching process, which in previous papers we have referred to as Extreme harvesting [4] because of its synergy with Extreme Programming. This involves the following mains steps -

1. establishing what interface the test is for,
2. performing a normal search on this interface, and
3. testing the results against the provided test-case to filter out those components that match

The resulting component recommendations are of much higher value than those typically generated by regular interface-based matching alone, because the components do what the developer has specified - that is, they pass the provided test-case. Suppose, for example, that the developer needs a Matrix component for his application. He starts by writing a test-case for a Matrix, specifying all the desired functionality. Code Conjurer then sends this to merobase where the interface defining part of the component is extracted and candidates are identified. In a secured virtual environment, these are then tested against the test-case and only those that match are returned to the developers IDE where they are highlighted in green.. The components can be directly weaved into the developers current project or into a separate one.

It may happen that the initial test is only partially complete so that the components in the recommendations view are a superset of the components that are ultimately of interest to the developer once he has finished writing test cases. Nevertheless, at an early point in the process of writing test cases the developer may already be interested in a group picture of what methods components of the kind he is writing generally implement. Code Conjurer allows him to explore the characteristic group picture and write tests accordingly. This kind of reuse is already mentioned in [13] where it is referred to as glass-box reuse.

4. Conclusion

We believe that tools like Code Conjurer that combine proactive reuse recommendation with test driven reuse can for the first time significantly tip the risk/cost/benefit tradeoff between reuse versus build towards the reuse option. The two technologies are also highly synergistic and complement each others weaknesses. The latter complements the former because it significantly enhances the quality of the search results. In fact the precision of the results from test-driven search is theoretically 1 (the recall is hard to estimate [5]) because all returned components are guaranteed to fulfill the developers functional requirements as defined by his test case. The former complements the latter because it hides the relatively long search times and low success rates of test driven search. If a developer is not even aware that a test-driven search is being performed on his behalf, the time taken or success ratio is of no concern to him. Any potentially reusable components that the tool is able to conjurer up are simply seen as a bonus.

Code Conjurer also provides various other helpful features. For example, component recommendations are generally accompanied with metrics information like the LOC, cyclomatic complexity or Halstead metrics so that non-functional properties of components can be evaluated (bottom right window of figure 1). As well as finding normal functional components Code Conjurer can also find reusable test cases as well. When a developer starts writing a test, Code Conjurer can look for previously indexed tests and offer these for reuse. These can be inspected to give the user an impression of what tests are generally written for componentsl similar to that he is developing (glass-box reuse), or they can be weaved directly into the developed project and extended or changed as necessary. Our tool also has a dependency resolution feature which analyzes selected components with respect to unresolved dependencies and tries to resolve them using several heuristics from fast and simple ones to more sophisticated ones. If it finds the needed components, it automatically incorporates them at the necessary places so the error messages of Eclipse disappear.

To conclude, we believe that Code Conjurer, driven by merobase, fulfills the basic vision of a software reuse environment outlined by Garcia et al. [1]. Nevertheless the technology is only just scratching the surface of the development support that can be offered by tools driven by code search engines, and once the remaining possibilities are elaborated we believe the technology will open up a whole new paradigm of search-driven reuse.

Code Conjurer is released under the GNU General Public License v3 and hosted at Sourceforge. More information and some demo videos can be found at the project website www.code-conjurer.org

References

[1] V. Garcia, D. Lucrédio, E. Almeida, R. Fortes, and S. Meira. Toward a Code Search Engine Based on the State-of-Art and Practice. In *the 13th IEEE Asia Pacific Software Engineering Conference (APSEC), Component-Based Software Development Track*, pages 61–70.

[2] R. Holmes, R. Walker, and G. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 952–970, 2006.

[3] O. Hummel. *Semantic Component Retrieval in Software Engineering*. PhD thesis, University of Mannheim, 2008.

[4] O. Hummel and C. Atkinson. Supporting Agile Reuse Through Extreme Harvesting. *LECTURE NOTES IN COMPUTER SCIENCE*, 4536:28, 2007.

[5] O. Hummel, W. Janjic, and C. Atkinson. Evaluating the efficiency of retrieval methods for component repositories. In *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007), Boston, Massachusetts, USA, July 9-11, 2007*, pages 404–409. Knowledge Systems Institute Graduate School, 2007.

[6] O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *Software, IEEE*, 25(5):45–52, 2008.

[7] O. Lemos, S. Bajracharya, and J. Ossher. CodeGenie:: a tool for test-driven source code search. 2007.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61. ACM New York, NY, USA, 2005.

[9] F. McCarey, M. Cinnéide, and N. Kushmerick. Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review*, 24(3):253–276, 2005.

[10] M. McIlroy. Mass produced software components. *Software Engineering Concepts and Techniques*, pages 88–98, 1969.

[11] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[12] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213. ACM New York, NY, USA, 2007.

[13] Y. Ye. *Supporting Component-Based Software Development with Active Component Repository Systems*. PhD thesis, University of Colorado, 2001.

Programmable Queries, or a New Design of Search Tools

Toshihiro Kamiya
Optimized Design Research Team, Center for Service Research
National Institute of Advanced Industrial Science and Technology
Akihabara Dai Bldg. 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 101-0021, Japan
t-kamiya@aist.go.jp

Abstract

This paper presents a novel design of search tools in reverse engineering, which enables describing core searching tasks (such as pattern searching, extraction, filtering, etc.) in a separated way from the management task of location data (such as line number or file name). By using example programs with a prototype implementation, we explain how the proposed design differs from a traditional design, and how the programs help the implementation of customizable tools.

1. Introduction

Coding conventions, dialects of programming languages, heuristics for filtering, etc. require the customization of reverse engineering tools. To make the development of such customizable tools a real possibility, this paper presents a new design for search tools utilizing a gimmick named the *origined string* for separating core-searching tasks from the management task of location data. This design helps a tool developer focus on a core-searching task written as a cohesive code, in terms of SoC (separation of concerns) and the coupling among (traditional) modules.

2. Traditional design

Basically, a search tool executes the following steps sequentially:

1. Inputs documents such as source files (hereafter referred to as a *physical representation*) and converts them to some internal representation, which is well-suited for searching algorithm(s).
2. Performs searching algorithm(s) for the internal representation.
3. Outputs the search results by formatting them into locations in the physical representation.

These steps imply that the developer also has to write code to store a mapping from the internal representation into the physical representation at Step 1, and write code to recall the mapping at Step 3. As a matter of course, if the data structure of the internal representation is to be converted or modified in Step 2, then additional code will be required to maintain such mapping. Therefore, both core-searching tasks and maintenance tasks are co-located in each step, and their codes tend to entangle easily with each other, increasing the threat of losing cohesiveness, understandability, and/or modifiability.

3. New design by origin-aware string

If we can make primitive data types (which are used in the internal representation) store such location data and also maintain the mapping through the operations that manipulate the data types, then the maintenance task will be separated and automated.

In this research, we chose a string as such a primitive data type. Simply said, this altered string, named an origined string, is an object that knows its origin and is manipulated just like a string. A tool developer can construct an instance of an origined string from some text file and apply various operations (such as calculating substrings, concatenating, or comparing) to it. Thereafter, when needed, the developer can ask each instance, “Where do you come from?”

With the origined string, the search tool steps mentioned in section 2 become as follows:

1. Inputs documents such as source files, and converts them into origined strings (or some data structures including origined strings) of the internal representation.
2. Performs searching algorithm(s) for the internal representation.

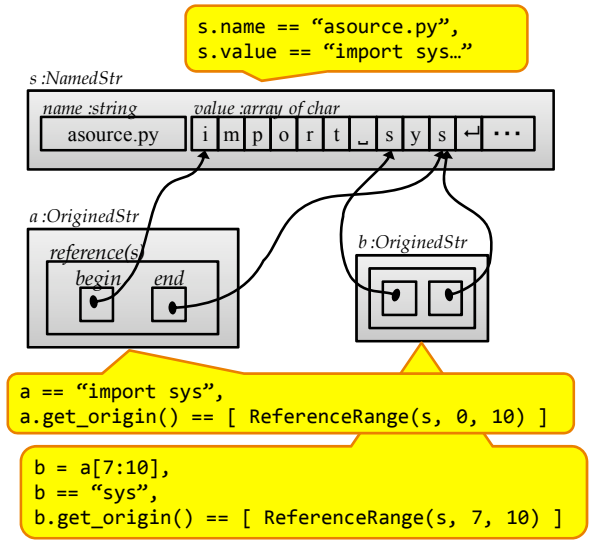


Figure 1. Illustration of NamedStr/OriginedStr

3. Outputs the search results by asking their location in the physical representation in relation to each of the search results.

At Step 1, the mapping from the internal representation into the physical representation will be generated and stored by origined strings. In Step 2, through operations to the origined strings, the mapping is kept updated by itself, in an implicit (in source code) and background (in program execution) way.

A prototype toolkit named *pyremics* has been implemented in Python programming language, in order to evaluate the proposed design with the origined string. There are two primal classes: `OriginedStr` and `NamedStr` (Fig. 1). The class `NamedStr` is intended to represent a source file (or any text file). The class has instance variables, `name` (file name) and `value` (text, content of the file). The class `OriginedStr` is intended to represent a substring (or list of substrings) of `NamedStr`, by storing each substring as a position in an instance of `NamedStr`.

The `OriginedStr` has two API sets. The first one includes methods as string types, such as calculating substrings, concatenating, and comparing. The second API set asks for locations, such as `get_origin` in Fig. 1. The methods as string types are designed and implemented to have the same interface and semantics as the primitive string type (class `str`), so that `OriginedStr` is interoperable with standard libraries and third-party libraries, as shown in the following section.

Note that the above implementation of the origined string is somehow inspired by a high-performance string type named *Rope*[1], despite having distinct primal design goals; origined strings aim for ease of development (including automation, understandability etc), while Rope works on performance issues.

```
 1 import os, re, sys
 2
 3 pattern = re.compile(sys.argv[1])
 4
 5 # read files and search pattern in files
 6 for root, dirs, files in os.walk(""):
 7   for f in files:
 8     if f.endswith(".py"):
 9       path = os.path.join(root, f)
10       lineNum = 1
11       for line in file(path):
12         for m in pattern.finditer(line):
13           print "%s %d: %s" % (path,
14               lineNum, m.group())
15         lineNum += 1
```

Figure 2. Program simplegrep.py

4. Examples

Entire source code in this paper is written in Python and they are executable, not pseudo codes (the source files are available at http://www.remics.org/).

4.1. Example #1: simplified grep

For comparison, a simplified *grep* program (Fig. 2) was implemented without using the toolkit. When the program is run, in loops starting at Line 6, the program searches source files in the current directory, reads each line of each file, and prints a line number for each time the pattern matches. Note that the code for maintaining the location data (that is, a variable `lineNum`)

```
1 import os, re, sys
2 import pyremics as rm
3
4 pattern = re.compile(sys.argv[1])
5
6 # read files
7 fileTable = rm.FileTable()
8 fileTable.read_from_directory("",
9   re.compile(".+[.]py$"), recursive=True)
10
11 # search pattern in files
12 matchedStrs = []
13 for _, ns in sorted(fileTable.iteritems()):
14   for m in pattern.finditer(rm.origined_str(ns)):
15     matchedStrs.append(m.group())
16
17 # print matches
18 for s in matchedStrs:
19   print "%s: %s" % (s.to_row_str(),  s)
```

Importing the toolkit!

Figure 3. Program easygrep.py

is mixed with the code to search for a pattern, and additional variables will be needed for untangling the deeply nested loops to reduce indent levels.

With the toolkit, almost the same program is spelled out as in Fig. 3. In this code, all classes and functions from the toolkit are prefixed with `rm.` When the program is run, first the source files are read, converted to instances of `NamedStr`, and stored in a variable `fileTable` in Lines 6-9. Then, in a routine for searching the pattern of Lines 11-15, a pattern searching is performed for each element of `fileTable` and the match results are stored in a variable `matchedStrs`, as instances of `OriginedStr`. Lastly, in a routine for printing the results of Lines 17-19, location data is extracted from each of the match results at Line 19, with method `to_row_str` of `OriginedStr`.

This example implies:

- **SoC and automated maintenance of location data.** The location data are automatically generated and stored in instances of `OriginedStr`.
- **Low coupling.** The routines are well separated with low coupling by a small number of variables.
- **Interoperability to standard libraries.** An instance of `Originedstr` is safely passed to/received from a standard library `re` (regular expression).

4.2. Example #2: visualization of dependency

The second example is a program to visualize dependencies between source files. The "dependency" of this program is somehow a relaxed, name-based one, that is, when a name defined in a source file *f* is used in a source file *g*, a dependency from *g* to *f* exists.

Fig. 4 shows the source code of the program. The following paragraphs explain each routine of the program. Note that there are only four variables (`fileTable`, `referredIds`, `definedIds`, and `graph`) shared among these routines.

The first routine for reading source files of Lines 5-

```
 1 import itertools, os, re, sys
 2 import graph # python-graph (http://code.google.com/p/python-graph/)
 3 import pyremics as rm
 4
 5 # read source files from current directory
 6 fileTable = rm.FileTable()
 7 fileTable.read_from_directory("", re.compile(".+[.]py$"), recursive=True)
 8 initFiles = [fileTable.pop(p) for p in fileTable.keys() if p.endswith("__init__.py")]
 9
10 # find identifiers
11 s = r'[ruRU]*"(¥¥.|[^"])*"|"""((!?""").)*"""'
12 pat = re.compile("|".join([r"¥b(?P<def>(def|class)¥s+)?(?P<id>[a-zA-Z_]¥w*)", # identifier
13     s, s.replace('"', "'"), r"#[^¥n]*¥n"])) # string literals, comment
14 referredIds, definedIds = [], []
15 for _, ns in sorted(fileTable.iteritems()):
16   for m in pat.finditer(rm.origined_str(ns)):
17     id = m.group('id')
18     if m.group('def'): definedIds.append(id)
19     elif id: referredIds.append(id)
20 referredIds = filter(set(definedIds).__contains__, referredIds)
21
22 # build graph of dependency (by name) between the source files
23 graph = graph.digraph()
24 graph.add_nodes(ns.name for ns in fileTable.values())
25 for defName, defs in itertools.groupby(sorted(definedIds)):
26   defFiles = [id.origin0.namedstr for id in defs]
27   refFiles = [id.origin0.namedstr for id in referredIds if id == defName]
28   if not set(refFiles) - set(defFiles): continue # skip if the id is used only in one file
29   graph.add_node(defName)
30   for f in defFiles: graph.add_edge(defName, f.name)
31   for f in refFiles: graph.add_edge(f.name, defName)
32
33 # draw the graph
34 for ns in fileTable.values():
35   for attr in ("shape", "box"), ("label", ns.name.replace(os.path.sep, "/¥¥n")):
36     graph.add_node_attribute(ns.name, attr)
37 file("graph.dot", "w").write(graph.write(fmt='dot'))
```

Figure 4. Program depgraph.py, a 37-line tool to analyze dependency among source files.

