

# Identifying the concepts that are searchable with keywords in code search engines

Toshihiro Kamiya

National Institute of Advanced Industrial Science and Technology  
Akihabara Dai Bldg. 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 101-0021, JAPAN  
t-kamiya@aist.go.jp

## 1 Introduction

Many code search engines, such as Codase ([www.codase.com](http://www.codase.com)), Codefetch ([www.codefetch.com](http://www.codefetch.com)), Google code search ([google.com/codesearch](http://google.com/codesearch)), JExamples ([www.jexamples.com](http://www.jexamples.com)), Koders ([www.koders.com](http://www.koders.com)), Krugle ([www.krugle.org](http://www.krugle.org)), and Merobase ([www.merobase.com](http://www.merobase.com)), have become available recently [1-6]. Most of them have Google-like interfaces through which a user can enter a set of keywords as a query to retrieve source code files that are related to the keywords in the query. Some code search engines also provide options that are specific to source code. For example, software developers can use options to specify the specific portions (such as comments, code, or functional definitions) in which the search keywords appear.

Such code search engines are important instruments to promote and support software reuse. However, their support for reuse may not be sufficient. When software developers consider reuse, they care about not only the functionality of the code, but also various characteristics such as performance (“Is the algorithm  $O(N)$  or  $O(N^2)$ ?”), usability (“Whether the API is easy to understand and use?”), and maintainability (“Can the code be easily customized to fit my code?”). This paper tries to evaluate the capabilities of keyword-based code search engines in terms of their support for searching reusable code based on multiple characteristics. The paper adopts what we call an oracle approach for the evaluation: it first identifies a classification schema that represents different dimensions of code characteristics, and then analyzes whether we are able to identify, for each dimension of characteristic, a set of intuitive keywords that can be used in a search query to retrieve effectively reusable code.

## 2 The Oracle Approach

We describe the oracle approach using a case study of searching code in the following scenario. A developer is writing a Java program that needs an array of bits with low memory consumption, that is, a class of bit array that uses one bit in the memory for each element. The developer guesses, by analogy of the class `Array` of the standard Java library, that the name of such a class may be called `BitArray`. So the developer can use `BitArray` as the functionality query for searching. However, the

