Chapter 8
Infrastructure for Building Code Search Applications for Developers

Sushil Krishna Bajracharya

Abstract The large availability of open source code on the Web provides great opportunities to build useful code search applications for developers. Building such applications requires addressing several challenges inherent in collecting and analyzing code from open source repositories to make them available for search. An infrastructure that supports collection, analysis, and search services for open source code available on the Web can greatly facilitate building effective code search applications. This chapter presents such an infrastructure called Sourcerer that facilitates collection, analysis, and search of source code available in code repositories on the Web. This chapter provides useful information to researchers and implementors of code search applications interested in harnessing the large availability of source code in the repositories on the Web. In particular, this chapter highlights key aspects of Sourcerer that supports combining Software Engineering and Information Retrieval techniques to build effective code search applications.

8.1 Introduction

Building a search application for source code available on the Web can be a major undertaking. The specificities and needs for code search pose interesting opportunities and challenges to build effective code search applications. For example, unlike natural language text, source code has an inherent scarcity of terms that describe the underlying implementation. From an information retrieval perspective source code can be much harder artifact to retrieve if we rely solely on the terms that are present in it. On the other hand, source code has rich structure compared to natural language text. The structure comes from the organization of source code entities in

S.K. Bajracharya (✉)
Black Duck Software, Burlington, MA, USA
e-mail: sbajra@acm.org

the implementation, and also the various relations that exist among those entities. As a result, code search applications can leverage structural information extracted from source code for effective retrieval.

Sourcerer is an infrastructure that facilitates collection, analysis, and searching of source code harnessing its inherent structural information. This chapter provides details on the aspects of Sourcerer that makes it a unique state-of-the-art platform to build code search applications. Rest of the chapter is organized as follows. Section 8.2 provides an overview of three code search applications that Sourcerer supported, and presents the infrastructure requirements demanded by the code search applications. Section 8.3 introduces the key elements of Sourcerer's Architecture (Fig. 8.1). Section 8.4 provides an in-depth discussion of various models that lie at the core of Sourcerer's architecture. Section 8.5 summarizes the contents that are stored in Sourcerer's repository, and Sect. 8.6 discusses services that allow access to the stored contents. Section 8.7 provides details on the tools developed to build the contents and services in Sourcerer. This chapter concludes by summarizing key features that enabled the three code search applications (Sect. 8.8) and discusses related work in Sect. 8.9.

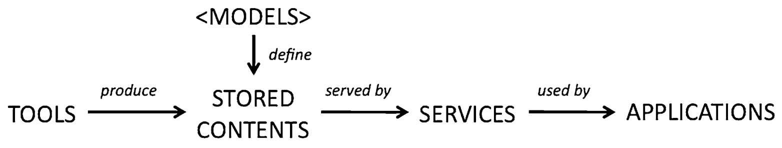


Fig. 8.1 Elements of Sourcerer infrastructure

8.2 Infrastructure Requirements for Code Search Applications

Sourcerer enabled building three code search applications during the course of its development. These code search applications put forth various requirements on Sourcerer as a code search infrastructure. Overall, these requirements boil down to three basic functionalities:

1. **Collection and Storage**: Fetching source code from forges on the Web and storing them locally with required metadata intact.
2. **Analysis and Indexing**: Extract both lexical (textual) and structural information (entities and relations) from the source code downloaded from the Web.
3. **Search and Retrieval**: Provide access to underlying contents (source, search index, and structural information) as needed by different applications

The core of these requirements demanded code-specific analyses and heuristics to be incorporated into various models, services and tools. A pragmatic decision was made to only provide support for the Java programming language in Sourcerer to be able to meet these requirements without being overburdened with the complexity of analyzing and supporting all possible programming languages.

Next, we briefly introduce the three code search applications and look at the infrastructure requirements they brought in.

8.2.1 Sourcerer Code Search Engine

Sourcerer Code Search Engine (SCSE) is a web-based code search engine to find code entities in open source projects. SCSE provides a central user interface, code specific search operators (e.g. limiting search on comments or code portions), and employs a ranking scheme that leverages underlying structure and relations in code. As a code search engine, SCSE aggregates and presents meta-data related to code entities in the search results. This meta-data includes the origin information about the code (i.e. the name and location of the open source project where the code came from), license information, version, and category of the project. SCSE also allows viewing and browsing source code, following usage relations, and provide detail information on code structure (threading properties, Java attributes, and micropatterns [11]).[1] With these features SCSE allowed a central access to search thousands of open source projects. Implementation of SCSE itself required development and integration of several core infrastructure pieces of Sourcerer. Since its development several commercial applications are now available that offer similar features as Sourcerer. Therefore, the infrastructure requirements for SCSE resemble much similarity to the requirements to build a large-scale code search engine.

8.2.2 CodeGenie: A Test-Driven Code Search Application

CodeGenie is a code search application that allows a developer to start from a unit test and search for a working set of code entities (classes) that would implement the desired feature as specified in the unit test. CodeGenie is a plugin for Eclipse IDE that works as a Test-Driven Code Search (TDCS) application. TDCS combines the 'Test-First' principle of Test-Driven Development with code search. CodeGenie allows a developer to start from an existing unit test that specifies a desired functionality to be implemented. After this, CodeGenie can construct a query from the unit tests, execute a search using the query on Sourcerer, and bring back found code entities. A developer can choose any of the found entity, look at its source code, and merge the code in her workspace to get the desired functionality originally being tested in the unit test. While merging the code, CodeGenie uses a special service provided by Sourcerer, called the code slicing service, that computes and extracts

[1] This code structure information was originally intended to be used in implementing code similarity techniques based on detailed code structures, but was not developed further as development in Sourcerer proceeded.

a dependency slice (a set of synthesized code entities that makes the found code entity compilable and workable in the workspace). CodeGenie has features external to the infrastructure that eases the selection and merging process, for example an ‘unmerge’ operation to get rid of previously merged code and select a new entity to be merged in the workspace. CodeGenie relies on Sourcerer for these features to work. For example, CodeGenie provides the merging and unmerging of code entities that came from the dependency slice by using a unique identifier given to the dependency slice by Sourcerer.

8.2.3 Sourcerer API Search

Sourcerer API Search (SAS) helps developers to find code snippets that serve as API usage examples. Developers working with large frameworks and libraries might not know or remember all APIs available to them. SAS attempts to provide an exploratory search interface to help such developers in finding code snippets to learn the names and usage patterns of APIs to perform certain programming tasks.

There are four major features in SAS that make finding code snippets easier. First, a list of code snippets that show sections of code with auto-generated comments highlighting the APIs that are used (showing their fully qualified names) and their patterns of usage (by showing relations such as calls, instantiations etc.). Second, a list of code entities that constitute the most popular APIs in a given search result; these APIs can be used as filters to narrow down search results. Third, a set of words as tag-cloud with every result set, where the words can be used for query reformulation. These words are picked by analyzing the names of the code entities found, names of the popular APIs used, and names of code entities that are similar to the entities in the result in terms of API usage. Fourth, a *more like this* feature to find similar code entities based on API usage. This allows users to get recommendation on entities that exhibit similar API usage patterns which is helpful to find more examples once a candidate example (or code entity) is found.

Structural Semantic Indexing (SSI): SAS uses an index with a set of relevant terms mapped to each code entity found in a code collection. These terms are extracted from various places: the source for the entity itself, the source for the APIs that the entity uses, and the source for the code entities that have similar API usage as the code entity. With terms coming from various places the index is able to match queries with relevant code entities (for API examples) even if the source for the entity does not contain all of the terms in the query. The indexing technique used for this is called Structural Semantic Indexing (SSI), and was enabled using various pieces of the Sourcerer infrastructure.