8 is very similar to the one in example #1, except this one gives special treatment to files named "*__init__.py*" (such a file is special because it is used to define a package in Python programming language).

The second routine of Lines 10-20 finds identifiers in the input source files. By checking whether or not a keyword `class` or `def` (at Line 12 and Line 18) appears before each identifier, the routine distinguishes the defined identifiers (that is, names of the classes or functions that are defined somewhere in the input source files) from the just-referred identifiers.

The third routine of Lines 22-31 generates a graph, where each node represents an input source file and each edge represents dependency between two files. For each of the defined identifiers, the routine generates edges between the identifier and the files that define/refer the identifier at Lines 30-31. Here, file names have been extracted directly from identifiers via an instance variable `origin0` of `OriginedStr` at Lines 26-27. Also, a trick at Line 28 reduces the number of nodes in a graph and improves the readability of the graph.

The last routine of Lines 33-37 prints the generated graph to a file. Some formatting is performed at Lines 34-36. The output is a ".dot" file, which can be converted into an image file with the tool *GraphViz* (http://www.graphviz.org/).

Fig. 5 shows a sample output of this program. The input consists of source files of the example programs, the source files of the toolkit, and a test program for the toolkit. A file of the toolkit has the incoming edges from all names because it defines them, while a program *simplegrep.py* doesn't have any incoming or outgoing edges since it doesn't use any names defined in the toolkit. This gives some evidence that the program works as intended.

To evaluate modifiability, we made customized versions of this program (their source code or their sample outputs are not shown in this paper, but are available at the Website). The first one is an abstracted version; a program to generate coupling between source files. The second one is a detailed version; a program calculates the same dependency graph but adds to each edge a label of line numbers where the identifier appears. In these customizations, the modification occurred only in the third routine, up to eight lines.

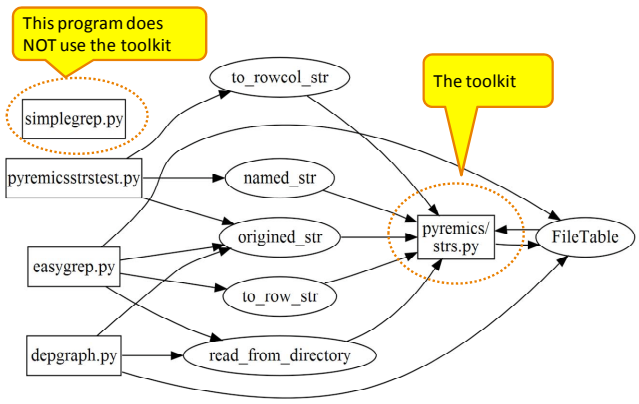


Figure 5. A sample of output

In addition to SoC, automation, interoperability to a standard library and low coupling among routines, this example indicates:

- **Modifiability.** The examples of customization were described in the preceding paragraph. Also, two heuristics in the original program at Line 8 and Line 28 have been introduced after test runs, in which we noticed the poor quality of the output because although correct by definition, unwanted data are included in the result (you can see this by downloading the program and running it after removing these lines). Such kinds of "noise" filtering are often required in analysis with reverse engineering tools.
- **Interoperability to a third-party library.** Instances of `OriginedStr` are passed (just as `str`) to a third-party library, *python-graph* (http://code.google.com/p/python-graph/).

5. Conclusions and further challenges

In this paper, we presented a new design in order to separate core-searching tasks from location-data management tasks and to automate the latter. The examples using the prototype toolkit *pyremics* show how the design/toolkit in automation achieves cohesiveness in terms of SoC and coupling between modules, interoperability, and modifiability.

In the future we will:

- Extend the toolkit for interoperability to AST-based parsers.
- Port to other programming languages. For example, a template (or concept-based) type system of C++ affords implementation to a toolkit with interoperability to existing libraries, as well as a dynamic type system, such as Python does.
- Develop real (not toy) applications with the toolkit. This also works as an evaluation of scalability of the toolkit.

References

[1] Hans-J. Boehm, Russ Atkinson, and Michael Plass, "Ropes: an alternative to strings", Software: Practice and Experience, vol. 25, issue 12, pp. 315 - 1330 (1995).

Exploring Java Software Vocabulary: A Search and Mining Perspective

Erik Linstead[1,2], Lindsey Hughes[2], Cristina Lopes[1], Pierre Baldi[1]
[1] School of Information and Computer Sciences. University of California, Irvine.
[2] Department of Math and Computer Science. Chapman University, Orange, CA.
elinstea@ics.uci.edu, hughe120@mail.chapman.edu, lopes@ics.uci.edu, pfbaldi@ics.uci.edu

Abstract

We conduct a large-scale analysis of Java source code vocabulary for 12,151 open source projects from SourceForge and Apache, a corpus substantially larger than considered previously. Simple statistical analysis demonstrates robust power-law behavior for word count distributions across multiple program entities. We then identify salient vocabulary trends for classes, interfaces, methods, and fields. Our results provide low-level insight into the vocabulary space governing Java software development, with direct application to program comprehension and software search. Supplementary material may be found at: http://sourcerer.ics.uci.edu/suite2009/suite.html.

1. Introduction

As the amount of publicly available source code continues to grow, so does the need for tools that can effectively search and mine software repositories to facilitate code reuse and automated program comprehension. Advances in information retrieval (IR) and text mining (TM) have acted as catalysts for such tools, allowing software developers to increase efficiency, reduce implementation redundancy, and effectively understand arbitrary amounts of source code at multiple levels of granularity.

While the techniques used in contemporary software search and mining engines vary greatly, there is general acceptance that source files are just another collection of documents and the tokens that comprise them are just another vocabulary [2]. Thus, even though programming languages differ from natural language in syntax and convention, many of the same approaches to search and mining that apply to English documents, for example, can be extended to a programming language. Indeed, traditional text information retrieval and modeling algorithms such as tf-idf and latent dirichlet allocation have yielded high performance when applied to software artifacts [6].

Despite the fact that many natural language-based techniques have been augmented for the purposes of code search and mining, developers of such tools still lack many of the standardized resources available in IR and TM domains. Stopword lists, for example, play an essential role in text indexing and search by pruning tokens of little value. While such lists are easily located for natural languages such as English and Arabic, community-vetted lists are not available for C++ and Java. The same is true for part-of-speech taggers, query analyzers, and other components essential in code search and mining. While there are many reasons why this is the case, we believe an aggravating factor is the lack of a large-scale statistical analysis of programming language vocabularies capable of supporting the generation of such resources. Text corpus vocabulary is at the heart of the IR and TM problem space, and though experiments have been carried out in the past to understand the word space of small software collections, we still need an Internet-scale understanding of software vocabulary that can form a foundation for Internet-scale search and mining.

Here we provide an Internet-scale vocabulary analysis for the Java programming language by investigating the word space of class, interface, method, and field names for 12,151 projects. Our results demonstrate strong evidence of power-law behavior for word distributions across program entities, and highlight distinct vocabulary features for each entity type, including part-of-speech data. Our hope is that this analysis will fuel additional progress in the creation of standardized resources for software search, with application to indexing, ranking, topic modeling, and query assessment. On a more fundamental level, it is our intent to demonstrate that the complete understanding of low-level source code vocabulary trends has great promise for further advancing these areas.

2. Data

To allow for the Internet-scale analysis of source code we have built an extensive infrastructure, Sourcerer [6], designed for the automated crawling, parsing, and storage of large software repositories in a relational database. The cur-

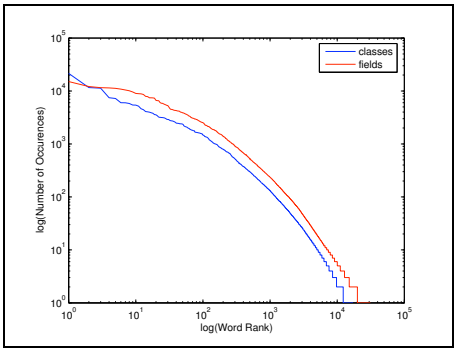


Figure 1. Power-law Distributions for Class and Field Vocabulary.

rent version of Sourcerer consists of 12,151 Java projects from Sourceforge, Apache, and other public repositories. When parsed these projects yield 514,369 classes, 43,965 interfaces, 2,902,186 methods, and 1,560,627 fields. The names of these entities form the basis for our vocabulary study.

3. Methodology

For this study we consider code entities of the types class, interface, method, and field. By further processing the fully qualified names (FQN's) of the entities stored in Sourcerer we are able to produce the vocabulary that underlies the corpus, as well as the number of occurrences for each word in the repository.

Processing FQN's consists of two distinct steps. The first is to extract the words that constitute the entity name, where the entity name is the rightmost portion of the FQN. In other words, we do not consider the name of the encapsulating package as part of the proper entity name. Once the proper entity name is obtained, it is tokenized to obtain individual vocabulary words. This tokenization is accomplished by leveraging common naming heuristics for Java software. For example, the class name "DynamicConfigurator" will generate the words "dynamic" and "configurator." Counts of each word are maintained and used to determine distributions for individual entity types, as well as to investigate fundamental vocabulary differences across types. As an additional step a stemmer can be applied to each word, but this was not done for the initial empirical analysis carried out in this paper. Parsing our full Java corpus yields a vocabulary of 44,060 distinct words. When examined individually, classes boast a vocabulary size of 17,286, interfaces a size of 5,091, methods a size of 23,699 and fields a vocabulary of 31,231 words.

The second step in our vocabulary analysis consists of tagging individual words with their parts of speech. This allows us to investigate "grammatical" differences between entity types, thus providing additional opportunities to leverage vocabulary structure in tools for search and mining. Validating that methods, for example, contain more verbs than their class counterparts may be useful in pruning the search space of code queries, or used as a classification feature in software-oriented machine learning. Since a part-of-speech tagger trained on source code is not readily available, we rely on the implementation described in [7].

4. Results

In this section we provide an overview of our analysis results. Complete results may be downloaded from the supplementary materials page.

4.1. Word Distributions

As a first step in exploring the vocabulary of our Internet-scale software repository, we consider the distribution of word counts. In doing so, one is immediately struck by the fact that the vocabulary for the entire repository exhibits strong power-law behavior. Power-law distributions have been observed in a wide variety of domains, ranging from word frequencies in text to molecular substructures in chemical informatics. Various structural aspects of software have been shown to follow power-law distributions, and it appears this model also explains the shape of Java vocabulary in the large scale. Indeed, power-law distributions were observed for individual entity types as well. Figure 1 depicts power-law behavior for class and entity word distributions, as demonstrated by the straight line on the log-log plot, ignoring boundary effects. (The reader may refer to [6] for a mathematical review of power-law distributions, as well as a survey of previously identified power-laws in software.)

Understanding the word distributions of entity types proves useful for honing in on interesting areas of vocabulary space. For example, one may choose to set a threshold on the high end of the distribution, and use all words occurring above that threshold to construct a stopword list. Similarly, one may focus on the low end of the distribution to locate rare words that may prove useful in distinguishing entity types, or identify vocabulary entries in need of further processing. While the ultimate ramifications of vocabulary power-law behavior are still to be digested by the community, being able to model the functional form of Java word distribution poses both theoretical and practical opportunities.

Table 1. The Ten Most Frequent Words by Entity Type.

Rank	Class		Interface		Method		Field	
	Word	*Count*	*Word*	*Count*	*Word*	*Count*	*Word*	*Count*
1	1	21478	event	740	get	768519	default	15042
2	2	11581	data	630	set	365749	type	12122
3	test	11288	object	408	is	124501	text	11552
4	3	7472	xml	388	test	79472	version	11422
5	file	7213	type	383	to	75232	is	11215
6	abstract	6000	service	377	add	70846	file	10829
7	data	5982	file	370	action	59264	serial	10460
8	table	5816	standard	368	create	57963	label	10083
9	list	5432	change	337	remove	29896	max	9604
10	type	5405	message	331	do	26696	to	8990

4.2. Vocabulary Trends

Having considered the general form of word distributions, a natural next step is to examine the words themselves for telling trends and patterns. Table 1 lists the 10 most frequent words for each entity type considered. Examining this table one makes several observations. The first is that cardinal numbers account for the two most frequent words in classes, but do not appear in the top 10 words for any of the remaining entity types. Turning back to the raw data it appears that these numbers are used frequently to differentiate different implementations of the same general type, and are thus more prevalent in class names relative to overall vocabulary size. Another striking observation is that most frequent words in the method vocabulary are dominated by verbs, as opposed to other entity types, which are dominated by nouns. Not surprisingly the most frequent words for methods are 'get' and 'set,' stemming from the use of accessor methods for reading from and writing to encapsulated class fields.

In addition to frequency, one may also differentiate entities based on words that are unique to their individual vocabularies. Our results identify 196 words unique to classes, 0 words unique to interfaces, 344 words unique to methods, and 714 words unique to fields. A small selection of such words appear in Table 2. Again, some of the same trends are found, most noticeably that classes and fields are dominated by nouns, while over 90% of method-specific words in the table are verbs. Moreover, looking at unique field words one finds several entries (eg. 'menuitem') that are concatenations of English words. This phenomenon is attributed to both a limitation in our parsing techniques and what appears to be a prevalent coding practice. We rely on case changes or common delimiters to tokenize entity names. In these cases the names consist of several distinct words, but the developers did not employ any techniques to increase readability. In almost all cases these names can be mapped to private class member variables. In discussing this matter, we came to the conclusion that developers are apparently more likely to adhere to best practices for public APIs, taking a more relaxed approach to private and local identifiers. From the perspective of search and mining, more sophisticated techniques are needed to parse such names for purposes such as concept location.