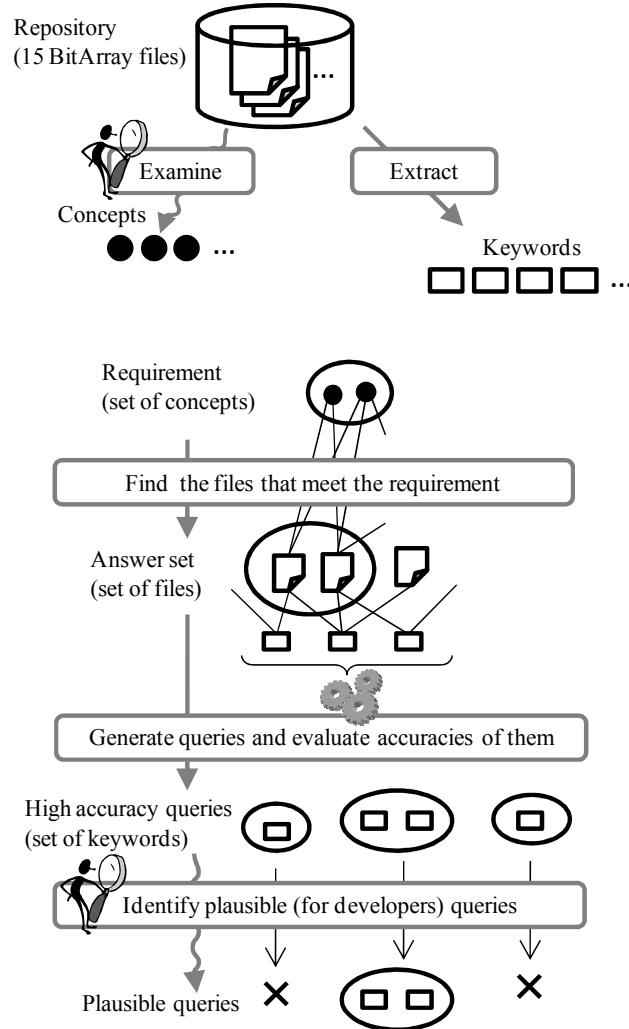


Fig. 1. Steps of the oracle approach

developer also has other requirements such as performance, comprehensibility and maintainability, and the question is what kind of keywords that he or she should use to represent such requirements for the purpose of searching.

To evaluate the oracle approach of finding effective search keywords that capture the requirements of multiple characteristics, we will use a toy keyword based code search engine that is prepared for this evaluation. The oracle approach (Fig. 1) of finding highly discriminative search keywords works as follows. For each predefined “correct” answer set of desired code that we want to find, we create a series of keyword sets that are used as search queries. Each query will return a set of search results, the search results are then measured against the predefined correct answer set.

Based on the measurement, we will find whether keywords with high discriminating power exist. More specifically, the approach consists of the following steps:

- (1) Prepare a repository of source files, which are the candidates for code search.
- (2) Examine each source file in the prepared repository and create a list of *concepts* that can be used to describe various features of source files, including basic functionality and other implementation details such as performance and potential usage pitfalls. This step needs to be performed by a subject expert.
- (3) Extract a list of keywords from each source file to represent the source file.
- (4) From the list of concepts, create a classification schema by putting each concept into different categories. This classification schema represents different dimensions of search *requirements*. A set of search requirements is created, and each search requirement contains a subset of the concepts, and in this case study, one search requirement contains one concept from each category.
- (5) For each requirement,
  - (5-1) Create an *answer set*, which is a set of source files from the repository that contain the concepts of the requirement.
  - (5-2) Determine what words can make a query that returns search results with high accuracy, namely, identifying the words that have the highest discriminative power in terms of search accuracy. To do this, we create search queries with arbitrarily selected keywords from the keyword lists, and then compare the search results of those queries with the predefined *answer set*.
  - (5-3) For each query that achieves high search accuracy, analyze whether it is plausible for a software developer to include such words in their search queries based on their search requirements. The search accuracy of a query is evaluated with precision and recall.

### 3 A Case Study

#### 3.1 Source files, concepts, and keywords

The repository in the case study contains 15 Java source files of different implementations of the `BitArray` class that are found with existing code search engines. Table 1 shows the concepts that are identified by analyzing the source files. The concepts are classified into 5 categories: *basic operation*, *scale*, *implementation issue*, *rich operation*, *conversion*, and *ease of development*. The concepts of limited-size, unlimited-size and re-size are mutually exclusive; that is, a source file can have only one concept from that category. Non-pack and pack in the implementation category are also mutually exclusive.

Keyword lists were generated from the 15 Java source files with a small script. Camel cased identifiers such as `getLength` are split into separate words (e.g. “get” and “length”). Tag names (e.g. `@author`) in JavaDoc comments were removed. Operators (such as `<<=`) were extracted as words too.

Table 1. Concepts extracted from the source files.

Concept	Classification	Description
basic	basic op.	Can store bits and retrieve the stored bits.
limited-size		The max count of bits are hard-coded in a source file (a constat).
unlimited-size	scale	Size of a bit array is specified on instance creation.
re-size		Size of a bit array can be modified with methods.
non-pack		One byte or more stroage is required to store a bit.
pack	impl.	Less than a byte is required to stroe a bit.
mask	issue	A workaround to prevent a time-consuming micro operation.
break-encap.		Has methods that return internal data (stored bits).
search		Has methods to search true bits in a bit array.
merge		Has methods to merge two bit arrays.
value	rich op.	Has a predicate for equality/comparison between two bit arrays.
shift		Has methods to shift bits in a bit array.
logical		Has methods to calculate "and" or "or" of two bit arrays.
range		Has methods to obtain or modify bits within a range on a bit array.
XML		Convertible from/to XML strings.
boolean[]	conv.	Convertible from/to a "boolean[]" object.
byte[]		Convertible from/to a "byte[]" object.
file-io		Can write to/read from a file.
copy	e. o. d.	Has methods (or constructors) to duplicate a bit array.
tostring		Has a method of a "debug" print.