8.2.4 Infrastructure Requirements

SCSE, being the first and the most general application has the basic requirements. CodeGenie and SAS have some common requirements as SCSE, along with some new ones of their own. The overall infrastructure requirements to build the three different code search applications emerge out as follows:

Common Requirements:

- Crawling open source forges, extracting project metadata; downloading and checking out source code from open source forges and associating project metadata with the checked out code
- Language aware parsing of source code to extract structural information (entities and relations)
- Indexing source code entities to make them searchable
- Ability to store and retrieve source code for code entities

SCSE:

- Compute rank using structural and lexical information

CodeGenie:

- Code search service that could accept a structured code query, where the query expresses matches on various parts of the code such as class name, method signature, return types, and method arity (number of arguments)
- Search results at the granularity of code entities such as classes and methods so that the IDE can show types and method signatures in the result pane
- A special code slicing service that could construct a set of synthesized code entities that makes the retrieved code entity declaratively complete (i.e., all dependencies must be resolved)

SAS:

- Data sources to produce an index using SSI
- For a given set of code entities in search result, ability to get a list of most popular APIs that are used by the code entities
- For a given code entity, ability to get a list of other code entities that have similar API usage
- For a given code entity, and a set of APIs, ability to provide details on where and how the APIs are used

8.3 Infrastructure Architecture

Sourcerer provides the requirements posed by three code search applications through a collection of *services*. These services provide programmatic access to the underlying data (*stored contents*) that Sourcerer produces and stores. The format and schema of the data is defined by a set of *models* that are developed considering various needs for storage and retrieval needed for the services. Sourcerer also consists a set of standalone *tools* that collect, analyze, produce, and persist the needed contents. Figure 8.1 depicts how services, stored contents, Models, Tools, and (Code Search) Applications constitute Sourcerer's overall architecture.

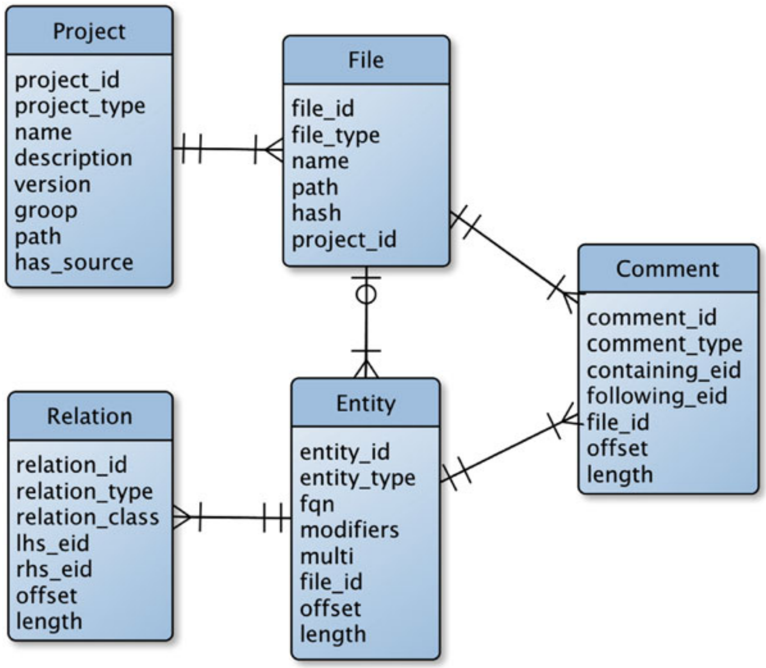


Fig. 8.2 Sourcerer's relational model

8.4 Models

Three models define the basic mechanisms for storing and retrieving information from the source code available in Sourcerer's repository.

8.4.1 Storage Model

The Storage Model defines the structure and physical layout of files in Sourcerer's local repository. A layered directory structure was chosen for two main reasons.

First, it allows projects from the same source to be grouped together, which makes adding or removing contents more straightforward. Second, some sort of branching turned out to be required, not to overburden the file system with tens of thousands of subdirectories in a single directory. The files collected from open source projects are stored in a folder according to the following template:

```
<repo_root>/<batch>/<id>
```

Above, `<repo_root>` is a folder assigned as the root of Sourcerer's file repository. Given the root folder, the individual project files are stored in a two-level directory structure defined by the path fragment `<batch>/<id>`. `<batch>` is a top-level folder in the directory structure that indicates a given batch. For example, a crawl from a specific online repository or a collection of fixed number of projects can denote a batch. Inside `<batch>`, another set of folders exists. Each second-level folder in the local repository, indicated by `<id>` in the above template, contains the contents of a specific project. Each `<id>` directory contains a single file and two sub-directories, as shown below:

```
<repo_root>/<batch>/<id>/project.properties
<repo_root>/<batch>/<id>/download/
<repo_root>/<batch>/<id>/content/
```

Above, `project.properties` is a text file that stores the project metadata as a list of name value pairs. `download` is a folder that contains the compressed file packages that were fetched from the originating repository (e.g., a project's distribution in Sourceforge). `content` contains the expanded contents of the `download` directory. Once the contents of the download directory have been expanded, the directory itself is usually emptied in order to free up space.

The project contents in the `content` directory can take two different forms, depending on its format in the initial repository. If the project contents are checked out from a remote software configuration management (SCM) system such as svn and cvs, the file located at a relative path `path` in the originating repository (e.g., Sourceforge) exists in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/content/<path>
```

Instead, if the project is fetched from a package distribution, a source file can be found in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/content/package.<i>/<path>
```

Above, `package.<i>` indicates a unique folder for each *i*th package that is found in a remote repository. `path` indicates a relative path of a source code file that is found inside the *i*th archived package, which is unarchived inside the `package.<i>` folder.

Project metadata: The `project.properties` file is a generic project description format that generalizes the project metadata from the online repositories. Many attributes in `project.properties` are optional, except for the following:

- `crawledDate`: indicates when the crawler picked up the project information
- `originRepositoryUrl`: URL of the originating repository; e.g., http://sourceforge.net
- `name`: project's name as given in the originating repository
- `containerUrl` : project's unique URL in the originating repository

And, one or both of the following: (i) Information on project's SCM system indicated by `scmUrl` (ii) Information on project's source package distributed on the originating repository, as indicated by the following fields:

- `package.size` indicating total number of packages distributed.
- `package.name.`*i* indicating name of the *i*th package, where $1 <= i <= package.size$, and *i* indicates a unique integer denoting a package number.
- `package.sourceUrl.`*i* indicating the URL to get the *i*th package from the originating repository.

The example below shows metadata description for a project crawled from Google code hosting.

```
#Thu Sep 24 16:15:01 PDT 2009
releaseDate=null
name=dlctarea1
category=DLC, Java, Netbeans, FileChooser
languageGuessed=Java
versionGuessed=$SCM
scmUrl=svn checkout
  http\://dlctarea1.googlecode.com/svn/trunk/
    dlctarea1-read-only
license=GNU General Public License v2
keywords=null
sourceUrl=null
exractedVersion=$SCM
projectDescription=Tarea n\uFFFD 1
fileExtensions=null
originRepositoryUrl=http\://code.google.com
containerUrl=http\://code.google.com/p/dlctarea1/
contentDescription=null
crawledDate=2009-Sep-23
```

Jar Storage: In addition to the top-level `batch` directories described above, the local repository also contains a single `jars` directory. The jars directory is structured as follows:

```
<repo_root>/jars/project/<jar_path>
<repo_root>/jars/maven/<jar_path>
<repo_root>/jars/index.txt
```

The `project` subdirectory contains all of the jar files that come packaged with the projects in the main repository. This directory is populated by crawling through the repository itself, and copying every jar found. The copying is done so that these jar files can be modified, if necessary, without altering the original projects.

The `maven` subdirectory contains a mirror of the Maven2 central repository[2] [43]. Lastly, `index.txt` contains an index that maps from the MD5 hash of a jar file to its location in the directory structure. This index is used to link the jar files from the projects to the files contained in the `jars` directory.

Sourcerer's project metadata format enables capturing description of projects and contents across various online repositories.

The Storage Model provides a standard for storing project files in Sourcerer and is not directly used by applications. Applications rely on other higher-level abstractions to access the contents stored in Sourcerer.

8.4.2 Relational Model

Sourcerer's relational model defines the basic source code elements and the relations between those elements. It supports a fine-grained representation of the structural information extracted from source code. It also links the code elements/relations with their locations in physical artifacts.

Two major goals guided the design of Sourcerer's relational model. First, it had to be sufficiently expressive to allow fine-grained structure-based analyses and search over code structure. Second, it had to be efficient and scalable enough to include the large amount of code from thousands of open source projects. To meet these two goals we decided to use an adapted version of Chen et al.'s [7] C++ entity-relationship-based metamodel as Sourcerer's relational model for source code. In particular, their decision to focus on what they termed a *top-level declaration* granularity provides a good compromise between the excessive size of finer granularities and the analysis limitations of coarser ones.