As a final component to our analysis we consider the prevalence of various parts-of-speech for individual entity types. Table 3 depicts, for each entity type, the percentage of words assigned to each part of speech for the N most frequent words. Results are reported for varying thresholds of N. (Note that some parts-of-speech are not included due to space constraints.) The results present quantitative evidence that verbs are most prevalent in methods for all levels of N, and are especially dominant for moderate levels of N. When complete vocabularies are considered, adjectives and adverbs are also marginally more prevalent in method names. The data also supports our previous observations that cardinal numbers are most prevalent in frequently occurring class words. In examining the output of our tagger we did notice several misclassifications. The word 'test,' for example, was classified as a noun for both classes and methods, though in the context of methods its usage was intended to be a verb. Such errors can be corrected manually, but further supports the need for taggers trained on code.

5 Applications

Our study only scratches the surface of the many types of vocabulary trends that can be used to improve search and mining techniques. Obvious applications include the creation of standardized stopword and concept lists, augmentation of code search queries based on parts-of-speech, as well as the classification of code entities based on word fingerprints. In the latter case, we have leveraged the vocabulary trends discussed here to train first-order Markov Models to classify class, interface, and method entities. In one experiment based on partitioning the Sourcerer database into static testing and training corpora we achieved 78% classification accurracy. This level of accurracy is striking given

Table 2. Unique Words by Entity Type.

Class	Method	Field
decomposes	qualifies	sacrificial
debrief	violate	menuitem
combinations	disambiguate	nodename
simplistic	commence	accesslevel
reconstruction	advertise	neptune
trainer	reify	uranus
interpolator	sanitize	saturn
analytical	formulate	fontcolor
synchronizer	definite	tabletype
subordination	nominate	toolbutton

the simplicity of first-order models, and underscores the importance vocabulary plays in discerning entity types. In this experiment we noted that 74% of misclassifications were due to confusion between classes and methods. We are currently moving toward higher-order models to leverage additional word cues and part-of-speech data to reduce such confusion.

In addition to classification, such vocabulary-centered Markov Models may be adapted to code assessment and search. Specifically, the statistical structure of the model allows one to efficiently calculate the likelihood that a given string was generated from a specific model (in our case a model for classes, interfaces, methods, or fields). In this way, programmer naming conventions for each entity type could be run through the appropriate model and analyzed for their adherence to practices observed in large software repositories. Turning to search, query strings based on entity names could be analyzed and re-written to reflect the vocabulary structure most suited for the type of interest, thus improving precision and recall subject to the tokenization rules used to generate the search index. A similar approach could be taken using part-of-speech data, adding additional refinements above and beyond the usual tf-ids approach.

6. Related Work

Our analysis of Java vocabulary extends previous efforts to understand software language. These efforts include building a grammar for function identifiers [3] and deducing the meaning of common verbs in the context of software [5] for repositories numbering in the tens of projects. An in-depth survey of previous studies is undertaken in [4], which examines the content of comments and identifiers to understand how domain terms are expressed within them and underscores the importance of vocabulary selection in such tasks. Recently, the work in [1] has considered the word structure of Java class names. This work is perhaps most similar to the direction we are exploring, but differs from own analysis thus far in that interfaces, methods, and fields are not considered, nor is part of speech tagging.

Table 3. Part-of-Speech Percentages Across Entity Types for Varying Thresholds.

Type	N	Noun	Verb	Adj	Adv	Num
class	10	0.6000	0.0000	0.1000	0.0000	0.3000
interface	10	0.8000	0.1000	0.1000	0.0000	0.0000
method	10	0.2000	0.7000	0.0000	0.0000	0.0000
field	10	0.7000	0.1000	0.1000	0.0000	0.0000
class	100	0.7500	0.0400	0.0700	0.0100	0.0900
interface	100	0.8900	0.0300	0.0700	0.0000	0.0000
method	100	0.5800	0.2300	0.0700	0.0000	0.0000
field	100	0.7600	0.0500	0.1000	0.0200	0.0000
class	1000	0.7130	0.1030	0.1360	0.0080	0.0110
interface	1000	0.7440	0.0920	0.1260	0.0070	0.0020
method	1000	0.6200	0.1720	0.1620	0.0080	0.0020
field	1000	0.6690	0.1160	0.1450	0.0130	0.0020
class	all	0.6397	0.1576	0.1806	0.0085	0.0021
interface	all	0.6498	0.1450	0.1797	0.0084	0.0012
method	all	0.6208	0.1686	0.1870	0.0109	0.0020
field	all	0.6664	0.1439	0.1701	0.0082	0.0017

References

[1] C. Anslow, J. Noble, S. Marshall, and E. Tempero. Visualizing the word structure of java class names. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 777–778, New York, NY, USA, 2008. ACM.

[2] P. Baldi, C. Lopes, E. Linstead, and S. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, New York, NY, USA, 2008. ACM. to appear.

[3] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 112, Washington, DC, USA, 1999. IEEE Computer Society.

[4] S. Haiduc and A. Marcus. On the use of domain terms in source code. *International Conference on Program Comprehension*, 0:113–122, 2008.

[5] E. W. Host and B. M. Ostvold. The programmer's lexicon, volume i: The verbs. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[6] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: Mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 2009. in press.

[7] K. Toutanova and C. D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*, pages 63–70, Morristown, NJ, 2000. Assn for Computational Linguistics.

Improving Software Quality via Code Searching and Mining

Madhuri R. Marri
Department of Computer Science
North Carolina State University
mrmarri@ncsu.edu

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
xie@csc.ncsu.edu

Abstract

Enormous amount of open source code is available on the Internet and various code search engines (CSE) are available to serve as a means for searching in open source code. However, usage of CSEs is often limited to simple tasks such as searching for relevant code examples. In this paper, we present a generic life-cycle model that can be used to improve software quality by exploiting CSEs. We present three example software development tasks that can be assisted by our life-cycle model and show how these three tasks can contribute to improve the software quality. We also show the application of our life-cycle model with a preliminary evaluation.

1. Introduction

Open source code available on the Internet has become a common platform for sharing source code. Programmers often reuse the design of code examples or adapt code examples of existing open source projects rather than discovering usage patterns by digging into documents. Currently, the amount of open source code available on the Internet is enormous. For example, *sourceforge.net*[1], the world's most popular website for open source software development, hosts about 179,518 projects with two million registered users and a large number of anonymous users. With such enormous amount of open source code available on the Internet, several code search engines (CSE) such as Google code search [6], Krugle [7], Koders [1], Sourcerer [9], and Codase [5] are developed to efficiently search for relevant code examples (i.e., source files containing a search term). These CSEs accept queries such as the names of classes or methods of Application Programming Interfaces (API) and search in CVS or SVN repositories of available open source projects.

Although CSEs can serve as a means for searching in enormous amount of open source code, the usage of CSEs is often limited to simple tasks such as searching for relevant code examples. In this paper, we propose a life-cycle model[2] that combines code searching through CSEs and mining common patterns of API usages from gathered code examples. Our proposed model can be used to assist three main software development tasks: (1) to learn about an API usage by automatically inferring programming rules (from the mined patterns), (2) to use mined patterns to detect defects in a program under analysis, and (3) to infer a fix that needs to be applied for a detected defect.

There exist approaches [4, 8] that mine common usage patterns (e.g., frequent occurrences of pairs or sequences of API method calls) as programming rules for software verification or software reuse. One common characteristic in these existing approaches is that these approaches mine patterns from a few code bases. Therefore, these existing approaches often cannot surface out many programming rules as common patterns because there are often too few data points in these code bases to support the mining of desirable patterns [10]. In other words, the number of data points to support a pattern related to a particular programming rule is often insufficient. The drawback of these approaches is reflected in empirical results reported by these existing approaches: often a relatively small number of real programming rules were inferred from huge code bases.

A natural question to ask is whether a larger number of code bases (such as a large scale of open source code) can serve as an alternative data source for smaller code bases. One issue with a larger number of code bases is that mining a larger number of code bases is often not scalable. To address this issue, we propose a life-cycle model based on code searching and mining. In our life-cycle model, we expand the data scope to a larger number of code bases and include techniques to address scalability issues. In particular, we search for relevant code examples using CSEs and mine *only* those code examples. Our life-cycle model can assist in improving software quality over different phases of software development. We refer to our model as life-cycle model since the mined patterns can be used for writing new code, which can again be used as input for our model.

[1] `http://sourceforge.net/`

[2] The term life-cycle model is inspired from software development life cycle and refers to the life cycle of mining patterns.

We applied our life-cycle model in our previous approach, called PARSEWeb [10], that identifies frequent method-invocation sequences to serve as solutions for queries of the form "*Source object type* → *Destination object type*". In the evaluation of PARSEWeb, we show that our model can address issues that cannot be addressed by a CSE or any existing mining approach individually. We also show that code examples gathered from a CSE require post-processing before mining common usage patterns. In this paper, we elaborate on the life-cycle model of code searching and mining. We describe issues (and related post-processing techniques) that need to be addressed before using gathered code examples for mining patterns. We also present the application of our model in improving software quality with an emphasis on the post-processing techniques.

2. Life-Cycle Model

We next describe our life-cycle model that exploits CSEs and mines common patterns from gathered code examples. These common patterns can be used in improving software quality. Our model includes two phases: *searching* and *mining*. In the searching phase, we use CSEs to gather relevant code examples. In the mining phase, we analyze these code examples and mine patterns that describe how to use an API. Figure 1 presents an overview of our life-cycle model. We next describe each phase in detail.

2.1 Searching

The searching phase includes two tasks: *query construction* and *duplicate elimination*.

Query Construction. In the query construction task, we construct queries with API names as search terms. For example, we construct the query "`lang:java org.apache.regexp.RE`" to gather relevant code examples of the `RE` class from Google code search (GCS). These code examples show how to use the `RE` class provided by the Apache library [3]. GCS returns around $2,000$ code examples for this query. Based on our experience with CSEs, one observation with query construction is that the *relevance*[3] of resulting code examples mainly depends on the format of the query issued to CSEs. Without a well-formulated query, CSEs can result in a high number of irrelevant code examples. For example, to search for relevant code examples of the `fopen` API, a basic search query on GCS is "`lang:c fopen`". This query returns around $752,000$ code samples. When the query is tuned to "`lang:c file:.c$ [\ s \ *]fopen [\ s]?\(`" (GCS supports search with regular expressions), GCS returns $689,000$ code examples. Among the top 50 returned code examples, the number of relevant code examples was found to be doubled among code examples returned by a specific query (query with regular expressions) when compared to that of a basic query. The

[3] A code example is relevant when it includes a call site of the required API that is searched for.

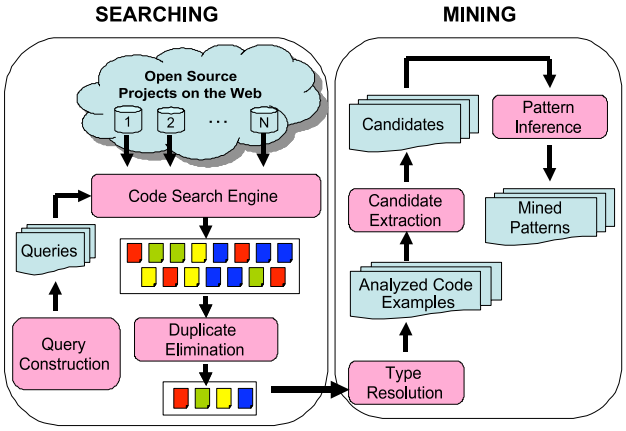


Figure 1. Phases in the life cycle of mining approaches based on code search engines

relevance (or quality) of gathered code examples plays an important role in mining common patterns from the gathered code examples. Although a programmer can decide to filter out irrelevant code examples during either the searching or mining phase, filtering out irrelevant code examples using an appropriate query in the searching phase can help reduce additional efforts in handling irrelevant code examples.

Duplicate Elimination. One observation with code examples returned by CSEs is that these code examples often include duplicate copies. We consider two code examples as duplicate of each other, if both belong to the same project and the same source file. For example, among the $2,000$ code examples returned by GCS for the query "`lang:java org.apache.regexp.RE`", the source file `JakartaRegexpRegexp.java` is found 13 times. Among these 13 copies, there are 5 different versions of the source file and the remaining 8 copies are duplicates of these 5 versions. There are both desirable and undesirable consequences with duplicate or multiple versions of source files among code examples. For example, code examples that are duplicate of the same source file, such as those belonging to a particular jar file, can be found to be used in various projects. The existence of duplicate or multiple copies for a code example can indicate that the code example is widely used and therefore the code example can be trusted more than those code examples that do not have duplicate or multiple versions. On the other hand, duplicate or multiple copies can bias the results of mining approaches that try to mine common patterns. To mine unbiased patterns used across a large number of code bases, we propose duplicate elimination to identify and filter out duplicate code examples.

2.2 Mining

The mining phase includes three tasks: *type resolution*, *candidate extraction*, and *pattern inference*. We refer to

```
01:import java.util.ArrayList;
02:import java.util.*;
03:Public class test {
04: public void method1(ArrayList list) {
05: Iterator iter = list.iterator();
06: while(iter.hasNext()) {
07:  String str = (String) iter.next();
08: ...}}
```

Figure 2. A code example using `Iterator` API.

these three tasks as *post-processing techniques* on gathered code examples.