*basic op.* = basic operation, *break-encap.* = break encapsulation, *conv.* = conversion, *e. o. d.* = ease of development, *impl. issue* = implementation issue, *rich op.* = rich operation.

The keywords extracted from source files were divided into equivalence classes: If keyword “a” appears in source files “f” and “g” only, and keyword “b” also appears in “f” and “g” only, then “a” and “b” are put into one equivalence class because they are equivalent in terms of their discriminative power. For each equivalence class of keywords, we need evaluate only one word from that class. In this case study, among the total 197 equivalence classes, 131 classes have only one word and the other 66 classes have an average 11.4 words, with the largest class having 236 words.

### 3.2 Overall evaluations

Table 2 shows queries with high accuracy (high precision and/or high recall) for each concept. We evaluated queries of one word, two words and three words. The queries shown in the table are queries of single word. If two or three word queries had higher precision and recall values than that of single-word queries for a particular concept, such queries are shown in the remark column in Table 2. Also, when the high discriminative queries in the second column are not intuitive ones, namely, those words seemed too difficult for a developer to guess from the concepts of the given

Table 2. High-accuracy queries for each concept

Requirement (concept)	Max prec. * recall	Max prec.	Max recall	Remark
basic	{ + } = 14/14 * 14/14	←	←	{ & } = 13/14 * 13/14
limited-size	{ supplied } = 1/1 * 1/1	←	←	{ size } = 1/13 * 1/1
scale unlimited-size	{ size } = <b>9/13 * 9/9</b>	{    } = 5/5	{ size } = 9/9	{ limitations } = 2/4 * 2/9, { ++, size } = 9/12 * 9/9, { ++, copy, size } = 9/11 * 9*9
re-size	{ initial } = 3/4 * 3/3	{ exceed } = 2/2	{ initial } = 3/3	{ size } = 2/13 * 2/3, { initial, size } = 2/3 * 2/3
non-pack	{ name } = 2/2 * 2/2	←	←	{ size } = 2/13 * 2/2, { byte } = 0/9 * 0/2, { representing } = 2/3 * 2/2
impl. issue pack	{ copy } = 12/13 * 12/12	{ ~ } = <b>10/10</b>	{ copy } = 12/12	{ packed } = 0/4 * 0/2, { & } = 12/14 * 12/12. The equivalence class of "packed" includes "\^".
mask	{ prevent } = 2/2 * 2/2	←	←	{ mask } = 2/8 * 2/2, { fast, mask } = 2/3 * 2/2
break-encap.	{ mutable } = 1/1 * 1/1	←	←	The equivalence class of "mutable" includes "corruption", "performance", and "sanity".
search	{ serialized } = 3/3 * 3/3	←	←	{ first } = 3/5 * 3/3, { pos } = 3/5 * 3/3, { position } = 2/6 * 2/3, { <=, pos } = 3/3 * 3/3
merge	{ merge } = 2/2 * 2/2	←	←	
rich op. value	{ code } = 5/5 * 5/5	←	←	{ equals } = 5/7 * 5/5, { !, * } = 5/5 * 5/5
shift	{ <<= } = 2/2 * 2/2	←	←	{ shift } = 1/3 * 1/2, { << } = 1/11 * 1/2, { >> } = 1/5 * 1/2, { >>> } = 1/5 * 1/2, { >>= } = { >>>= } = 1/1 * 1/2
logical	{ ^= } = 3/3 * 3/3	←	←	{ and } = 3/13 * 3/3, { or } = 2/5 * 2/3, { nor } = 2/2 * 2/3, { and, or } = 2/5 * 2/3
range	{ word } = 1/1 * 1/1	←	←	{ range } = 0/3 * 0/1, { bounds } = 1/7 * 1/1
XML	{ replace } = 1/1 * 1/1	←	←	{ xml } = 1/4 * 1/1, { string, xml } = 1/1 * 1/1
conv. boolean[]	{ booleans } = 3/3 * 3/3	←	←	{ boolean } = 3/7 * 3/3
byte[]	{ gets } = 4/5 * 4/4	{ reserved } = 3/3	{ gets } = 4/4	{ byte } = 4/9 * 4/4
file-io	{ input } = 6/6 * 6/6	←	←	{ file } = 4/7 * 4/6, { file, input } = 4/4 * 4/6
e. o. d. copy	{ >= } = 8/10 * 8/8	{ >> } = 5/5	{ >= } = 8/8	{ copy } = 8/13 * 8/8, { clone } = 6/6 * 6/8, { !=, >= } = 8/9 * 8/8, No three-word queries outperformed.
tostring	{ string } = 11/12 * 11/11	{ / } = 9/9	{ string } = 11/11	{ to, string } = 11/11 * 11/11