The relational model consists of the following five elements: Project, File, Entity, Comment, and Relation.

A **Project** model element exists for every project contained in Sourcerer's repository, as well as every unique Jar file. A project therefore contains either a collection of Java source files and jar files, or a collection of class files. A **File** model element represents these three types of files: source (`.java`), jar (`.jar`) or class (`.class`). Both source and class files are linked to sets of **Entities** contained within them, and to the **Relations** that have these entities as their source and target. Jar files, on the other hand, are linked to their corresponding jar projects, which in turn contains all of the **Entities** and **Relations**.

[2] Maven is a build system for Java that provides the facility to fetch required libraries from a central repository [42].

An **Entity** model element either corresponds to an explicit declaration in the source code (e.g., Class, Interface, Method), a Java package,[3] or Java types that are used but do not correspond to a known explicitly declared type (e.g., Array, Type Variable). An entity type is `UNKNOWN` when the type cannot be determined due to uncertainty in the analysis. Table 8.1 lists all entity model element types defined in Sourcerer. These types adhere to their standard meaning in Java, as defined in the Java Language Specification (JLS) [12].

A **Relation** model element represents a dependency between two Entities. A dependency *d* originating from a source entity *s* to a target entity *t* is stored as a Relation *r* from *s* to *t*. Table 8.2 contains a complete list of the relation types with a brief description and example for each. All of the relations are binary, linking a source entity to a target. The source entity for a relation is smallest entity that contains the code that triggers that relation. While containment is clear for most of the entities, it should be noted that `FIELD`s are considered to contain their initializer code and `ENUM CONSTANT`s are considered to call their constructors. The source entity is always found within the project being examined. This is not necessarily true of the target entity. It can be a reference to the Java Standard Library or any other external jar. In fact, due to missing dependencies, sometimes it is impossible to resolve the type of the target entity.

PACKAGE
CLASS
INTERFACE
ENUM
ANNOTATION
INITIALIZER
FIELD
ENUM CONSTANT
CONSTRUCTOR
METHOD
ANNOTATION ELEMENT
PARAMETER
LOCAL VARIABLE
PRIMITIVE
ARRAY
TYPE VARIABLE
WILDCARD
PARAMETRIZED TYPE
UNKNOWN

Table 8.1: Entity types

A **Comment** model element represents the comments defined in the Java source code.

[3] Packages are not considered to be standard declared entities as they do not have a single declaration.

Figure 8.2 shows Sourcerer's relational model using an ER-diagram. It shows the five elements of Sourcerer's relational model and a set of attributes for each of them. Table 8.3 provides the details on all the attributes of the model elements. Figure 8.2 and Table 8.3 provide information on how the model elements are linked with each other, and how the attributes in the relational model link the relational model elements with the storage model. For example, Project element's 'path' attribute links it to the physical location defined by the storage model.

Various tools in Sourcerer make use of this information to connect the relational information with the textual contents stored in the physical files.

Entities and *Relations* are the key elements of the Sourcerer's relational model that enables code specific search capabilities. Capturing and associating fully qualified names for code entities allows referring and looking up code entities across projects using the FQNs as keys. Therefore, FQNs for entities enables analysis of relations across projects. This led to innovative use of structural information in code search applications such as: (i) computing CodeRank (adaptation of Google's Pagerank algorithm on code graph) and using it as a ranking heuristic in SCSE, (ii) and using feature vectors made up of FQNs of used entities as a basis to compute usage similarity for entities in SSI.

8.4.3 Index Model

The **Index Model** complements Sourcerer's relational model by facilitating application of information retrieval techniques on the code entities. The index model specifies a **Document** representation for each code Entity in the relational model. A document in the index model is made up of a collection of **Field**s. Each field has a name and different types of values associated with them, the most fundamental being a collection of **Term**s. A term is a basic unit for search/retrieval. Terms are extracted from various parts of an entity, and stored in a corresponding field of a document representing a code entity.

Sourcerer's information retrieval component is based on the popular Lucene [41] information retrieval engine. Therefore, its index model confirms to how Lucene models its contents. More details on Lucene's contents model are available in [25].

Fields in Sourcerer's index models can be categorized into five types:

1. Fields for *basic retrieval* that store terms coming from various parts of a code entity.
2. Fields for *retrieval with signatures* that store terms coming from method signatures and also terms that indicate number of arguments a method has.
3. Fields storing *metadata*, for example the type of the entity, so that a search could be limited to one or more types of entities.

Relation	Description	Example
`INSIDE`	Physical containment	`java.lang.String INSIDE java.lang`
`EXTENDS`	Class extension	`java.util.LinkedList EXTENDS java.util.AbstractSequentialList`
`IMPLEMENTS`	Interface implementation	`java.util.LinkedList IMPLEMENTS java.util.List`
	Interface extension	`java.util.List IMPLEMENTS java.util.Collection`
`HOLDS`	Field type	`java.lang.String.offset HOLDS int`
`RETURNS`	Method return type	`java.lang.String.toCharArray() RETURNS char[]`
`READS`	Field read	`...String.<init>(java.lang.String) READS java.lang.String.offset`
`WRITES`	Field write	`java.lang.String.<init>() WRITES java.lang.String.offset`
`CALLS`	Method invocation	`...String.indexOf(int) CALLS java.lang.String.indexOf(int,int)`
`INSTANTIATES`	Constructor invocation	`foo() INSTANTIATES java.lang.String.<init>`
`THROWS`	Declared checked exception	`java.io.Writer.write(int) THROWS java.io.IOException`
`CASTS`	A cast expression	`java.langString.equals( java.lang.Object) CASTS java.lang.String`
`CHECKS`	An instance of expression	`java.langString.equals( java.lang.Object) CHECKS java.lang.String`
`ANNOTATED BY`	Annotation	`java.lang.Override ANNOTATED BY java.lang.annotation.Target`
`USES`	Any reference	`java.lang.String.<init>() USES char`
`HAS ELEMENTS OF`	Array element type	`char[] HAS ELEMENTS OF char`
`PARAMETRIZED BY`	Associated type variables	`java.util.List PARAMETRIZED BY <E>`
`HAS BASE TYPE`	Generic base type	`java.util.List<java.lang.String> HAS BASE TYPE java.util.List`
`HAS TYPE ARGUMENT`	Generic type argument	`java.util.List<java.lang.String> HAS TYPE ARGUMENT java.lang.String`
`HAS UPPER BOUND`	? extends	`<? extends java.util.List> HAS UPPER BOUND java.util.List`
`HAS LOWER BOUND`	? super	`<? super java.util.List> HAS LOWER BOUND java.util.List`

Table 8.2: Relation types

	Description
Project	
project_id	Unique identifier for a project
project_type	Denotes whether this project represents a crawled project, or a Jar file
name	Name of the project as it appears in the originating Internet repository
description	Description of the project from the originating Internet repository
version	Version of this project as extracted from originating Internet repository
groop	Specific field applicable to Maven Jars
path	Corresponds to the `<batch>/<id>` path fragment as defined by the storage model
has_source	Denotes whether the project contains source files
File	
file_id	Unique identifier for a file
file_type	Denotes the file's type – source, Jar, class
name	Name of the file in the file system
path	Corresponds to either `<batch>/<id>/content/<path>`, or `jars/<jar_path>` as defined by the storage model
hash	Unique MD5 hash, applicable for Jars only
project_id	project_id that this file belongs to
Entity	
entity_id	Unique identifier for an Entity
entity_type	One of the several code entity types. (e.g., CLASS, METHOD)
fqn	Fully qualified name (FQN) of the entity
modifiers	Modifiers defined for the code entity
multi	Denotes array dimension, applicable for ARRAY types only
file_id	file_id that this entity is extracted from
offset	Start position of this entity in the source file
length	Length of this entity in the text (source file)
Relation	
relation_id	Unique identifier for a relation
relation_type	One of the several code relation types. (e.g., CALLS, EXTENDS)
relation_class	Denotes whether the relation terminates to a library or a local entity
lhs_eid	The source entity that the relation originates from
rhs_eid	The target entity that the relation terminates into
offset	Start position in the source entity's corresponding file where this relation exists
length	Length of the text in source code where this relation spans
Comment	
comment_id	Unique identifier for a comment
comment_type	Denotes the comment's type – Javadoc, Block, Line
containing_eid	The immediate code entity that contains this comment
following_eid	The immediate code entity that follows this comment
file_id	File where this comment is found
offset	Start position of comment in the source file
length	Length of this comment in text (source file)

Table 8.3: Sourcerer's relational model elements details

4. Fields that store information to facilitate *retrieval based on structural similarity* (e.g., fields storing fully qualified names (FQNs) of used entities and terms extracted from similar entities).
5. Fields that pertain to some *metric* computed on an entity.
6. Fields that store unique identifiers (ids) of entities for *navigational/browsing queries*

Being based on Lucene, Sourcerer's index model is quite flexible. Depending on a specific search application, an instance of a Sourcerer's index schema can have a subset of various field types listed above. The three code search applications built on top of Sourcerer have used code index schemas with different configurations of fields and associated data sources.