Type Resolution. In the type resolution task, we resolve object types such as the return object type of a method call in gathered code examples. These object types are necessary for analyzing gathered code examples. In our model, we cannot use traditional techniques for parsing and resolving object types. The primary reason is that CSEs often return only individual source files (i.e., code examples) including the search term, and these code examples are often partial and not compilable. In our context, a partial code example indicates that the code example is complete; however, the other source files on which the code example is dependent upon are not available. To achieve the task of type resolution, we use partial program analysis for resolving object types. In our PARSEWeb approach [10], we developed 16 heuristics and these heuristics are contrary to type checking done by a compiler. We next present a sample heuristic for inferring fully qualified names using a code example (shown in Figure 2) to show the use of these heuristics in analyzing partial code examples.

Inferring fully qualified names. In Java, classes and interfaces have fully qualified names that can be extracted from the class declaration. However, as our gathered code examples are partial, we infer fully qualified names from `import` statements in these code examples. For example, the fully qualified name of the `ArrayList` class is inferred from the `import` statement in Line 1. However, this heuristic cannot infer the fully qualified name for the `Iterator` class referred in Line 5. The reason is that the related `import` statement in Line 2 uses `*` instead of the `Iterator` class. Our heuristics are not complete as these heuristics cannot resolve entire type information. However, the evaluation results of our PARSEWeb approach show that these heuristics are often effective in resolving required type information.

Candidate Extraction. In the candidate extraction task, we analyze gathered code examples to extract pattern candidates. These pattern candidates include information about API usage. For example, a pattern candidate extracted from the code example in Figure 2 is "`Iterator.next` should be preceded with a `boolean` check on `Iterator.hasNext`".

Pattern Inference. In the pattern inference task, we apply mining techniques such as frequent subsequence mining [2] on extracted pattern candidates to mine common patterns of API usage. The details of these two tasks (candidate extraction and pattern inference) vary across the types of problems being addressed using our model, whereas type resolution is an essential task to resolve type information in gathered code examples.

3. Application of the Life-Cycle Model

In this section, we describe example software development tasks that can be assisted by our life-cycle model and show the utility of our model with a preliminary evaluation done for one of these tasks.

3.1 Tasks Assisted by Life-Cycle Model

We expect that our life-cycle model can be used to improve software quality over different phases of software development by assisting three example major tasks.

Development. The patterns mined using our life-cycle model can be used to assist programmers during the development phase. These mined patterns provide common usage scenarios of how to reuse APIs and can be referred to as specifications while writing code. We implemented an approach, called PARSEWeb [10], based on our life-cycle model. PARSEWeb can be used to assist programmers during software development. The utility of the PARSEWeb approach over a traditional approach based on a CSE is shown in Section 3.2.

Verification. The patterns mined using our life-cycle model can be used to detect deviant behavior in a program under analysis. These patterns can be treated as specifications of an API usage and any deviation from the pattern in a program under analysis can be reported as a violation. For example, consider a pattern mined for the `fopen` API of standard C library (`stdio.h`) as shown below:

```
API method: fopen
Condition check on "return" value
Condition Type: NULL-CHECK
```

The preceding pattern describes that a majority of gathered code examples contain a `NULL` condition check on the return value of the `fopen` method call. This pattern can be used to detect defects related to missing condition checks after the `fopen` API call in the program under analysis. This example illustrates that our life-cycle model can be used to detect defects in the verification task.

Maintenance. The patterns mined using our life-cycle model can also be used for suggesting defect fixes during the maintenance task. For example, consider the following pattern related to the `Iterator.next` method:

```
API method: Iterator.next
Condition check on "return" value of
Iterator.hasNext
Condition Type: BOOLEAN-CHECK
```

The preceding pattern describes that there should be a `boolean` check on the `Iterator.hasNext` method before invoking the `Iterator.next` method. Failing to perform the boolean check can cause `NoSuchElementException`. Consider that the verification task detects a violation of the preceding pattern in a program under analysis. In this scenario, we can suggest a defect fix based on the pattern. For example, we can automatically perform defect fixing by inserting a `boolean` check on the `Iterator.hasNext` method before the `Iterator.next` method.

3.2 Preliminary Evaluation

We next show the utility of our life-cycle model over a traditional approach of directly searching via a CSE with an example task related to software development. We implemented our life-cycle model in our previous approach, called PARSEWeb [10]. PARSEWeb accepts queries of the form "*Source object type* → *Destination object type*" and finds method-invocation sequences that produce the destination object type from the source object type.

We use a programming problem "`org.eclipse.ui.IWorkbenchWindow` → `org.eclipse.ui.IViewPart`" described in a previous related approach [8]. The programming problem can be interpreted as that a programmer has an object of the `IWorkbenchWindow` class, and the programmer wants a method-invocation sequence to obtain an object of the `IViewPart` class. We use four code search engines (GCS, Koders, Krugle, and Codase) and PARSEWeb to investigate how they can assist in addressing this programming problem.

The minimal requirement for a code example to include a solution method-invocation sequence is that the code example should include both classes `IWorkbenchWindow` and `IViewPart`. Therefore, we constructed a query with both class names as search terms and used all four CSEs to gather relevant code examples. GCS, Krugle, Koders, and Codase returned 775, 112, 478, and 0 code examples, respectively. We inspected the top ten code examples returned by each CSE to check whether these code examples include a solution method-invocation sequence. We found that GCS and Koders include a solution method-invocation sequence in the sixth and eighth code examples, respectively. We could not find any solution method-invocation sequence among the code examples returned by Koders and Codase. There are two major tasks that need to be carried out in using CSEs directly for addressing this programming problem. First, the programmer has to browse to the sixth or eighth code example for getting a solution sequence. Second, the programmer does not have any knowledge whether this solution sequence is a commonly used method-invocation sequence. We next used PARSEWeb to recommend a solution for the programming problem. The solution sequence recommended by PARSEWeb is shown as below.

```
... IWorkbenchWindow iwwObj;
IWorkbenchPage iwpObj = iwwObj.getActivePage();
IViewPart ivpObj = iwpObj.findView(String);
```

PARSEWeb analyzed code examples gathered from GCS to generate sequence candidates. PARSEWeb used these sequence candidates to mine commonly used method-invocation sequence. PARSEWeb recommended a single solution sequence that is a common sequence among gathered code examples. This example shows the utility of our life-cycle model used to develop our PARSEWeb approach.

4. Conclusion

We proposed a life-cycle model that can be used to develop approaches based on code searching and mining. We elaborated on the two phases of our life-cycle model and suggested post-processing techniques for mining patterns from gathered code examples. We also presented three example software development tasks (that contribute to improve software quality) that can be assisted by our life-cycle model. Additionally, we highlighted the utility of our life-cycle model with an approach developed based on our life-cycle model.

Acknowledgments. This work is supported in part by NSF grant CCF- 0725190, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

References

[1] Koder's Zeitgeist. `http://www.koders.com/zeitgeist/`.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

[3] The Apache Jakarta Project, 2007. `http://jakarta.apache.org/regexp/`.

[4] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proc. ISSTA*, pages 163–173, 2007.

[5] Codease, 2005. `http://www.codase.com/`.

[6] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[7] V. Magotra. The art of ranking code search results, 2006. `http://blog.krugle.com/?p=184`.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. PLDI*, pages 48–61, 2005.

[9] B. Sushil, N. Trung, L. Erik, D. Yimeng, R. Paul, B. Pierre, and L. Cristina. Sourcerer: a search engine for open source code supporting structure-based search. In *Proc. OOPSLA Companion*, pages 681–682, 2006.

[10] S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.

Hybrid Storage for Enabling Fully-Featured Text Search and Fine-Grained Structural Search over Source Code

Oleksandr Panchenko
Hasso Plattner Institute for Software Systems Engineering
P.O. Box 900460, 14440 Potsdam, Germany
panchenko@hpi.uni-potsdam.de

Abstract

Searching is an important activity in software maintenance. Dedicated data structures have been used to support either textual or structural queries over source code. The goal of this ongoing research is to elaborate a hybrid data storage that enables simultaneous textual and structural search. The naive adjacency list method has been combined with the inverted index approach. The data model has been enhanced with the use of recent data compression approaches for column-oriented databases to allow no-loss albeit compact storage of fine-grained structural data. The graph indexing has enabled the proposed data model to expeditiously answer fine-grained structural queries. This paper describes the basics of the proposed approach and estimates its feasibility.

1 Introduction

Source code search is an essential activity in software maintenance. Approximately 30% of the actions performed on source code involve searching [9]. Depending on the task, developers perform different search strategies. Search queries include full-text queries, structural queries or a combination of the two [14]. Since both aspects are equally important, the combination of both types of data in one storage seems to be preferable. However, existing tools treat these two aspects independently: some methods focus on information retrieval (IR) techniques to enable full-text search on source code [12, 11], other tools rely on structural analysis methods [2, 7, 8]. Maintaining and integrating several data structures results in higher costs.

This paper proposes a data model that uses recent compression and graph indexing techniques to enable simultaneous full-text search, structural search or a combination of the two. Since queries are often formulated in terms of the application domain [4], a mapping between application domain terms and implementation domain terms should be supported. The proposed data model allows expeditious answering of various queries: text queries in application domain terms and implementation domain terms; structural queries on multiple levels of detail, e.g. find all writing accesses for a variable, find all classes with more then 30 methods or find a class with only static final fields.

2 Existing Approaches

The inverted index is a popular storage for textual and numerical information. Amongst many other scenarios this technique has been used to index source code [12]. Although IR methods have been successfully used in software engineering for a long time, structural information in source code remains mostly unutilized [10, 11]. Hash table is a naive technique to implement inverted index. Although some advanced compression and indexing methods based on b-trees or bitmap indexing have been developed for both numerical and categorical values [15], so far none of those methods have been applied to storing source code, and, especially, to its structural properties.

Holmes et al. used a relational database to store source code and to query it to recommend relevant examples [7]. Nevertheless, this approach assumes storing source code with a coarse level of detail.

Begel proposed enriching terms stored in an index with metadata extracted from the source code about the role of the term, type of usage and few others [3]. Similar approach is used by existing code search engines, e.g. Codase, Koders, Krugle, and Google Code Search. Here, additional metadata about the used programming language, license and other information is captured. However, stored metadata does not contain

important information about relations between entities and can be used to answer only relatively simple questions about source code.

Bajracharya et al. have used a simplified relational data model [2], which has been considered appropriate to store data with a middle level of detail. Information about more fine-grained characteristics is precomputed and aggregated into fingerprints. If new queries should be answered, new fingerprints should be introduced.

Many tools for storing and querying source code exist. However, to keep high performance and acceptable requirements for disk and memory space, the amount of information stored in the database is kept small. Thus only a few predefined queries can be answered based on coarse structural information or on precalculated and stored facts about source code. Storing fine-grained data using existing data models will significantly increase space requirements. Ad hoc queries result in low performance. Hence, a new type of storage should be developed. The data model proposed in this paper enables no-loss storing structural information and answering free style queries. The goal is to fit index of even large projects into main memory.

3 Hybrid Storage Structure

The maintainer needs several representations of source code: textual representation, abstract syntax tree (AST), call graph, data flow graph and different subgraphs of it, slices, etc. These representations are used for navigation as well for search. Although these representations are quite different in their nature, the elements of all the representations remain the same. Only the way these entities are connected and represented is changed. The proposed data model contains all representations in one index as shown in Figure 1.

Adjacency matrix and adjacency list are crude approaches for storing structural information represented as a tree or a graph. The adjacency list approach is a table, each row of which corresponds to an edge with references to the start vertex and to the end vertex. This simple data model is applicable on a limited basis for large structures because of non-optimal space usage. Moreover, storing related elements in one index results in multiple self-joins if it comes to traversing the relations. The proposed data model overcomes both challenges.

The first improvement is the utilization of compression algorithms. A row in the index for structural information can store different types of relations. The column RELATION_TYPE distinguishes between edges of different structures. Since all relation types refer to the same entities, the column ENTITY_ID refers to the dictionary where the entities are stored. Storing a higher amount of data in a column without increasing the cardinality increases opportunity for compression. High compression enables storing fine-grained information. In the proposed approach the dictionary encoding and run-length encoding [16, 13] are used. Thus, each dictionary value is encoded with a minimal possible number of bits. Nevertheless, depending on the actual distribution of values in the columns, other compression methods can be used [1]. An elaborate analysis is needed to identify the best compression algorithm for this scenario. More detailed information leads to a higher number of repeated entities in the index and to lower entropy. Low entropy results in a higher compression rate and in manageable memory consumption. Thus, a much higher degree of detail results in only a marginal increase of the required space.

The second improvement is the use of an indexing algorithm. To overcome the performance challenge caused by multiple self-joins, the pre- and post-order indexing method can be used. To each vertex in the tree, two numbers are assigned: pre-order number and post-order number. These numbers are ascertained during a depth-first traversal of the tree before and after the visit of the corresponding vertex. If vertex A is reachable from vertex B, A must have a higher pre-order and lower post-order number than B. Although the method was originally developed for trees, the GRIPP approach [17] is an extension for graphs. As graph vertices can have more than one ancestor, those vertices receive additional pre- and post-order numbers during indexing. The result of this transformation is a tree with additional vertices that receive independent pre- and post-orders but which are in fact just pointers to the original vertex. During query processing this leads to additional queries. However, the number of additional queries usually remains reasonable, depending on the graph density. The preliminary analysis shows that most vertices with more than one ancestor are leaves and do not require additional queries. To indicate this, a special field is included in the table.

Two dictionaries define the vocabulary of all entities extracted from source code. The application domain dictionary includes all automatically extracted keywords that describe the application domain. The implementation domain dictionary contains all tokens that form the program: class names, method and variable names, parameters, etc. To simplify the decision if the term belongs to the application domain, a dictionary lookup has been made: all English words were assigned to the application domain terms. The index for textual information is used to persist the many-to-many mapping between application domain and imple-

mentation domain terms. This paper discusses only the possibility of storing such a mapping. Readers interested in how tuples [application domain term, implementation domain term] can be identified are referred to the work of Cleary and Exton [5]. Fields DOCUMENT_ID and POSITION refer to the place in source code, where the term can be found.

The data is supposed to be stored in a central column-oriented main memory database. The column-oriented architecture is more suitable for the selected types of compression [1]. It is also possible to parallelize indexing and querying, to partition index horizontally based on projects, areas, versions, etc. and to distribute the index on several servers.