The "{...}" are queries. The values at right side of "=" are precisions and recalls. Each of these values is denoted by a fraction, whose denominator and numerator are counts of source files, without canceling down (reduction). A bold-font query is the query that looks the most intuitive one for the given requirement. A left arrow "←" means the query in the cell is the same to one in the left cell.

requirement, the more intuitive queries were shown in the remark.

Recall values in Table 2 are relatively high, and this is not surprising for this study because each concept and queries are both extracted from the source files. In other

words, it is guaranteed that some source files contain words of that concept. If the set of concepts were prepared without reading the source files, the recall values would be smaller.

Precision values varied among categories of concepts. From Table 2, we can find that the category of the ease of development have both intuitive and high-precision queries. The categories of implementation issue and rich operation, some concepts have both intuitive and high-accuracy queries while some don't. For the categories of conversion and scale, no intuitive and high-precision queries were found. For the category of basic operation, because 14 of total 15 source files have the concept, practically there is no need to query about this category.

### 3.3 Conclusions

The findings of the case study can be summarized as follows. Intuitive and high-precision queries are possible when (i) the name of the method that implement a concept is easy to guess from the coding convention of Java, such as `copy`  $\rightarrow$  `clone()`, `value`  $\rightarrow$  `equals()`, and `toString`  $\rightarrow$  `toString()`, or (ii) some unique words (operators) are required to implement the concept, such as `pack`  $\rightarrow$  `~` and `shift`  $\rightarrow$  `<<=`. On the other hand, if the concepts are implemented without unique words, such as `scale` and `conversion`, it is difficult to find intuitive and high-precision queries.

**Acknowledgements.** I am deeply grateful for Dr. Yunwen Ye for his comments, advice, and proofreading. This research is supported by JSPS Grant-in-Aid for Challenging Exploratory Research (No. 21650008).

### References

1. O. Augusto, L. Lemos, S. Bajracharya, J. Ossher, "CodeGenie: a Tool for Test-Driven Source Code Search", Proc. ASE'07 pp. 525-526 (2007).
2. S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search", Proc. OOPSLA'06: pp. 681-682 (2006).
3. K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, S. Kusumoto, "Component Rank: Relative Significance Rank for Software Component Search", Proc. ICSE'03, pp14-24, (2003).
4. D. Mandelin, L. Xu, R. Bodik, D. Kimelman, J. Mining, "Helping to Navigate the API Jungle", Proc. PLDI'05, pp.48-61 (2005).
5. Steven P. Reiss, "Semantics-Based Code Search", Proc. ICSE'09, pp. 243-253 (2009).
6. Y. Ye, G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information", Proc. ICSE'02, pp. 513-523 (2002).