Fields for retrieval with signatures allowed precise construction of queries for expressing desired method signatures and relations expected in test cases in CodeGenie. Fields storing retrieval based on structural similarity enabled retrieval schemes in SSI, and *more like this* queries based on usage in SAS. Rest of the index fields supported basic operations of the code search applications as in SCSE.

8.4.3.1 Structured Retrieval

Table 8.4 presents a subset of the fields available in the Sourcerer index. Sourcerer's search index can be searched using Lucene's query language [25, 41]. The following Lucene query demonstrates how different fields are utilized to express a query that incorporates textual as well as structural information:

```
short_name: (day of week)
  AND entity_type: METHOD
  AND m_ret_type_sname_contents: String
  AND m_args_fqn_contents: date
  AND cdef: (date util)
```

Index field	Description
Fields for basic retrieval	
fqn_contents short_name	Tokenized terms from the FQN of an entity Right most fragment of the FQN (w/o method arguments for methods)
Fields for retrieval with signatures	
m_args_fqn_contents m_ret_type_sname_contents	Method's formal arguments tokenized into terms Short name of the method's return type tokenized into terms
Fields Storing metadata	
entity_type	String representation of entity type. (e.g., "CLASS")
Fields for navigation	
fan_in_mcall_local	Entity ids of all local callers for a method from the same project

Table 8.4: Sample search index fields

The above query has the following meaning: find a method with terms `day`, `of` and `date` in its short name (or simple name in JLS [12]), that returns a type with short name `String`, and takes in any number of arguments with term `date` as part of its argument in their FQNs. This is an example of a query that CodeGenie would construct for a unit test that would have an assertion that looks like:

```
Date date = ...
Assert.assertTrue(''Tuesday'',DateUtil.dayOfWeek(date));
```

With an index structure that has fields resembling various structural elements in code, Sourcerer provides a code-specific index model.

8.4.3.2 Code-Specific Retrieval Schemes

Sourcerer's index model enables implementation of retrieval schemes for a variety of code search applications.

A *retrieval scheme* tuned for code search takes a query and returns relevant code entities using a combination of code specific heuristics. A *heuristic* is an idea to associate meaningful terms to code entities.

Consider a source code document in Java as shown in the top right part in Fig. 8.3. If we focus on the method entity (`createResource`) shown inside the code document, there can be multiple ways to associate meaningful terms to that entity. On the top-left part in Fig. 8.3, several metadata related to the method `createResource` are shown. For example 'FQN' indicating the fully qualified name of the method entity, 'Used FQNs' listing the FQNs of the APIs that the code entity uses, and 'Similar Entity' indicating another method entity `makeIcon` that uses the same two APIs as `createResource` uses.

Lower part of Fig. 8.3 shows how we can define several heuristics that would associate different meaningful terms with the method entity `createResource`.

The first heuristic 'Code as Text' treats source code entities as normal text document. Based on some code specific parsing (such as removing symbols and splitting on camel case) 'Code as Text' will associate the following terms with the method entity `createResource`: create, resource, file, open.

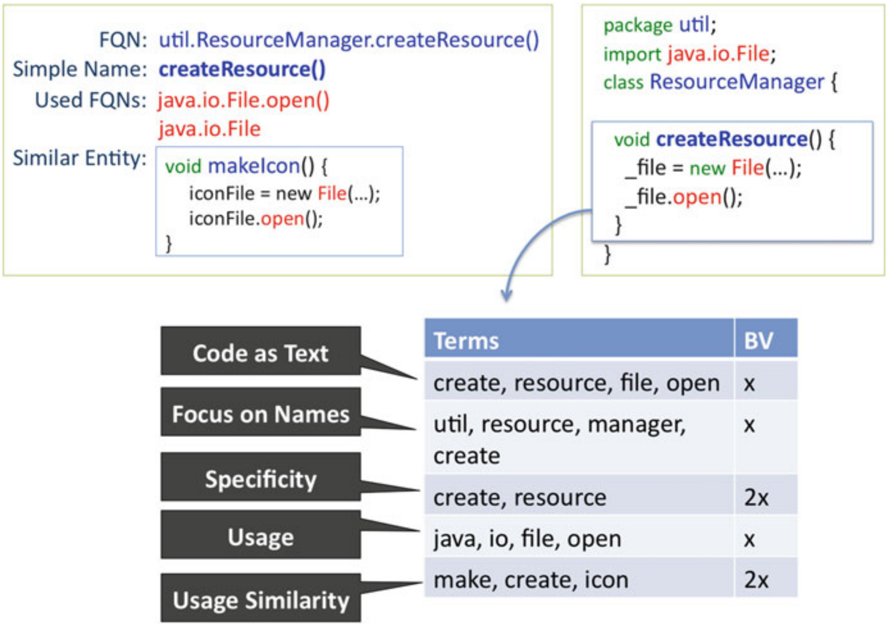
While writing code developers often express their design in some hierarchic fashion; for example the method `createResource` is defined inside the class entity `creatResource` that is further defined inside the package `util`. Programming languages allow expressing such information about hierarchic containment in a naming scheme resulting in fully qualified names (FQNs) for entities. For example, in Java, the FQN of the method `createResource` is given as follows: `util.ResourceManager.createResource()`. The second heuristic 'focus on names' assumes FQNs express structure and design of code entities, and associate terms extracted from FQNs with code entities.

The third heuristic 'Specificity' says that the simple name of the method carries more specific information about a code entity, and therefore terms extracted from simple name should have some higher priority compared to others. This is represented as a boost value (shown as BV in Fig. 8.3) for list of terms associated with 'Specificity' heuristics.

The ability to prioritize the heuristics differently allows experimentation and choosing the most effective retrieval performance.

The fourth heuristic 'Usage' says that the FQNs of the used entities also carry some important information about the functionality of the code entity, as it is by using these FQNs the entity is implementing some feature in the code. Therefore this heuristic extracts terms from the FQNs of the used entities.

Finally, 'Usage Similarity' says that, terms found in code entities that have similar API usage patterns can be used to describe each other. For example, as shown in Fig. 8.3 both methods `createResource` and `makeIcon` are implementing same behavior by using same APIs. This suggests that, to some extent, terms extracted from `makeIcon` can be used to describe the functionality implemented in `createResource`.



```
FQN: util.ResourceManager.createResource()
Simple Name: createResource()
Used FQNs: java.io.File.open()
           java.io.File
Similar Entity: void makeIcon() {
                    iconFile = new File(...);
                    iconFile.open();
                }
```