Although both dictionaries have similar structure and content these are separated into two tables due to maintenance ease. Industry size column-oriented databases such as TREX [13] can populate and administer the implementation domain dictionary automatically. The application domain dictionary is populated by a tool for mapping application and implementation domain terms.

4 Sizing Example

This section illustrates some important characteristics of source code in the context of the data model and determines the size of the index for a selected project. A prototype has been implemented that traverses source code, constructs ASTs and call graphs, analyses these, and stores the statistics in the index. One open source project has been selected (JAllInOne ERP System v.0.9.17) to exemplify how such an index can appear. Table 1 illustrates some facts about the project. In addition to the entities presented in Table 1, there are a number of other elements such as method parameters, numerical and string literals, exceptions, etc. The total number of implementation domain entities is 151 thousand. A number of terms were identified as application domain terms. Each entity appears at least once in ASTs. The number of all vertices in ASTs is 543 thousand.

To demonstrate how several different types of structural information can be coalesced in the index, a second type of structural information has been stored there, namely call graph. The number of edges in the call graph is 49 thousand. Each [vertex, ancestor vertex] relation as well as each [caller, callee] relation corresponds to one row in the index. Therefore, the index has 592 thousand rows.

Now, the amount of memory needed to store data is investigated. For more details see Table 2. The size of the index is 592,260 rows * 96 bits per row = 7,107,120 bytes. Some space can be saved by the run-length encoding. The complete infrastructure including the implementation domain terms dictionary and control files requires 11 MB. The source code (only *.java files, without configuration files, etc.) needs 8.4 MB disk space. Therefore, the proposed data model requires about the same amount of memory as the original source code, yet includes no-loss fine-grained structural information and the possibility of mapping between application domain terms and implementation domain terms. It is expected that including other representations, e.g. data flow graph, into the index will only marginally increase its size.

This estimation in no way represents a complete analysis. Nonetheless, these measurements have been helpful in a feasibility assessment of the proposed approach and in the determination of problem areas.

Table 1. Number of entities

Entity type	Number
TypeDeclaration	1,239
MethodDeclaration	8,508
FieldDeclaration	8,056
VariableDeclarationFragment	16,012

Table 2. Row size of the index

Column	Cardinality	Bits needed
PRE_ORDER	682,322	20
RELATION_TYPE	2	1
ENTITY_TYPE	83	7
ENTITY_ID	281,406	19
POST_ORDER	682,322	20
DOCUMENT_ID	1,089	11
POSITION	61,393	16
IS_VIRTUAL	2	1
IS_LEAF	2	1
Total		96

5 Conclusions and Further Work

This paper illustrates the possibility of creating a compact and fine-grained representation of source code in a main memory index, which enables high performance search. The structure for the mapping between application domain terms and implementation domain terms is directly incorporated into the data model.

Such an extensive indexing is time-consuming, therefore this data model can be primarily used in

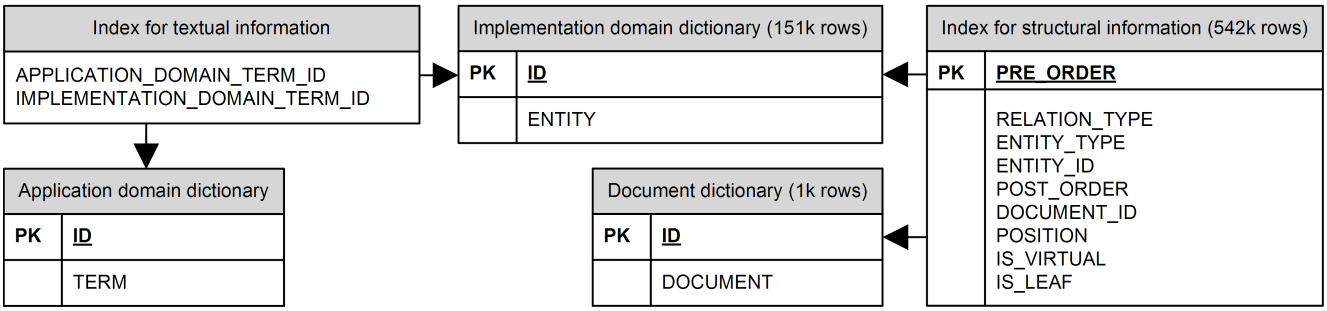


Figure 1. Data model

maintenance, where software is changed continually, but not intensively. An incremental indexing technique should be developed.

Since the output of the GRIPP method is in fact a tree, it seems reasonable to apply methods for querying XML [6] to source code structures. For example, one can provide an XPath-like interface to the repository.

A benchmark is planned to compare performance of the approach with other approaches. Moreover, a substantial experiment with maintainers is intended to measure usability of the tool, and to estimate performance of queries and the number of queries needed in different scenarios.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the international conference on Management of data*, pages 671–682, New York, NY, USA, 2006. ACM.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[3] A. Begel. Codifier: A programmer-centric search user interface. In *Proceedings of the Workshop on Human-Computer Interaction and Information Retrieval*, pages 23–24, 2007.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–82, 1994.

[5] B. Cleary and C. Exton. Assisting concept location in software comprehension. In *Psychology of Programming Workshop*, pages 42–55, 2007.

[6] T. Grust. Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD Int-l Conference on Management of Data*, pages 109–120. ACM, 2002.

[7] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[8] O. Hummel and C. Atkinson. Using the web as a reuse repository. In *Proceedings of the International Conference on Software Reuse*, pages 298–311, 2006.

[9] T. Lethbridge and J. Singer. Studies of the work practices of software engineers. In *Advances in Software Engineering: Comprehension, Evaluation, and Evolution, H. Erdogmus and O. Tanir (Eds)*, pages 53–76. Springer-Verlag, 2001.

[10] D. Liu and S. Xu. Challenges of using LSI for concept location. In *Proceedings of the 45th annual southeast regional conference*, pages 449–454. ACM, 2007.

[11] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223. IEEE Computer Society, 2004.

[12] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with Google. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 334–338. IEEE Computer Society, 2006.

[13] J. Schaffner, A. Bog, J. Krüger, and A. Zeier. A hybrid row-column OLTP database architecture for operational reporting. In *Proceedings of the international workshop on Business Intelligence for the Real Time Enterprise*, 2008.

[14] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 180–187. IEEE Computer Society, 1998.

[15] K. Stockinger, J. Cieslewicz, K. Wu, D. Rotem, and A. Shoshani. Using bitmap index for joint queries on structured and text data. *Annals of Information Systems*, pages 1–23, 2008.

[16] F. Transier and P. Sanders. Compressed inverted indexes for in-memory search engines. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, 2008.

[17] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 845–856. ACM, 2007.

Specifying What to Search For

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI. 02912 USA
spr@cs.brown.edu

Abstract

In this position paper we look at the problem of letting the programmer specify what they want to search for. We discuss current approaches and their problems. We propose a semantics-based approach and describe the steps we have taken and the many open questions remaining.

1. Motivation

One of the first things a programmer should do when writing new code is to find existing, working code with the same functionality, and reuse as much of that code as possible. With the large amount of open-source code available and the fact that most applications are not completely novel, one could imagine that a significant amount of the code that is being written today has been written before in some form, and much of it is available in an open-source repository.

This is the type of motivation given for code search. The emphasis here is on enabling reuse, avoiding writing what has been written before, making effective use of open source software, speeding up development, and producing higher quality software systems.

Code search, if it is going to achieve these ends, has to be substantially different from traditional web search. In particular it has to be designed and implemented so that:

- It is easier to use the results returned from the search engine rather than creating the code from scratch. Reuse should be relatively simple.
- The results returned must do what the programmer wants them to do. They shouldn't be an approximation or just related.
- The programmer must be able to use the resultant code. This means that the code must conform to **all** the requirements for the potential application.
- Programmers must be able to find what they are looking for. It is much trickier to determine this by looking at the summary or even the code itself than it is for textual information.

Achieving these goals should be the aim of the code search community. This involves addressing several problems. The first is getting access to all the appropriate open source (and other) code. The second involves organizing this data set and providing an appropriate query mechanism. The last is to provide the appropriate user interface for code search.

The first two of these problems, while difficult, have proven tractable, as can be seen in the various existing solutions. The difficulties lie in the fact that most code is not designed primarily for human readability and other factors such as program structure can be important aspects of the search. The last problem however, attempting to make the code search interface meet the programmer's needs, is the one that I find the most interesting and the least tractable.

2. Current Solutions

Current interfaces to code search take three principle forms. The first and most prevalent approach utilizes keywords. Here one depends on matching the user's vocabulary with that of the original programmer. It also requires finding keywords that are unique enough to actually identify the code in mind. Keyword searches typically yield lots of unevaluated results. The problem with this is that programmers have to read each instance of returned code, attempt to understand what it does, and then determine if it meets their requirements.

My experience with keyword based code search is that it creates a lot of work for the programmer. For any simple piece of code, the effort required to analyze the returned results is more than the effort that would have been required to write the code in the first place. Also, it can be a significant amount of work converting the code into the programmer's target framework after finding an appropriate method or class. This makes reuse based on code search difficult and unappealing.

The second approach is to use information about program structure, for example method signatures or expected loop structures [1-3]. These can be combined with keyword search. While the additional information can be helpful, its applicability is relatively limited. When searching for a method, the program structure

used in the algorithm is generally unknown or irrelevant to the search. Moreover signatures are only relevant if the code base being searched and the user's code share the same environment or if the programmer happens to be looking for a code snippet that is independent of their coding environment, a rarity in practice.

The third approach used today is to let the programmer specify test cases possibly accompanied by a set of keywords. When keywords are not provided, the test cases themselves can be used to derive keywords.

This approach is closer to what is needed for effective code search. Here the programmer is attempting to define what they are looking for, and define it in a way that the system might understand. Test cases can go beyond simple functionality and check additional code properties such as security and error handling. They also can define the context in which the code has to work.

There are several problems here. The first is that test cases can be difficult to write. In many cases the amount of code required for testing can exceed the amount of code being retrieved. When part of test-driven development, this might be an acceptable cost, but otherwise it can be a deterrent [4]. The second problem is that the problem might be one where test cases are difficult to define because the solution is not well defined. An example I've run across here is a method that determines what countries a news article is about, returning a probability vector where there is no real "correct" answer. The third problem is that, as the test cases and methods become more specific and constrain the set of possible solutions, the likelihood of finding any code that does exactly what the programmer wants becomes diminishingly small.

3. Examples

To make this analysis more concrete, consider some examples of code search I have attempted.

My first case was relatively simple. I needed to get the text from an HTML page using a Java class or method. The difficulty was that in my application white space in the text was relevant and I needed to understand one type of tag. Moreover, I wasn't particularly fussy about an interface; I could work equally well with a parser that works with callbacks or a parser that generated a tree of nodes.

Searching for the keywords "html parser": tends to yield lots of results, most of which are not relevant. Even when I found an implementation of a HTML parser this way, I still had to determine if that parser collapsed white space or not. Because the parsers are relatively sophisticated components, this involved significant work. Often, it seemed easier to do this dynamically, running the parser on a sample file and seeing what it returned. However, this required writing a framework that instantiated and used the parser first, and the framework was essentially different for each of the available parsers. This relates to the problem of using test cases in this situation; because the parsers have different interfaces, selecting a single test harness will essentially limit or eliminate what might be viable parsers from being considered. Once I select a particular callback framework or DOM model, it is very unlikely that I will find more than one parser and that parser is unlikely to do what I need with spaces.

In another example I needed to compute a topographical ordering for the set of nodes in a graph. The problem here is that I have my own graph structure. Code search finds lots of code that does topological sort. The results fall basically into two categories. First are those that are embedded in a graph class (or actually a whole hierarchy of graph classes). Of course the graph class here is significantly different than mine, and it looks like attempting to reuse the code is going to be more work than writing topological sort from scratch. Second are those that take sets of nodes and edges as arguments. Here, the problem was that the sets contain objects representing nodes or edges and these objects were quite different from the objects in my graph model. A further complication arose in understanding the behavior of the different algorithms when the graph was cyclic and there was no topological ordering.

Finally, I needed code that would find a least squares solution to a system of linear equations. Here code search helped to identify a relevant library that could be used almost directly. The real problem arose when we noted that we needed to add constraints so that all the resultant values were non-negative. Searching for this was not productive until it was pointed out that this was an instance of a quadratic programming problem. Searching for quadratic programming yielded solutions, but none were close to the desired specifications. To use them I had to write significant code that would convert the system of linear equations into a corresponding quadratic, set up the constraints, and then map the resultant values into the actual solution.

These examples only touch the surface of what programmers face when they attempt to use code search to facilitate reuse. In each case the programmer had a good idea of what he was searching for, however it was either difficult to specify or the results required significant work in order to be usable or both. Before code search becomes usable, these problems will need to be addressed.

4. Semantics-Based Search

In order for code search to be effective, the programmer needs to be able to specify what they are interested in finding. They need to state what they want the identified code to do functionally, where it has to fit in, and what constraints (e.g. performance, error han-

dling, security, privacy, synchronization) they want to impose on it.

This information is precisely a description of the semantics of the code where semantics is taken in a broad sense. It might include a formal description of the behavior of the code (formal semantics), a high level description of the code in terms of keywords or text (informal semantics), pseudo code for the function (structural semantics), test cases (semi-formal semantics), security and privacy limitations, error handling (formal or informally), recovery information, performance requirements, synchronization requirements, the context where the code will be used, and the desired coding style and conventions.

The programmer generally knows a subset of this information and needs to convey it to the code search tool in some manner. The goal of a code search interface should then be to encourage the specification of as much of this information as possible.

But even if the programmer could be precise here, it would not be enough. As programmers become more precise as to what they want, the odds of identifying code that exactly matches their specifications approaches zero. Moreover, many of the problems, for example signatures and use of the programmer's environment or context, need to be addressed in order to correctly interpret other parts of the specifications such as test cases.

