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

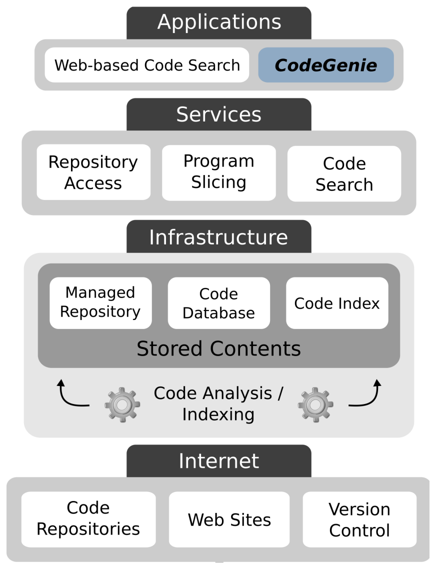

Figure 1. Sourcerer system architecture

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.

Table 1. Sample search index fields

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

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.