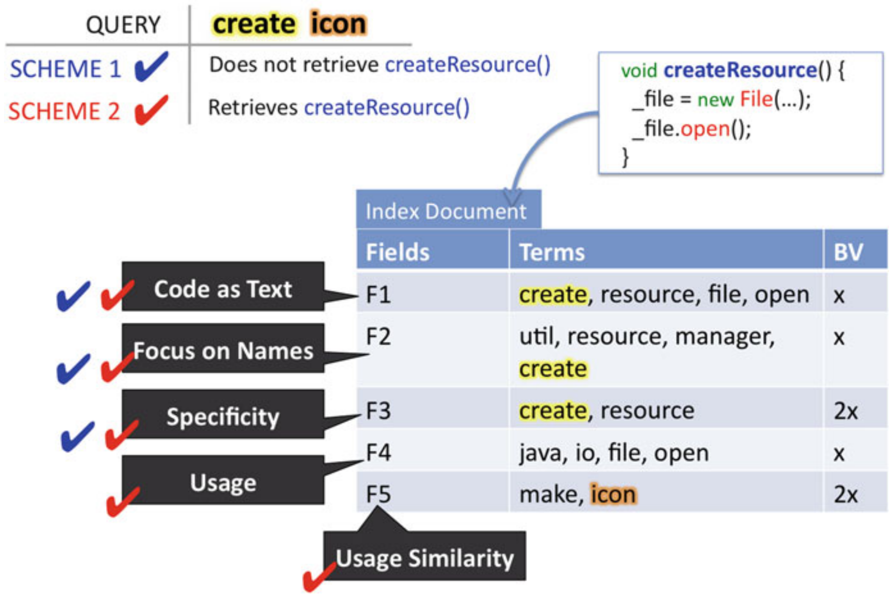
```
package util;
import java.io.File;
class ResourceManager {

  void createResource() {
    _file = new File(...);
    _file.open();
  }
}
```

Terms	BV
create, resource, file, open	x
util, resource, manager, create	x
create, resource	2x
java, io, file, open	x
make, create, icon	2x

Fig. 8.3 Heuristics for code retrieval

Sourcerer's index model allows incorporating these code specific heuristics by leveraging the semi-structured document model of Lucene. For each of the heuristics the index model introduces a field that would store terms extracted based on the heuristic. Each field is given an appropriate boosting value so that some heuristics could be given higher priority (depending on the code search application). With such an index model, a retrieval scheme for a code search application simply specifies which fields to choose to match the user query. A different strategy to retrieve code entities can be implemented by varying these schemes. For example, the top right corner of Fig. 8.4 shows the code snippet for the method entity `createResource` (previously shown in Fig. 8.3). The bottom part of Fig. 8.4 shows an index document with five different fields capturing five different heuristics respectively. The top left part of Fig. 8.4 shows in a tabular form, how two schemes would match the same query `create icon` to the index document (and thus the method entity) differently. **Scheme 1** uses only three heuristics, compared to **Scheme 2** that uses all five.

Scheme 1 looks over a limited set of terms associated with the method entity `createResource`. This set only includes one of the terms `create` present in the query `create icon`. **Scheme 2** includes two more fields that makes it look over a richer set of terms that includes both of the terms found in the query. Assuming that all terms in query need to be matched for a document to be retrieved, **Scheme 2** outperforms **Scheme 1** because **Scheme 2** uses additional heuristics to harvest more meaningful words describing code entities.



QUERY	create icon
SCHEME 1 ✓	Does not retrieve createResource()
SCHEME 2 ✓	Retrieves createResource()

```
void createResource() {
  _file = new File(...);
  _file.open();
}
```

Index Document

Fields	Terms	BV
F1	create, resource, file, open	x
F2	util, resource, manager, create	x
F3	create, resource	2x
F4	java, io, file, open	x
F5	make, icon	2x

Fig. 8.4 Incorporating heuristics in index model

Vocabulary problem is a fundamental problem in information retrieval. It arises from the fact that humans have different vocabulary to describe similar concepts. Consequently, terms used in a query might not be present in all relevant documents. This can severely hinder retrieval because not all users would know the right terms to use to retrieve a relevant document. Sourcerer provides a solution to harvest more meaningful words for code entities by incorporating code-specific heuristics in the index model. This enables developing retrieval schemes that allows code entities to be matched with relevant query terms even when the terms themselves are not originally present in the code entity. This contributes a unique solution to tackle the vocabulary problem in code search.

For an elaborate description of vocabulary problem, see [8]. SAS used **Scheme 2** retrieval scheme shown in Fig. 8.4 and used all five heuristics shown in Fig. 8.3.

8.5 Stored Contents

The Sourcerer infrastructure maintains a collection of stored contents corresponding to each of the three models.

A **File Repository** keeps a collection of files downloaded and fetched from open source repositories in the Internet. The structure of the file repository follows the storage model.

Two different databases store the relational information about the contents in the file repository. First, **ArtifactDB** stores limited information about the jar files found in the repository in order to enable the automated resolution of missing dependencies [28]. Second, **SourcererDB** stores the relational information on all projects, files and code entities that exist in the file repository. Both databases exist as MySql databases whose schemas confirm to Sourcerer's relational model.

A Lucene-based **Search Index** is available that stores information about terms extracted from each code entity in the corresponding documents and fields. The search index uses a code index schema following the index model.

Sourcerer's web site [34] provides details on the most recent statistics on the size of its contents. Currently, its repository contains above 3 million source files from 18,826 open source projects.

8.6 Services

All the artifacts managed and stored in Sourcerer are accessible through a set of Web services. These services provide a layer of abstraction and programmatic access to rapidly build applications that can leverage the underlying contents stored in Sourcerer.

Relational Query: Both ArtifactDB and SourcererDB are implemented as MySql databases. They provide direct access to query the underlying structural/relational information in Sourcerer using standard SQL. Relational Query is the basis for rich structural queries over code. Dependency slicing, code rank, and usage similarity all relied on SQL queries. As another use-case of using relational information, given below are some details on snippet extraction (taken from [3]) implemented for SAS.

Snippet Extraction in SAS: The retrieval scheme for SAS takes a keyword query and returns a ranked list of code entities as search result. This ranked list of entities is called hits and each entry in the list is called a hit. The retrieval scheme also returns the total number of entities in the index that match the query. For each hit the corresponding 'entity_id' (a unique identifier for a code entity) is available. Further details about the code entity can be queried from SourcererDB using the 'entity_id'. SAS uses the information returned by its retrieval scheme to extract a corresponding code snippet for each hit (entity) in the list.

```
  input  : hits = top 'n' hits returned as search results; where, n = max_of(10,
           10% of total hits)
  output: top_used = list of top used entities
1 begin
2     list_eid = all entity ids from hits;
      /* getTopApis(..) selects top 5 non-JSL (Java Standard Library) entities of
         each type (Interface, Method, Constructor, Classes) from SourcererDB
         such that they are used by at least 3 entities in the hits            */
3     top_used = getTopApis(list_eid);
4 end
```

Algorithm 1: Getting the list of top used entities

Snippet extraction proceeds in two steps. First, given a set of hits, a list of top APIs (used entities) is generated. This process is shown in Algorithm 1. As an input Algorithm 1 takes a list of top 'n' hits where, 'n' is the greater of 10 or 10 % of the total number of hits. These 'n' hits give 'n' unique entity ids (Line 2). To find the list of top used entities, the search application queries SourcererDB for the top non-JSL entities that are used by the entities in the list (Line 3). For each entity the top five Interfaces, Methods, Constructors, and Classes are selected. Among all these used entities in the list, only those entities that are used by at least three different entities are returned as the top used entities (output of the algorithm).

```
   input  : eid = entity id, top_used = top used entities
   output: snip = an annotated code snippet
 1 begin
 2     snip = empty string;
       /* getUsedPositions(..) looks up SourcererDB and returns all positions in
          the code where top_used entities are used. Positions are mapped to a list
          of used entities                                                      */
 3     used_pos_map = getUsedPositions(top_used, eid);
 4     forall the position IN used_pos_map do
 5         rationale = empty string;
 6         forall the used_entity IN used_pos_map[position] do
               /* Below, append(a,b) returns a new string by appending string ‘b’
                  to ‘a’.createRationale(..) selects relation type and FQN of used
                  entity and creates a rationale as a comment                   */
 7             rationale = append(rationale, createRationale(used_entity, eid));
 8         end
           /* extractFragment(..) extracts the surrounding expression in a code
              entity from position                                              */
 9         snip_fragment = extractFragment(eid, position);
           /* appendSnip(..) works same as append(..) and returns true if
              rationale and snip_fragment do not already exist in snip          */
10         if appendSnip(rationale, snip_fragment) $\notin$ snip then
11             snip = appendSnip(snip, rationale);
12             snip = appendSnip(snip, snip_fragment);
13         end
14     end
15 end
```

Algorithm 2: Snippet extraction

The second step involves generating code snippet for each entity in the hits. This is done using the list of top used entities and the ‘entity_id’ of a given hit. The algorithm for this process is shown in Algorithm 2. The procedure first queries SourcererDB to locate all the positions in the source of an entity where any of the top APIs are used (Line 3). For all APIs that are used in a position, a rationale comment is generated (Lines 5–8). A rationale comment indicates the type and FQN of the used API. Then, a few of the surrounding lines of code are extracted from each starting position (Line 9). Rationale comments are inserted on top of these extracted lines (Lines 10–13). Finally, a sequence of these commented code fragments is returned as an example code snippet. A sample Java code snippet generated using a hit returned for a query “write to workbench error log” is shown in Fig. 8.5.