Thus, an effective code search tool has to automate much of the way that the programmer would want to use the result of the search. In particular it has to automatically refactor the located code to fit the programmer's environment. Examples of the transformations or refactorings that might be done here include: adapting the signature of the identified code to the programmer's specification, eliminating unnecessary functionality, isolating the desired functionality from the middle of an existing routine, bringing in additional classes and methods that are required to make the code functional, modifying the code to use existing support classes rather than their own, converting the coding style to meet the current project's requirements, adapting the code to fit into an existing class, converting a class-based implementation into a method-based one (or vice versa), generalizing or specializing parameter and return types, changing the way errors are reported, and adding or removing logging or debugging statements.

5. The S^6 Project

The goal of our research is to create an appropriate front end for code search that allows semantic specification of what to search for and automatically performs the appropriate transformations that programmers would otherwise have to do to make the identified code usable in their application.

Our approach to date shows that this might be an achievable goal, but that significant work still needs to be done [5]. Our front end concentrates on allowing the user to specify test cases quickly and effectively. It requires keywords as a starting point for the search. In addition, it allows contracts to be defined for each method and the code to be run in a restricted Java security context. A front end to the system is available at http://conifer.cs.brown.edu/s6. Source code for the system is available from the author.

While this is a start, much more needs to be done. Keywords are sometimes hard to find or select. Complex test cases, for example test cases that require the user to write code, are not supported by the front end. The Java security model is limited and does little about privacy concerns. Contracts are only checked when running test cases, not statically against the code. Performance can only be evaluated in terms of the performance on the test cases (which are generally too simple for this purpose), and then only as part of sorting the output. No information about the structure of the target code is used, nor is anything done about synchronization. There is only limited support currently for handling user context such as existing classes and methods.

Our code search engine also applies a suite of transformations to attempt to adapt the code to meet the programmers' requirements. While this is necessary to accommodate test cases, and has been quite effective in simpler cases, much still needs to be done. The current transformations do not take into account the target environment or attempt to map the implicit environment of the identified code to the target environment. This is where most of the work of adapting identified code actually occurs. In addition, it only has a limited set of type mappings and does not support many class-level transformations. While much of this is easy to add in theory, in practice it is difficult because of the exponential number of possible mappings that can arise.

6. Challenges

Based on our experiences, we can identify several challenges that need to be met before code search will be really practical, challenges that we hope will be taken up by the code search community.

The first is finding a practical approach to letting the programmer specify the semantics of what they are looking for. This approach has to handle of the complexities of real world problems and situations. The approach will have to be an amalgam, since no one specification method or technique will work for all (or even a majority of) cases.

The second is developing the underlying code representations and search mechanisms that will support such a front end. Keywords alone are not sufficient. Signatures or program structures are helpful, but have to be flexible enough to accommodate the possible transformations. One should be able to search not only

based on the target code, but also on the desired and original contexts.

The third is finding a suitably broad set of transformations that mimic what programmers do when they adapt the result of code search to fit their applications, and then automating these transformations in a practical way. The main problem here is making this process intelligent, avoiding the potentially exponential number of results, and integrating transforms with the back end and the semantic specifications.

7. Acknowledgements

This work is supported by the National Science Foundation through grant CCR0613162.

8. References

1. Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," *Proc. OOPSLA 2006*, pp. 682-682 (October 2006).

2. Andrew Begel, "Codifier: a programmer-centric search user interface," *Workshop on Human-Coputer Interaction and Information Retrieval*, (October 2007).

3. Raphael Hoffmann and James Fogarty, "Assieme: finding and leveraging implicit references in a web search interface for programmers," *Proc. UIST 2007*, pp. 13-22 (October 2007).

4. Otavio Lemos, Sushil Bajracharya, Joel Ossher, Ricardo Morla, Paulo Masiero, Pierre Baldi, and Cristina Lopes, "CodeGenie: using test-cases to search and reuse source code," *ASE '07*, pp. 525-526 (November 2007).

5. Steven P. Reiss, "Semantics-based code search," *ICSE 2009*, (May 2009).

On the Evaluation of Recommender Systems with Recorded Interactions

Romain Robbes
REVEAL
Faculty of Informatics
University of Lugano
romain.robbes@lu.unisi.ch

Abstract

Recommender systems are Integrated Development Environment (IDE) extensions which assist developers in the task of coding. However, since they assist specific aspects of the general activity of programming, their impact is hard to assess. In previous work, we used with success an evaluation strategy using automated benchmarks to automatically and precisely evaluate several recommender systems, based on recording *and* replaying *developer interactions. In this paper, we highlight the challenges we expect to encounter while applying this approach to other recommender systems.*

1. Introduction

Developing software systems is a complex and difficult task relying on a large skill set, including program comprehension, creative thinking, problem solving and algorithmic skills. To assist developers as they program, Zeller envisions the future IDE as featuring a set of recommendation and assistance systems, each focusing on a type of commonly recurring problem [12]. We refer from now on to this type of tools simply as recommenders. Zeller's vision is already partially fulfilled, as state-of-the-art IDEs such as Eclipse already feature several recommenders, such as:

- Code completion, which assists the the seemingly simple task of typing program statements;
- Error correction, which, based on compilation warning and errors, proposes automated edit operations to address common classes of errors, such as importing missing namespaces (Eclipse's Quickfix);
- Change prediction, exemplified in tools such as eROSE, recommends entities to change alongside currently changed entities, based on the history of previous changes to the system [13];
- Navigation aids, such as Mylyn [4] or NavTracks [10], which based on what the programmer is currently looking at, recommend other entities to look at.

If all these recommenders are intuitively useful, having more definitive proof is difficult. The most common evaluation strategy that comes to mind is to perform a user study. A simple experimental protocol is to gather two groups of people, and ask them to perform a given development task, one group with the help of the recommender system, the other without. Each subject is either observed while they perform the task by the experimenter, or asked to fill a questionnaire after completing the task. From this data, the improvement that the recommender yields can be quantified. Many variants of this design exist, but they all share the following shortcomings:

Many variables: Each developer has a distinct experience with programming languages, tools, and a different way to solve problems. Some might type much faster than other. In short, there are many variables that could explain an observed variation in productivity. A larger number of subjects is needed to smooth out individual differences.

Subjectivity: Since coding relies on an array of skills, the developer or the observer themselves may have trouble discerning the impact of the recommender. How can a developer evaluate the accuracy of a code completion engine when he is focusing on finishing a non-trivial development task? Answering a questionnaire afterwards may hence yield vague or subjective answers that are hard to interpret.

Expensive: Recruiting a sufficient amount of people and carefully laying out the experimental protocol is time-consuming and potentially expensive. "Dry runs" of the experiment are necessary to weed out mistakes in the protocol. Finding the subjects for the experiment is also a difficult task as people value their time.

Hard to reproduce: User studies must have a very detailed protocol in order to be repeated. Lung *et al.* documented the hurdles they went through when they attempted to reproduce an experiment [5]. Reproducing user studies is hence hard and uncommon.

Of course, a user study is essential to ensure the good usability of a recommender, but these shortcomings mean that incremental improvements to a recommender are hard to evaluate this way. Indeed, the smaller the increment in productivity is, the larger the group of users need to be in order to rule out statistical error. The difficulty in reproducing a study and to compare the results of two studies is a further impediment to gradual optimizations. Such an optimization of the recommender is however essential to ensure that it is as accurate and useful to the developers as it could be. Without it, the algorithms and heuristics used by the recommender may be far from optimal.

There is however an evaluation strategy that is better suited to the comparative evaluation that is needed to optimize the algorithms at the heart of recommenders: Automated benchmarks [9]. An automated benchmark is a fully automatic process that takes as parameters a recommendation algorithm to evaluate and the data to evaluate it on, and computes the accuracy of the algorithm. This allows easy reproducibility and comparison between variants of the algorithm, making the technique ideal to optimize a recommender system.

In order to evaluate recommender systems in this way, the challenge lies in gathering the data necessary for the evaluation. In previous work, we introduced Change-Based Software Evolution (CBSE), which models with accuracy how software systems evolve over time [6]. CBSE relies in recording the changes as they happen in the IDE. Based on the change data we gathered on several systems, we applied a benchmarking strategy in order to evaluate several variants of two recommender systems, Code Completion and Change Prediction. The focus of this paper is hence to draw from this experience and outline the challenges that lies ahead in order to generalize this approach to other recommenders.

2 Benchmarking Recommenders

The approach we propose is based on the openness of state-of-the-art IDEs. IDEs such as Eclipse, Squeak or Visualworks allow one to easily monitor and record how the developer is using the IDE to incrementally develop a piece of software. Such an IDE issues all kinds of events (necessary for its internal architecture) in response to actions the developer perform. Examples of such events are navigation events indicating that the developer is looking at a given part of the system, tool usage event indicating that the developer is using the refactoring engine, the compiler or the debugger, and edition events describing how the developer changes the system, from keystroke events to higher-level events such as addition of entities.

If the IDE is open enough and allows access to these events to third-party extensions, one can *record* these events, and, ensuring that they are descriptive enough, *replay* them at will in an automated manner. If the information is accurate enough, such an approach allows one to effectively simulate –in an entirely automatic way– the interactions of the developer with the IDE as he is building the system.

Automation is key to allow the definition of interaction benchmarks as a methodology to repeatedly and accurately measure the accuracy of recommender systems. It allows one to inexpensively run several variants of the same recommendation algorithm and compare their performance with a precisely and objectively computed metric. This allows one to truly optimize the recommendation algorithm and make the recommender as accurate as possible.

Assuming that a recorded interaction history *H* is available, computing the accuracy *A* of a recommender *R* on the interaction history proceeds as shown by Algorithm 1 (*E* is the model of the system and the developer that the recommender uses).

Input: H, R
Output: A
foreach *Interaction* I *in* H **do**
  if I *is of interest to* R **then**
    Ask R to predict I, given the environment E
    Compare R to I and update A
  end
  Replay I in order to update E
end

Algorithm 1: Computing the accuracy of a recommender on an interaction history

Of course, Algorithm 1 assumes that the interaction history *H* exists. For this to be the case, one must follow the following steps:

Frame the problem of the evaluation of the recommender in terms that allow the automation. This implies isolating only the parts of the recommender that are relevant to the task, such as the central algorithm providing recommendations from unnecessary parts like the GUI.

Identify the information needed by the recommender to function properly, and from it, the interactions that need to be recorded to rebuild the information needed by the recommender.

Define the prediction format that the recommender uses and the kind of interaction triggering its evaluation.

Define the accuracy measure that will be computed. Depending on the types and format of the predictions, different measure will work, such as a single accuracy measure or both precision and recall.

Record interactions. Once the type of information needed is defined, one has to gather it by monitoring the activity of a large enough set of developers as they work for a long enough period of time, so that the set of interaction histories gathered can be deemed representative.

We now illustrate this process with the two examples in which we applied it successfully. In both cases, we reproduced and introduced several variants of each recommenders, and improved on the state of the art.

Example: Code Completion We used recorded interactions to measure the accuracy of several code completion engines [7]. Code completion is a recommender used to assist typing that presents to the user a list of words or function names that may correspond to the word the programmer is presently typing, saving her keystrokes.

To automate testing, we consider the completion engine only, that is the part of the recommender taking as input the prefix of a word, and returning a list of candidates matches. Since variants of the code completion engine rely on the state of the program and the recent changes to the system, the interactions we recorded were the changes made to build the system. Of these, the interactions of interest to the recommender were the changes that inserted new method calls in the system: The completion engine was asked to return a list of candidates which ideally contained the name of the method being inserted. The list was cut off at ten items, as a longer list of candidates was deemed too long to be read in its entirety by the programmer. To compute the accuracy, we measured the index of the correct match in the list, and rewarded correct matches that were in the first items, and for shorter prefixes (we tested each insertion of a method call by asking for the recommender's guess with a prefix of 2, 3 ... up to 8 letters). The data used in the benchmark were the recorded changes performed on 8 small to medium scale systems, totalling more than 3 years of development.

Example: Change Prediction Our evaluation of change prediction approaches [8] was very similar as it relied on the same recorded interactions, the changes to the system as they evolve. Change prediction attempts to predict the entities (classes or methods) that the programmer is going to change after the one he changes, in order to either remind him to change them (error prevention) or to provide easy access to them (navigation assistance).

The portion of the change predictor we tested was the algorithm that, given entities changed in the past, proposed a list of potential change-prone entities. The data recorded was the change sequence developers made when building programs. The entities of interest for which the change predictors was tested where the sequence of changed entities, filtering out repetitive changes to the same entities and changes originating from automated tools. The accuracy of the change predictor was defined as the similarity between a list of n change-prone entities returned by the predictor with the actual n next changing entities. The data set we used was very similar (it contained one additional change history from a Java program).

Note that a similar evaluation strategy was used by previous change prediction approaches, but based on SCM transactions, rather than recorded change sequences [3][13][11][2]. Using SCM transactions instead of recorded IDE interactions is more convenient, since a lot of data is already available, but less accurate, since the data is more coarse.

3 Challenges in Further Applications

We believe this approach is applicable to other kinds of recommenders and we expect the same benefits from its application. However, certain particularities of the approach mean that care must be taken in fulfilling the steps we described above. Indeed, recording the data is a costly and time-consuming operation. Hence the kind of data that has to be recorded must be carefully defined upfront. In the following we make a tentative list of recommender systems that we envision being evaluated in the same way, and highlight the needs for each of them.

Code Completion and Change Prediction: In our comparison of approaches, missing data prevented us to reproduce every approach we wished. The navigation information necessary for some approaches was missing. Further, a precise notion of task (*i. e.* the set of entities related to a task) was only approximated. This missing data is needed to further improve our results.

Task Detection: The missing notion of task context could be a recommendation by itself. We would like to annotate our change information with task information and experiment with several approaches to detect them.

Clone Detection: The presence of duplicated code in code bases is an established fact, and several tools exist to detect it. Based on our recorded change histories, we could annotate changes that introduce new clones in the system as interactions of interest.

Clone Evolution: A possible solution to the clone problem is to co-evolve clones when one of them changes [1]. The techniques proposed so far are based on simple

string rewriting. An automated benchmark comparing the actual changes with the future changes could assess whether more complex techniques are needed.

Error prevention and correction: With the necessary annotations of the change data, tools such as Quickfix could be formally evaluated, and new heuristics fulfilling their shortcomings could be defined.

Based on the potential applications, we identified the following issues that are open to discussion:

Additional sources of information. An effort is needed to identify all the necessary sources of information to be recorded, beyond those that we already identified, changes and navigation information. Tool support should then be implemented to record this data.

Annotations of the interactions. Annotations are needed to mark the entities of interest for each recommender, such as clones, tasks, errors and the interactions causing and/or solving them. A systematic review of the recorded interaction is needed to annotate them, and tool support is needed to perform it efficiently. Such a review would also filter out interactions featuring unwanted behavior, such as cases where the developer was in the wrong track for a part of the session.

Recording more data. The amount of data we recorded so far is still small, and some of it is incomplete. To provide more significant result, an effort is needed to record much larger interaction histories.

Community involvement. Recording a large amount of data, implementing the necessary tools, and improving on the state of the arts of recommenders once the infrastructure is there is a significant effort for which we welcome members of the community. In particular, a shared effort to record development histories of student projects would be the most immediate way to gather a larger amount of data.

4 Conclusion

Recommenders are a growing part of a programmer's tool set, yet optimizing them remains a difficult problem. We presented a general approach to evaluate the performance of recommenders in a systematic way, allowing incremental optimization of a recommender's overall usefulness to developers. The approach is based on the record and replay of programmer interaction histories in order to repeatedly simulate the activity of a developer. We outlined some of the challenges that we need to overcome in order to adapt the approach to various kinds of recommenders, namely identifying the kind of information one needs to record, recording of a large and representative enough set of interaction histories, annotating the interaction history in order to emphasize the relevant interactions when needed, and the development of a common infrastructure allowing the sharing of the data and the easy dissemination of the results necessary to foster a community around recommenders [9].

References

[1] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 158–167, 2007.

[2] T. Girba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society.

[3] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3):335–367, 2006.

[4] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of SIGSOFT FSE 2006*, pages 1–11, 2006.

[5] J. Lung, J. Aranda, S. M. Easterbrook, and G. V. Wilson. On the difficulty of replicating human subjects studies in software engineering. In Robby, editor, *ICSE*, pages 191–200. ACM, 2008.

[6] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, 2008.

[7] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd ACM/IEEE International Conference on Automated Software Engineering)*, pages 317–326. ACM Press, 2008.

[8] R. Robbes, M. Lanza, and D. Pollet. A benchmark for change prediction. Technical Report 06, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, October 2008.

[9] S. E. Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE*, pages 74–83. IEEE Computer Society, 2003.

[10] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005)*, pages 325–335. IEEE Computer Society, sep 2005.

[11] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.

[12] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In *Proceedings of the 2nd Future of Software Engineering Conference (FOSE 2007)*, pages 316–325, 2007.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572. IEEE Computer Society, 2004.

Internet-Scale Code Search

Rosalva E. Gallardo-Valencia
University of California, Irvine
rgallard@ics.uci.edu

Susan Elliott Sim
University of California, Irvine
ses@ics.uci.edu

Abstract

Internet-Scale Code Search is the problem of finding source on the Internet. Developers are typically searching for code to reuse as-is on a project or as a reference example. This phenomenon has emerged due to the increasing availability and quality of open source and resources on the web. Solutions to this problem will involve more than the simple application of information retrieval techniques or a scaling-up of tools for code search. Instead, new, purpose-built solutions are needed that draw on results from these areas, as well as program comprehension and software reuse.

1. Introduction

Open source is the practice of distributing source code along with the executable code for a computer program. The increasing availability of high quality open source code on the Internet is changing the way software is being developed [9]. It has became commonplace to search the Internet for source code in the course of a software development project.

Developers are increasingly using an Opportunistic Software Systems Development (OSSD) approach to put together software pieces that they found. This approach is used to face the market demands of delivering software quickly and with more functionality [6]. Although, these software pieces provide functionality that programmers need to include in a system, often, they are unrelated and were not designed to work jointly.

Developers who are using these approaches search the Internet looking for open source to reuse in their projects. We will refer to this specific type of source code search as Internet-Scale Code Search. Locating the right component for as-is reuse or a reference example at the right time can have significant impact on how the project progresses.

The Internet-Scale Code Search has many similarities with other areas of research and we should build on their contributions. These areas include software reuse, code search, information retrieval, and program comprehension. We will discuss the similarities and differences with these areas.

In this paper, we argue that Internet-Scale Code Search is a new kind of problem. Not only is Internet-Scale Code Search more than the sum of the parts, different kinds of technological possibilities are available due to emerging computational practices.

2. What is Internet-Scale Code Search?

Internet-Scale Code Search is searching the Internet for source code to help solve a software development problem. Results from a web-based survey have shown that developers search for code on the Internet with the motivation of finding a piece of software to reuse or a reference example to use as a guide. The target piece of software varies on size ranging from a block (a few lines of code), a subsystem, and a system [11]. Some examples of search targets from a previous empirical study are summarized below.

Table 1. Examples of software pieces classified by motivation and target size.

	As-Is Reuse	**Reference Example**
Block	Code snippets, wrappers, parsers	To learn language syntax and idioms
Sub system	Algorithms, data structures, GUI widgets, libraries	To help in the implementation of algorithms, data structures, GUI widgets. To aid in the use of libraries
System	Stand-alone tools, ERP packages, DBMS	To get ideas about an existing similar system

In the cases where developers are searching for a component for as-is reuse, the search parameters are more tightly defined, and developers are most often looking for a piece of functionality, that is, a portion of

code that will perform a particular task. This type of search target was also evident in the studies by Chen et al. [1], Madanmohan and De' [4]. Developers preferred components that could be used as-is, with little or no modification. In fact, they avoided components that required an understanding of the inner workings.

Developers are also motivated to search for a reference example of how to use or do something. In other words, software developers are using the web as a giant desk reference manual. While this kind of knowledge reuse has been acknowledged in the software reuse literature, it has been overshadowed by as-is component reuse. Searches for reference examples are qualitatively different from those for reusable components. The underlying problem being solved is different and so too are the selection criteria.

Some tools that support Internet-Scale Code Search are available on the Web. These tools include Google Code Search[1], Koders[2], Krugle[3], and Sourcerer[4] among others. Although these tools already help developers to find open source, a better understanding of the challenges behind code search on the Internet can suggest improvements to these tools.

2.1. Motivating examples

Here, we present some motivating examples of Internet-Scale Code Searching. These are composite descriptions based on data collected in our earlier empirical study [11].

Waldo was writing a Java program to send out meeting notifications by email. His program needed to send notifications of meetings in participants' local time zones. To do this, he needed to use the Calendar classes, which have a complex interface and can be used in many different ways. Rather than reading the Javadocs, he searched the web for examples of how the classes were used. Waldo found a number of useful blog posts and tutorials that gave him the information that he was looking for.

The example above describes a developer looking for a reference example for a subsystem, i.e. the Calendar classes in Java. In such cases, general-purpose search engines, such as Google and Yahoo, do reasonably well. The code that Waldo found was surrounded by natural language explanations that matched his search keywords. It should also be noted that he was not looking for a program element or identifier.

Wenda was looking for an implementation of the Trie tree data structure in C. A Trie tree is an ordered tree data structure where the keys are strings. She started out using a general-purpose search engine, but got too many matches. She added "C" as a search term to reduce the number of matches, but this did not help at all. She tried some code-specific search engines, but 'c' appeared a lot, so she switched to filtering by programming language. Also, "trie" was a substring of retrieval, so these too were a false start. Wenda ultimately found what she needed by going to a site were developers share ideas and resources with each other, such as www.codeproject.com. Here, she found a number of annotated examples that she could reuse as-is in her project.

This second example depicts a developer looking for a subsystem-sized reusable component. She wanted source code that implemented a well-understood abstract data structure. The main goal behind these searches is that the source code is commonly available and saves time. This example illustrates ways in which both general-purpose and code-specific search engines fall short. "C" was not a good search term, because it is too short and too common. "Trie" and "tree" were not much better. Also, it was difficult to judge the suitability of the various matches returned for Wenda's project. While some of the code returned by the code search engines had good comments, they generally lacked instructions for (re)use. As well, extracting the code and incorporating into her project would have required a non-trivial amount of work. In the face of uncertainty regarding the costs and benefits of adapting unfamiliar code, the safest option is often to implement it yourself.

3. Comparison with code search

Code search typically occurs within an Integrated Development Environment while working on the source code for a single project. This activity is often done during the development and maintenance of software, and involves searching for specific program elements in a software project. Developers have mainly four motivations to search for pieces of software: defect repair, code reuse, program understanding, and impact analysis. The pieces of software they are looking for are declaration, definition, use, and all uses of functions, variables, and classes [8].

Internet-Scale Code Search involves looking in not just one project but in a great number of different open source projects. In addition, Internet-Scale Code

[1] http://www.google.com/codesearch/

[2] http://www.koders.com/

[3] http://www.krugle.com/

[4] http://sourcerer.ics.uci.edu/sourcerer/search/index.jsp

Search will also search for source code in other types of information besides to source code repositories. It also includes searching on web pages, forums, mailing lists, and other sources.

Internet-Scale Code Search is an activity that is not restricted to the maintenance of software; this type of search expands to different phases in the software development process, such as feasibility study, analysis, design, implementation, testing, and maintenance.

Internet-Scale Code Search will build on the contributions from research into code search activity and scale it where possible to the Internet. However, we believe the usefulness of searching for program elements or certain kinds of identifiers will have limited applicability. More often, developers are looking for functionality or knowledge, and not for where a method or variable is declared.

4. Comparison with information retrieval

The area of information retrieval focuses on finding material of an unstructured nature, usually text, that satisfies an information need from within large collections stored on computers [5]. Internet-Scale Code Search is different because the material that developers are searching for is source code, which is structured in nature due to the fact that it follows strict syntax rules specific to a programming language.

Information retrieval commonly allows keyword-based searches. However, developers are searching for source code in terms of features, functionality, and requirements; source code is written in a programming language, while search keywords are in natural language. The only place where natural language appears in source code is within comments. Consequently, searches for code tends to rely on the comments and the text surrounding an excerpt of source code. Blocks of code on web pages have more descriptive text around them than in a version control repository. For these reasons, general-purpose search engines work surprisingly well in code search. Still, there is room for improvement, as our second motivating example illustrates.

The effectiveness of conventional information retrieval techniques can be attributed to both human ingenuity and metadata. Developers often find creative ways to make use of the tools available. For instance, they use general-purpose search engines to find repositories or caches where they can search further. Metadata, we believe, will play a major role in the design and implementation of improved Internet-Scale Code Search engines.

5. Comparison with software reuse

Software reuse is the process of finding and using existing components or libraries in the creation of new software [3]. It is now common to create software by hacking, mashing, and gluing together existing open source code [2]. Although, software has not been built from scratch since function libraries were invented, the Internet, open source, and Opportunistic Software System Development have significantly increased the scope and scale of source code being reused.

The Internet-Scale Code Search process differs from software reuse, where the recommended process is to identify the requirements and use them to evaluate the suitability of candidate libraries. Instead, the requirements are defined iteratively based on the available functionality. This process more resembles engineering design than looking for a set of lost keys. The former process proceeds by optimizing constraints in a cost effective manner. The latter process seeks to find an object that is known to exist, is well defined, and can only be located in a limited number of places. In other words, software developers acquire an understanding of what they are looking for by searching.

Software reuse contributes knowledge about the different facets of reuse, such as substance, scope, technique, and products [7]. Internet-Scale Code Search can use this knowledge when developers are looking for open source code on the Internet opportunistically.

6. Comparison with program comprehension

Program comprehension research has focused on the cognitive theories that help us understand how programmers comprehend software in a single body of source code and on the tools to aid users in their comprehension tasks [10]. A key step in the Internet-Scale Code Search process is the evaluation of thousands of candidate matches that have been returned by a search engine in order to find the right piece of open source code to incorporate into a project. This evaluation requires developers to understand the source code, but in contrast with program comprehension, this evaluation involves discerning the characteristics of a candidate piece of code without becoming entangled in the internals.

In conventional program comprehension, developers use source code and documentation to understand the program [10]. We believe that program comprehension in Internet-Scale Code Search is very different. Developers tend not to look at the source

code when selecting a component for reuse, but rather, they rely on surface features and external information sources. Preliminary research has identified two kinds of judgments. Relevance judgments are made by software developers while identifying promising candidates among the available matches. These decisions are rapid, taking only a few seconds, and use relatively little information. Suitability judgments are made when determining whether a promising candidate is appropriate for the software project. These decisions are slower and typically involve a careful cost-benefit analysis. These judgments are made based on characteristics of the open source project, fellow users, price, terms of license, documentation, and functionality.

7. Summary

In this paper, we argued that Internet-Scale Code Search is a novel problem, in need of novel solutions. Although, it is similar to a number of existing problems, research is needed to combine and create research contributions. This emerging field is similar to software reuse, source code searching, information retrieval, and program comprehension.

Internet-Scale Code Search is similar to software reuse, because developers are often looking for code to reuse as-is on their projects. But they also look for code to use as reference examples. Source code searching and Internet-Scale Code Search have in common the fact that developers are searching for source code. But, in the former developers are typically looking for program elements within a single project. In the latter, they look in a great number of open source projects on the Internet. Like information retrieval, Internet-Scale Code Search involves searching large collections. An important difference is that developers are searching for source code and not for natural language text in unstructured documents. Some program comprehension is needed during Internet-scale code search, but not the kind normally performed during software maintenance. During Internet-Scale Code Search, developers need to evaluate thousands of candidate matches using superficial information and from different sources.

In summary, Internet-Scale Code Search is a new problem that has arisen as a result of evolving technologies and software development practices. The solution to this problem will require similar innovation and creativity to both use existing results and to create know-how.

8. References

[1] Weibing Chen, Jingyue Li, Jianqiang Ma, Reidar Conradi, Junzhong Ji, and Chunnian Liu. “An empirical study of software development with open source components in the Chinese software industry.” *Software Process: Improvement and Practic*e, 13:89–100, January 2008.