In Fig. 8.5, Lines 5 and 14 are two positions in the code where some top APIs were found to be used. Lines 1–4 show rationale comments for two APIs (`IStatus`

```
 1 // USES org.eclipse.core.runtime.IStatus
 2 // INSTANTIATES org.eclipse.core.runtime.Status
 3 //     .<init> (int, java.lang.String, int, java.lang.String,
 4 //                    java.lang.Throwable)
 5  } catch (BackingStoreException e) {
 6      IStatus status = new Status(IStatus.ERROR,
 7          UIPlugin.getDefault().getBundle().getSymbolicName(),
 8          IStatus.ERROR,
 9          e.getLocalizedMessage(),
10          e);
11
12 // CALLS org.eclipse.core.runtime.ILog.log(org.eclipse.core.runtime.IStatus)
13 // CALLS org.eclipse.core.runtime.Plugin.getLog()
14 UIPlugin.getDefault().getLog().log(status);
```

Fig. 8.5 Annotated API usage example for the task of programmatically writing to eclipse workbench's log

and the constructor for class `Status`) that are used in Lines 6–10. Similarly, Lines 12 and 13 show rationale comments for two APIs that are used in Line 14.

Repository Access: This service provides access to the textual contents of three of Sourcerer's relational model elements: File, Entity, and Comment. Repository access is a simple HTTP-based Web service that returns the full text for one of the three relational model elements given their unique ids as parameters.

Dependency Slicing: This service provides dependency slices of the code entities in SourcererDB. A dependency slice of an entity is a program (collection of Java source files) that includes the entity as well as all the entities upon which it depends. Requested slices are packaged into zip files, and should be immediately compilable. The dependency slicing service can take in one or more entity ids and return a zip file containing the collection of sliced/synthesized Java files that the given set of entities depend on. The chapter by Ossher and Lopes in this volume provides an in-depth discussion of dependency slicing.

Code Search: This service implements a query processing and a code retrieval facility. Code search applications (such as CodeGenie [16, 17, 18, 19] and Sourcerer API Search) 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 using which clients can express structural information in the queries. The matching and scoring (ranking) of entities follow Lucene's implementation. Details on how Lucene matches the query terms in index fields and score the matched entities are given in [37]. In summary, a boolean retrieval is performed based on a Lucene query as described earlier in Sect. 8.4.3, then all matched entities (documents) are ranked using the TF-IDF measure [23].

Similarity Calculation: The Similarity Calculation service takes in an entity_id of an entity 'e' and returns a list of other entities that are similar to 'e'. Currently, the

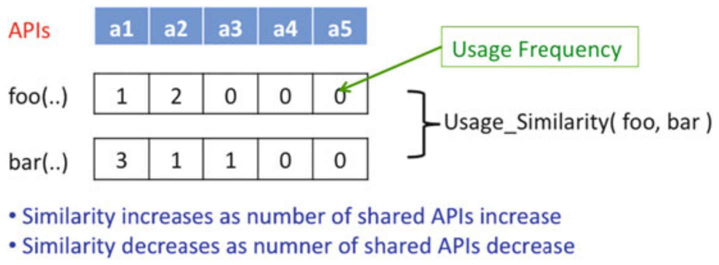

APIs
a1
a2
a3
a4
a5
Usage Frequency
foo(..)
1
2
0
0
0
bar(..)
3
1
1
0
0
Usage_Similarity(foo, bar)
• Similarity increases as number of shared APIs increase
• Similarity decreases as numner of shared APIs decrease

Fig. 8.6 Usage similarity computation based on feature vectors

similarity calculator can suggest similar entities based on three different measures of usage similarity. For this purpose, the similarity calculator uses the usage information stored in SourcererDB. The similarity calculation service works based on a feature vector representation of code entities. As shown in Fig. 8.6 for each code entity such as the methods `foo(..)` and `bar(..)` a vector representation of used APIs are stored, where each entry in the vector indicates usage frequency (could be binary for certain similarity measures). For example, Fig. 8.6 shows that `foo(..)` uses API `a1` once and API `a2` twice. Given a measure of similarity based on feature vector (for example Cosine Distance [23]), the similarity measure between two code entities `foo(..)` and `bar(..)` can be computed (Usage_Similarity(`foo(..)`, `bar(..)`)). With this collection of feature vectors, for each entity a given set of top similar entities based on API usage can be computed by choosing an appropriate similarity function that works on feature vectors. The Structural Semantic Indexing (SSI) technique makes use of the similarity calculation service and uses three different measures of similarity. Further details on similarity calculation is available in [3] and [6].

Except the Relational Query service, all other services are HTTP-based services. Currently three services are open to the public. A detailed description of how to use these services is available online [35].

8.7 Tools

A number of loosely coupled tools are available in the Sourcerer infrastructure. These tools are primarily responsible for collecting/analyzing source code and producing the stored contents.

Code Crawler: Sourcerer consists of a multithreaded plugin-based code crawler that can crawl the Web pages in online source code repositories. One of the challenges in designing the Code Crawler was to adapt with the changes and differences with Web pages in different Internet repositories. To address this challenge, the crawler follows a plugin-based design. A separate plugin can be written targeting the crawl of a repository. This makes it possible to just update the plugin (or add new plugins) when a different (or new) Web site has to be crawled. Currently the crawler consists of plugins for Sourceforge [44], Java.net [40], Tigris [45], Google Code Hosting [39], and Apache [38]. The crawler takes a set of root URLs as an input and produces a list of download URLs and version control links along with other project specific metadata. This project specific metadata is in the form as specified by (the `project.properties` file in) the storage model. Since Sourcerer only supports Java source files, the crawler uses heuristics to detect the presence of Java source files in a repository's Web page. These heuristics are common patterns specific to each repository. For example, a tag named 'Java' in a project from Google Code Hosting, and the presence of keywords such as 'java', 'eclipse', 'ant', etc. in a project from Java.net are used as indicators that a project has source code written in Java. These projects are candidates to be picked for further processing.

Repository Creator: The repository creator tool is responsible for parsing the code crawler's output list, filtering noise from the list (e.g., removing duplicate links), and downloading the contents from the online repositories to Sourcerer's local file repository. Given a local file repository's root folder, the repository creator creates the required folder structure and places the contents as specified by Sourcerer's storage model. The repository creator first creates the two level folder structure based on the number of projects it needs to add to the repository. Second, it creates the `project.properties` file describing each project. Third, it fetches the files from remote/original repositories. `project.properties` has metadata about two contents sources in remote repositories: (i) SCM systems such as svn and cvs, and (ii) downloadable packages such as compressed distributions (zips, tars, etc.). When information on a SCM repository is available, the repository creator first tries to check out contents from the respective SCM system. If errors are encountered, or if the SCM check out brings no contents, then the repository creator downloads all the packages, given that the information on links to the packages exist in `project.properties`. After the download, the repository creator explodes the archives inside the `content` folder corresponding to the project. The end result of this process is a local Sourcerer file repository, based on the storage model, which contains contents fetched from remote open source repositories.

Repository Manager: The repository manager tool is responsible for two tasks: (i) library management, and (ii) optimizing the local repository for feature extraction. Under library management, the repository manager creates and maintains a local mirror of all jar files from the Maven2 central repository. It also aggregates all of the jar files from the individual projects into the jars directory. It then creates an index of all the unique jar files in the repository. These jars can be used to provide missing types to projects in Sourcerer's file repository during feature extraction if needed. Under optimizing the local repository, the repository manager performs tasks such as compressing the contents inside a project's folder, and cleaning the jars' manifest files to avoid problems due to unexpected classpath additions.