[2] Bjorn Hartmann, Scott Doorley, and Scott R. Klemmer. “Hacking, mashing, gluing: A study of opportunistic design.” *Technical Report CSTR 2006-14*, Department of Computer Science, Stanford University, September 2006.

[3] C. W. Krueger. “Software Reuse.” *ACM Computing Surveys*, 24(2):131-184, June 1992.

[4] T.R. Madanmohan and Rahul De’. “Open source reuse in commercial firms.” *IEEE Software*, 21(6): 62–69, 2004.

[5] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[6] Cornelius Ncube, Patricia Oberndorf, Anatol W. Kark, "Opportunistic Software Systems Development: Making Systems from What's Available," *IEEE Software*, 25 (6): 38-41, Nov./Dec. 2008.

[7] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, New York, 1997.

[8] S. E. Sim, C. L. A. Clarke, and R. C. Holt. “Archetypal source code searches: A survey of software developers and maintainers.” *In Proceedings of the 6th International Workshop on Program Comprehension*, page 180, Los Alamitos, CA, 1998. IEEE Computer Society.

[9] Diomidis Spinellis and Clemens Szyperski. “Guest editors’ introduction: How is open source affecting software development?” *IEEE Software*, 21(1):28–33, 2004.

[10] Margaret-Anne D. Storey. “Theories, tools and research methods in program comprehension: past, present and future.” *Software Quality Journal*, 14(3):187–208, 2006.

[11] Medha Umarji, Susan Elliott Sim, and Cristina V. Lopes. “Archetypal internet-scale source code searching.” *In Barbara Russo*, editor, OSS, page 7, New York, NY, 2008. Springer.

Working with Search Results

Jamie Starke
University of Calgary
Calgary, Canada
jrstarke@ucalgary.ca

Chris Luce
University of Calgary
Calgary, Canada
cluce@ucalgary.ca

Jonathan Sillito
University of Calgary
Calgary, Canada
sillito@ucalgary.ca

Abstract

Source code search is an important activity for programmers working on a change task to a software system. We are at the early stages of a research program that is aiming to answer three research questions: (1) How effectively can programmers express (using today's tools) the information they are seeking? (2) How effectively can programmers determine which of the matches returned from their searches are relevant to their task? and (3) In what ways can tools be improved to support programmers in more effectively expressing their information needs and exploring the results of searches? To begin answering these questions we have conducted a study in which we gathered both qualitative and quantitative data about programmers' search activities. Our analysis of this data is still incomplete, however this paper presents several of our initial observations about how programmers interact with the results from their searches.

1. Introduction

Code search is an important activity for programmers working on a change task to a software system. Various tools exist for searching source code, including tools included with today's Integrated Development Environments (IDEs). Previous research studies have looked at various aspects of search as it relates to change task activities. We highlight just four such studies here. Ko et al. explored the strategies of programmers in finding, understanding, and using relevant information, in the context of a larger change task [3]. LaToza et al. focused on how experience effects work on changes tasks and found, for example, that more experienced programmers tend not to explore as many irrelevant elements [4]. Robillard et al. were interested in the differences between effective, and ineffective programmers and found that successful programmers often performed keyword and cross-reference type searches [5]. Finally, our own research that has been previously reported [6, 7] found that searches tend to produce many irrelevant results and that time consuming exploration is required to determine what is relevant.

To build on this previous work we are at the early stages of a research program that is aiming to answer three research questions: (1) How effectively can programmers express (using today's tools) the information they are seeking? (2) How effectively can programmers determine which of their matches returned from their searches are relevant to their task? and (3) In what ways can tools be improved to support programmers in more effectively expressing their information needs and exploring the results of their searches?

To begin answering these questions we have conducted a study in which we observed programmers performing an assigned change task. As they performed the task we gathered qualitative and quantitative data about their search activities. In this way we have gathered information about 96 search episodes. Our analysis of this data is still incomplete, however this paper presents several of our initial observations about how programmers interact with the results from their searches.

2. Methodology

The study we conducted involved ten participants and ten sessions. In this paper we refer to our participants as P1...P10. Eight of our ten participants (P2, P4, P5, P6, P7, P8, P9, P10) had industrial programming experience. The remaining two (P1, P3) were graduate students. All participants had experience working with Java and the Eclipse development environment.

Each study session lasted 30 minutes. During the session the participant was asked to work on a change task to a large software system called Subclipse[1] (a popular open source plugin that provides support for Subversion[2] within the Eclipse IDE[3]). Subclipse contains approximately 70,000

[1] http://subclipse.tigris.org
[2] http://subversion.tigris.org
[3] http://www.eclipse.org

lines of code and all of our participants were initially newcomers to the system. We did not expect that our participants would have sufficient time to complete the task, and we were primarily interested in their searching activities as they worked on the task.

For each session we randomly assigned one of two different change tasks. Both tasks were taken from the Subclipse project's issue tracking system (specifically task one was based on issue 798 and task two was based on issue 801). These issues were addressed in revisions 3993 and 4012 of the Subclipse version control system, so we had our participants work with revision 3992 for task 1, and 4011 for task 2.

To facilitate data collection we had each participant work as a pair [9] with one of the researchers (the second author of this paper). The researcher was at the keyboard and the participant directed the work on the task by giving instructions to the researcher. We took this approach as a variation on the Think Aloud method [8]. The discussion between the participant and the researcher was recorded and a screen capture video was made for each session. Following the session a second researcher (the first author of the paper), who was also present during each session, conducted a short interview with the participant to further explore issues around their searching activities.

To carry out the change task, our participants used the Eclipse Java Development Environment[4] and they were given a one page document describing the eight major kinds of searches supported by Eclipse. In the following we describe only the kinds of searches used by our participants.

Open Type: Supports searching for classes or interfaces based on a partial name or pattern.

File: Supports searching for text within all of the files in the Workspace.

Find in File: Supports searching for a piece of text within a specified file.

Java: For finding declarations, references and occurrences of Java elements (packages, types, methods and fields).

References: Performs a search for all references to a specified code element or elements matching a keyword.

Declaration: Performs a search for all declarations of a specified code element or elements matching a keyword.

Each time a participant used one of these searches we call this a search episode. Our data set consists of 96 completed episodes and in our ongoing analysis we are using these episodes as the basic unit of analysis.

[4] http://eclipse.org

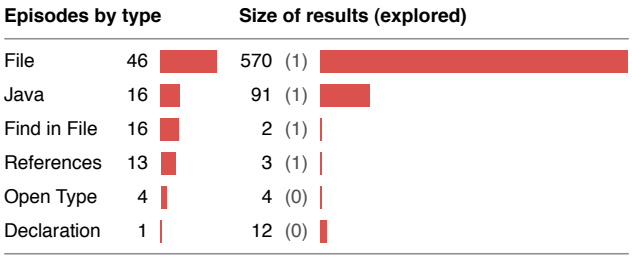


Figure 1. The number of times our participants conducted each type of search. For each type of search the average number of results is shown. The average number of results explored is shown in brackets.

3. Findings

3.1. Searches and Results

Figure 1 shows the number of episodes for each kind of search that our participants performed. Also shown in the figure are the average number of results and the average number of results that were explored. Eclipse's File search was by far the most popular amongst our participants, accounting for approximately half of of all search episodes. File search is the most inclusive search and as compared to a Java search has fewer options for scoping the search, so it is unsurprising that on average it returns the most results (570).

Some of our data suggests that participants use File search when they have very little information to base their exploration. For example, P1 began with File searches because "I don't feel like there's a starting point other than the keyword." Similarly: "when I don't know what I'm looking for, then usually I start with a text search" (P2). Participant P7 performed two similar File searches consecutively (the first search was for the keyword "image"; the second was for the keyword "icon") as he explored which term was used for this concept in the system. These cases suggests that if our participants were not newcomers to the code base or if our sessions were longer we may have had proportionately fewer File searches in our data set.

The average number of results in a result set was 290. We found that the 96 search episodes could be divided into three nearly equal categories, based on the size of the result sets:

- **No results.** 30 of the 96 searches performed by our participants returned no matches.
- **1 – 9 results.** 32 of the searches performed by our participants returned between 1 and 9 matches.

Matches	Episodes	Explored	Time
0	30	0	27s
1–9	32	1.2	66s
10+	34	1.0	107s

Table 1. The number of episodes, the average number of elements explored and the average time taken for an episode for three different categories of result sizes.

- **10 or more results.** Finally, the remaining 34 searches returned 10 or more matches. The largest result set returned had 4770 matches.

Some quantitative data, organized around these three "bins" is summarized in Table 1.

This means that a full one third of the time our participants did a search they got back no results. At times this was surprising or frustrating for our participants, and in some cases it meant that they were off track and needed to consider a different approach. For example, "it should be contained there" and " why the **** is it not finding anything then?" (P8). A very large number of matches was also difficult for our participants to deal with. We discuss this more in the next two sections.

3.2. Time Spent

To analyze the amount of time spent on each search episode we have given each of the 96 complete episodes in our data set a start and an end time. We consider a search episode to be started as soon as a participant starts to describe a search that they would like to perform. We considered this episode ended when they last explore one of the results in the result set. If no results are explored, we considered the episode time to be complete at the latter of when the search returned the last result into the result set, or the participant last comments on the results.

The longest amount of time spent in a single search episode was about five minutes and the shortest amount of time for a particular search episode was 5 seconds. The mean amount of time spent in a single search episode was 69 seconds and the median was 39 seconds. Table 1 shows the average amount of time spent on episodes in each of the bins described above (27 seconds, 66 seconds and 107 seconds). On average more time was spent on episodes where ten or more results were returned than on episodes in the other bins. However, in our data there is not a clear trend that would suggest that more results means more time spent.

In some cases, a large result set seemed to deter the participants from spending much time going through the results–or in even looking at any of the results. When one of participant P4's searches returned 4770 matches he said "that doesn't seem to be especially helpful" and moved on to another search. Later he performed a search which returned 2584 matches and said "ok, so that didn't really work out" before abandoning the search.

Explored	0	1	2	3	4	5
Episodes	17	38	5	3	2	1

Table 2. The number of episodes in which a particular number of elements were explored.

3.3. Exploration Activity

For each search episode we have also tracked the number of elements in the search result that the participant chose to explore. For our analysis of exploration activity we are omitting episodes in which no matches were returned. This leaves us with 66 episodes in which there was potential for exploration activity.

As has been noted in previous research, we found that our participants were generally exploring only a small number of matches. The mean number of results that were explored within a single search episode was 1.06 results, and the median was 1. Table 1 shows the average number explored by bin. Note that the average number explored goes down slightly as the number of results increases.

As shown in Table 2, the most common behavior exhibited by our participants was to explore exactly one result from a result set, regardless of the size of the result set. Interestingly, another quite common behavior was to explore none of the results.

How exactly programmers choose which items to explore is still an open question. When asked about their choices most of our participants said that they guess that something is relevant based on the package and element name displayed in the results view. Participant P3 said that in making this decision he "... looked at the result that looked most promising". This participant when asked how they decide what looks promising stated that it was "Totally based off name" and their understanding of a similar system. Many of our participants seemed to look for any element that seemed relevant, rather than looking at all of the results and selecting the most relevant element.

We also found that making such a decision simply based on name is not always reliable. Participant P2 performed a search that returned four matches, one of which was arguably the element most directly related to the assigned change task. He looked over the names of the elements and

decided to try a different type of search: "I'm not sure this is useful, maybe you should do a text search for the same string".

The result sets that were returned by Eclipse were displayed by default in a tree format, with the exception of the Open Type which was in a list with no context, and Find in File which would jump to the elements one at a time. For the File search the top level contained the base directory of the project, followed by deepening levels based on the directories as far as the files. For the other tree results, the top level of the tree contains packages and the next two levels contains types and method or field names. If either of the trees are expanded further, Eclipse will also show one line of context which shows the match in context. Participant P6 was our only participant that made use of this contextual information.

4. Limitations

Our study involved a relatively small number of programmers, performing two specific software change tasks. Also, our analysis of the collected data is incomplete. As a result, care should be taken in drawing general conclusions based on our results. In the following we discuss a few specific limitations.

Our participants were newcomers to the system we asked them to change. In our experience this is a common scenario for programmers in industry and is convenient for conducting controlled studies. However, it may also mean that our results provide limited insights into the search behavior of programmers performing change tasks to systems with which they are familiar.

Our study setup involved one of the researchers being the "driver" during the session, similar to a pair programming experience. This was effective as it encouraged the participant to vocalize their intentions and allowed them to perform searches without necessarily knowing how to perform it in Eclipse. While this approach was helpful, it is an open question what effect the pairing had on the search behavior.

5. Conclusion

The analysis of our data set is not yet complete so we are not yet in a position to draw strong conclusions. However, we want to highlight two key observations that have relevance to the design of source code search tools.

First, it is clear that the way that search results are presented to programmers in IDE's such as Eclipse provides little support for programmers. Specifically, it appears that element names are not sufficient and we believe that more contextual information may be helpful, though it is not yet clear what form that should take. Programmers can obviously get significantly more information about search results by exploring the source code of an element, however we found that when they do this, they tend not to return to consider other results.

Second, along with only exploring a small number of matches (often zero or one) a large set of matches is not first examined before selecting which element(s) to explore. Based on this we would argue that the sorting of results should be given more attention, possibly based on confidence values such as those used by the Hipikat system [1]. We have found that programmers are put off by large result sets, but if the results were ranked in a meaningful way, it is possible that programmers would be able to make use of such results. How best to rank results is an open question, however we believe that a successful ranking would likely need to be context aware, possibly based on contextual information maintained by tools such as Mylyn [2].

References

[1] D. Cubranic and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering*, pages 408–418, 2003.

[2] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM, 2005.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(10):971–987, 2006.

[4] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. pages 361–370. Association for Computing Machinery, 2007.

[5] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[6] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the SIGSOFT Foundations of Software Engineering Conference (FSE)*, 2006.

[7] J. Sillito, G. C. Murphy, and K. D. Volder. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[8] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg. *The Think Aloud Method; A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

[9] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair-programming. *IEEE Software*, 17(4):19–25, 2000.