Feature Extractor: The feature extractor in Sourcerer is responsible for extracting the detailed structural information from the source code files stored in Sourcerer's file repository. The feature extractor is built as a headless Eclipse plugin, to make use of Eclipse's (Abstract Syntax Tree) AST Parser. Before running the feature extractor, the source code is preprocessed to detect missing libraries using import statements. Some additional heuristics are used to be able to fully resolve the bindings in the source code types and links to the libraries. These heuristics are fully explained in an earlier publication [28]. The repository manager and the feature extractor together implement the required techniques for *Automated Dependency Resolution*, a key feature available in the Sourcerer infrastructure, that enables feature extraction from large number of open source projects despite missing dependencies and errors. In summary, automated dependency resolution works as follows. First, the feature extraction runs through the available projects to detect missing types. It creates the AST representation of code available in the projects and generates a list of missing types reported by the underlying Eclipse parser. From the list of missing types, the feature extractor generates a list of possible FQNs for those types to be found. It then looks up the ArtifactDB for possible jar files where the missing FQNs could be found. While doing so it selects the jar files that can provide the maximum number of missing FQNs. Once the jars are selected, they are included in the classpath of the project with missing types and then the feature extractor runs again. This process is repeated until all missing types are found or if no jars could be located for remaining missing types. After this step, the feature extraction does a full extraction of entities and relations from the projects. Our evaluation of automated dependency resolution has shown that it can increase the percentage of declaratively complete projects in Sourcerer's file repository from 39 to 69 %. Automated dependency resolution is fully explained in [28].

Database Importer: This tool allows importing the Feature Extractor's output into the code databases: ArtifactDB and SourcererDB.

Code Indexer: The code indexer tool is responsible to index all code entities in Sourcerer's file repository using the textual and structural information available for the entities. The code indexer obtains this information using three services, the File Access Service – to obtain the full text corresponding to a code entity, SourcererDB to retrieve entities and comments related to a code entity being indexed, and Similarity Calculation service to retrieve similar entities. As a result of the indexing process, the code indexer produces a semi-structured full text index based on Lucene [41]. To index a code entity, the code indexer can retrieve all or some the following data: the full-text for the corresponding entity, the fully qualified names (FQNs) of related entities, comments of the used libraries, and FQNs of used entities. The search index schema will consist of fields to store the terms corresponding to these data types. The terms are extracted from the FQNs and full text of source code documents using code-specific analysis techniques (e.g., camel case splitting and removing language keywords as stop words). The code indexer tool consists of several of these code-specific analyzers.

Code Ranker: The code ranker tool constructs a graph representation of source code analyzed in Sourcerer. Entities constitute the nodes and relations constitute the edges in the graph. After constructing this graph, code ranker applies Google's Pagerank [15] algorithm on top of this graph to compute the Pagerank (called CodeRank) for each entity which can be used as a measure of popularity of a code entity in the code graph. SCSE used the value of CodeRank as one of the heuristics to rank retrieved results.

8.8 Summary

The combination of models, services, and tools makes Sourcerer a unique infrastructure supporting three different code search applications. Going back to the requirements that were listed (in Sect. 8.2) for the three code search applications, we can summarize how Sourcerer meets these requirements.

SCSE: The storage model, stored contents, and the crawler in Sourcerer allowed collection of source code from large number of open source repositories, and store them locally making available for required further processing. The relational model and the code parser tool allowed fine grained parsing and storing parsed information in a readily available form. Being able to parse source code allowed storing and retrieving source code at the level of finer entities such as classes and methods. Using fully qualified names as keys for entities, and following relations in SourcererDB, SCSE provided a structure-based measure of CodeRank to rank code entities. As discussed in the index model, several code-specific heuristics were supported to build retrieval schemes that were specific to source code.

CodeGenie: The semi-structure index model with fields that supported retrieval using signatures provided basic retrieval for CodeGenie. Information about code entities and relations between them, allowed implementation of dependency slicing – a novel technique to extract and synthesize declaratively complete code snippet collection for CodeGenie.

SAS: Information on entities and usage (relations such as method calls and class extensions) allowed building API usage profiles for each code entities in the form of feature vectors. This served as the basis for usage similarity computation among code entities, allowing to devise novel indexing technique such as SSI using the usage similarity heuristic. Furthermore, full relational information on relations among code entities allowed computing useful API usage statistics that helped implementing useful snippet extraction technique.

The three code search applications were built one after another and Sourcerer evolved as it had to support the requirements for the applications. These requirements can be seen as major challenges that code search infrastructure builders need to address. A major lesson learnt with the implementation of three code search applications was that structural information provides valuable ways to build effective code search applications, and challenges inherent in building such applications can be overcome by harnessing large collection of source code and libraries available over the Web. Two important factors contributed to Sourcerer's success. First, a principle of leveraging structural information in source code to build effective search applications. This principle guided its design and implementation. Second, a loosely coupled architecture that made it possible for selective use of smaller set of elements across applications.

While SCSE, CodeGenie, and SAS represent three state-of-the art research prototypes for code search, Sourcerer does not address needs to develop every code search application that developers would need. For example, Sourcerer does not provide support for information related to evolution and code changes, and therefore does not support search requirements around the problems related to evolution. Also being focused solely on Java as the language of choice, Sourcerer does not provide support to search in other languages. Sourcerer does not do any form of deduplication of source code while maintaining the repository for the three code search applications. These could be some possible future improvements for Sourcerer and next generation code search infrastructures.

Sourcerer's contents as well as its implementation are freely available for others to use. The content is released as a citable dataset [21]. The implementation is available as an open source project in Github [36]. These efforts have enabled external researchers to use Sourcerer's content and services in their research [22, 24, 27, 30, 33].

8.9 Further Reading

Descriptions of earlier versions of Sourcerer are available in [2] and [20]. SCSE was first described in [1]. Code specific heuristics used in SCSE and their formal evaluation is discussed in [20] and [6]. Further details on CodeGenie is available in earlier publications [17, 18]. For details on user experiments and effectiveness evaluation of CodeGenie consult [6]. For detailed discussion on implementation and evaluation of SSI refer to [3]. More details on SAS is given in [4]. A definitive resource on details of the Sourcerer infrastructure, in particular the research contribution it made along with all three code search applications presented earlier (SCSE, CodeGenie, and SAS) is the author's doctoral dissertation [6]. A revised version of Chap. 3 from [6] appears in [5]. The chapter by Ossher and Lopes in this book provides the most recent and detailed discussion on dependency slicing that is one of the core services available in the Sourcerer infrastructure. The Software Engineering research community has produced a large body of work related to code search. A detailed review of some of these closely related to Sourcerer is available in [6] (Chap. 1). Next we summarize some of the work that focused on building code search application on top of a large-scale repository.

Merobase [14] is an infrastructure similar to Sourcerer. Like Sourcerer, Merobase has built a large code repository, a code/component search engine and a Test-Driven Search application using its repository. Merobase offers syntax aware code search, and covers additional languages (C++ and ADA). There is no documented evidence that Merobase includes structural ranking such as Sourcerer Code Search Engine's

CodeRank, or advanced indexing techniques leveraging structural similarity such as Sourcerer's SSI. Its Test-Driven Code Search application, Code Conjurer, provides a feature to do background search not present in CodeGenie (Sourcerer's TDCS application), but lacks automatic dependency slicing that allows declaratively complete program slices to be merged into a developer workspace to create self-complete code fragments satisfying the unit tests. Sourcerer also provides techniques to do deep parsing of declaratively incomplete code found in repositories; this makes Sourcerer resilient and superior in terms of extracting and leveraging structural information from source code collected from the 'wild'. The chapter by Hummel and Janjic in this volume provides an in-depth discussion of CodeConjurer.

Maracatu [9, 10] is another infrastructure built for code search. Similar to Sourcerer, it is limited to searching Java source code. The authors of Maracatu present useful requirements such as index update and optimization, but it is not clear whether Maracatu implements all of such requirements. Sourcerer does not have a proper mechanism to update its index to deal with changes in code repositories. Maracatu also supports faceted search, where the facets are platform, component type and component model. Sourcerer's index model (being based on Lucene) supports faceting out-of-the box on any metadata present in its index. However, the only faceting that has been implemented in an end-user search application is in Sourcerer API Search, where the top API elements can be used as facets to filter the code results.

S6 [29] is another Test-Driven Code Search application, that applies code transformations to convert source code found via code search into workable solutions. Parseweb [31], is another code search application that uses source and destination object types as input query to retrieve code files from existing code search engines. It applies program analysis on retrieved files to extract method sequences that work as code samples to get destination object types from source types. Applications such as S6 and Parseweb can easily benefit from code search infrastructure such as Sourcerer.

Portfolio [26] is a code search application that incorporates structural information in ranking and retrieval. One of its unique feature is to show the call graph of functions involved in the search results. Portfolio provides search access to over 18,000 C/C++ projects and 13,000 Java projects. As reported in its web site, the Java projects used in portfolio come from Sourcerer and Merobase repositories [33].

Although not a code search infrastructure, FLOSSmole [13] is another major undertaking in building large collection of metadata about open source projects on the Web. Currently, FLOSSmole reports a massive data collection of more than 500,000 open source projects in its web site [32]. For code search infrastructure builders, now it is possible to leverage FLOSSmole's project metadata to build code repositories instead of spending an effort in implementing custom spiders and crawlers for code.

Acknowledgements The author would like to thank Joel Ossher, Otavio Lemos, Trung Ngo, Huy Hunh, Paul Rigor, and Erik Linsted for their contributions to the Sourcerer infrastructure. The author would like to thank Cristina Lopes and Pierre Baldi for their advice and support in making Sourcerer successful.

References

[1] Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. pp. 681–682. ACM Press, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1176617.1176671

[2] Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An internet-scale software repository. In: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, pp. 1–4. IEEE Computer Society (2009)

[3] Bajracharya, S., Ossher, J., Lopes, C.: Leveraging usage similarity for effective retrieval of examples in code repositories. 18th International Symposium on the Foundations of Software Engineering (2010)

[4] Bajracharya, S., Ossher, J., Lopes, C.: Searching API usage examples in code repositories with sourcerer API search. In: Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation, pp. 5–8. ACM, Cape Town, South Africa (2010). DOI 10.1145/1809175.1809177

[5] Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An infrastructure for the large-scale collection and analysis of open-source code. Science of Computer Programming (To Appear) (2012)

[6] Bajracharya, S.K.: Facilitating internet-scale code retrieval. Ph.D. thesis, University of California Irvine (2010)

[7] Chen, Y., Gansner, E.R., Koutsofios, E.: A c++ data model supporting reachability analysis and dead code detection. IEEE Trans. Softw. Eng. **24**(9), 682–694 (1998)

[8] Furnas, G.W., Landauer, T.K., Gomez, L.M., Dumais, S.T.: The vocabulary problem in human-system communication. Commun. ACM **30**, 964–971 (1987). DOI 10.1145/32206.32212

[9] Garcia, V., de Almeida, E., Lisboa, L., Martins, A., Meira, S., Lucredio, D., de M. Fortes, R.: Toward a code search engine based on the State-of-Art and practice. In: Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific, pp. 61–70 (2006)

[10] Garcia, V., Lucrédio, D., Durão, F., Santos, E., de Almeida, E., de Mattos Fortes, R., de Lemos Meira, S.: From Specification to Experimentation: A Software Component Search Engine Architecture. In: I. Gorton, G. Heineman, I. Crnkovic, H. Schmidt, J. Stafford, C. Szyperski, K. Wallnau (eds.) Component-Based Software Engineering, *Lecture Notes in Computer Science*, vol. 4063, pp. 82–97. Springer Berlin / Heidelberg (2006)

[11] Gil, J.Y., Maman, I.: Micro patterns in java code. In: OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, pp. 97–116. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1094811.1094819

[12] Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The, 3 edn. Addison Wesley (2005)

[13] Howison, J., Conklin, M., Crowston, K.: FLOSSmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering **1**(3), 17–26 (2006)

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

[15] Lawrence Page Sergey Brin, R.M., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Stanford Digital Library working paper SIDL-WP-1999-0120 of 11/11/1999 (see: http://dbpubs.stanford.edu/pub/1999-66)

[16] Lemos, O.A.L., Bajracharya, S., Ossher, J., Masiero, P.C., Lopes, C.: Applying test-driven code search to the reuse of auxiliary functionality. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 476–482. ACM, Honolulu, Hawaii (2009). DOI 10.1145/1529282.1529384

[17] Lemos, O.A.L., Bajracharya, S.K., Ossher, J.: CodeGenie: a tool for test-driven source code search. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pp. 917–918. ACM, Montreal, Quebec, Canada (2007). DOI 10.1145/1297846.1297944

[18] Lemos, O.A.L., Bajracharya, S.K., Ossher, J., Masiero, P.C., Lopes, C.V.: A test-driven approach to code search and its application to the reuse of auxiliary functionality. Information and Software Technology (2011)

[19] Lemos, O.A.L., Bajracharya, S.K., Ossher, J., Morla, R.S., Masiero, P.C., Baldi, P., Lopes, C.V.: CodeGenie: using test-cases to search and reuse source code. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 525–526. ACM, Atlanta, Georgia, USA (2007)

[20] Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery **18**(2), 300–336 (2009). DOI 10.1007/s10618-008-0118-x

[21] Lopes, C., Bajracharya, S., Ossher, J., Baldi, P.: UCI source code data sets (2010). URL http://www.ics.uci.edu/~lopes/datasets/

[22] Lungu, M., Lanza, M., Nierstrasz, O.: Evolutionary and collaborative software architecture recovery with softwarenaut. In: Science of Computer Programming (SCP), (to appear) (2012)

[23] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval, 1 edn. Cambridge University Press (2008)

[24] Masuhara, H., Murakami, N., Watanabe, T.: Duplication removal for a search-based recommendation system. In: Proceedings of the 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, SUITE '12. ACM, New York, NY, USA (2012)

[25] McCandless, M., Hatcher, E., Gospodnetic, O.: Lucene in Action, 2 edn. Manning Publications (2010)

[26] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: Software Engineering (ICSE), 2011 33rd International Conference on, pp. 111–120 (2011). DOI 10.1145/1985793.1985809

[27] Murakami, N., Masuhara, H., Watanabe, T.: Optimizing a search-based code recommendation system. In: Proceedings of 3rd International Workshop on Recommendation Systems for Software Engineering, RSSE '12. ACM, New York, NY, USA (2012)
[28] Ossher, J., Bajracharya, S., Lopes, C.: Automated dependency resolution for open source software. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 130–140. Cape Town, South Africa (2010). DOI 10.1109/MSR.2010.5463346
[29] Reiss, S.P.: Semantics-based code search. In: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering - Volume 00, pp. 243–253. IEEE Computer Society (2009)
[30] Takuya, W., Masuhara, H.: A spontaneous code recommendation tool based on associative search. In: Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, SUITE '11, pp. 17–20. ACM, New York, NY, USA (2011). DOI 10.1145/1985429.1985434
[31] Thummalapenta, S., Xie, T.: 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, pp. 204–213. ACM, Atlanta, Georgia, USA (2007). 10.1145/1321631.1321663
[32] Web Page for FLOSSmole Project: *http://flossmole.org* (2012)
[33] Web Page for Portfolio: *http://www.searchportfolio.net/* (2012)
[34] Web Page for Sourcerer Project and the Sourcerer Code Search Engine: *http://sourcerer.ics.uci.edu* (2012)
[35] Web Page for Sourcerer Web Services: *http://sourcerer.ics.uci.edu/services* (2010)
[36] Web page for Sourcerer's github repository: *http://github.com/sourcerer/Sourcerer* (2010)
[37] Web Page on Apache Lucene Scoring: *http://lucene.apache.org/java/2_4_0/scoring.html* (2010)
[38] Web Site for Apache Software Foundation: *http://apache.org* (2010)
[39] Web Site for Google Code Hosting: *http://code.google.com/projecthosting* (2010)
[40] Web site for Java.net: *http://java.net* (2010)
[41] Web Site for Lucene: *http://lucene.apache.org* (2010)
[42] Web site for Maven: *http://maven.apache.org* (2010)
[43] Web Site for Maven's Central Repository: *http://repo1.maven.org/maven2/* (2010)
[44] Web Site for Sourceforge: *http://sourceforge.net* (2010)
[45] Web site for Tigris: *http://tgris.org* (2010)