In Proceedings of the 3rd International Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE 2011). Co-located with 33rd International Conference on Software Engineering (ICSE 2011), Waikiki, Honolulu, Hawaii, 2011

A Spontaneous Code Recommendation Tool Based on Associative Search

Watanabe Takuya
Graduate School of Arts and Sciences
University of Tokyo
sodium@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
masuhara@acm.org

ABSTRACT

We present Selene, a source code recommendation tool based on an associative search engine. It spontaneously searches and displays example programs while the developer is editing a program text. By using an associative search engine, it can search a repository of two million example programs within a few seconds. This paper discusses issues that are revealed by our ongoing implementation of Selene, in particular those of performance, similarity measures and user interface.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*

General Terms

Languages

Keywords

Recommender, associative search, software repository

1. INTRODUCTION

Practical software systems often exploit libraries and frameworks that consists of many modules, classes and methods, which require the developers to have deep and wide knowledge of their usage and idioms. For example, the Swing framework in Java 6 contains 633 types that the developer can access to.

We therefore believe that the developers spend considerable amount of time for finding and understanding usage and idioms of the libraries and frameworks that the system is using. For example, a developer, who is writing an application program with a graphical user interface by using the Swing framework in Java 1.4, wants to show a window with a button. Even with limited experience on Swing, the developer manages to write the following lines by remembering the names of the window and button classes.

```
JFrame w = new JFrame("a window");
JButton b = new JButton("OK");
```

Then the developer realizes that he/she does not know the ways to place the button on the window. There are several typical actions that the developer can take, each of which has problems.

Inspecting types.

The developer looks into the list of the methods defined in the directly related types (`JFrame` and `JButton` in this case) to seek for an appropriate method to add a button. Most integrated development environments (IDEs) have functions to display such information in a quick way.

However, this approach does not work well when the solution is indirect [8]. For the above example, a solution is as follows.

```
w.getContentPane().add(b); // add a button
w.pack();                  // adjust layout
```

Since calling `add` of `JFrame` is not appropriate, it is not easy to find the calling sequence from the list of methods. In addition, idioms like calling the `pack` method for adjusting the layout are rarely found.

Reading a tutorial text.

The developer reads a tutorial text of the library or framework to find a typical procedure to create a window with components.

Though it might work well for simple problems like above, finding an appropriate tutorial text and locating an appropriate description in the text would take some time. For more complicated problems, it takes more time and becomes less likely to find an appropriate description.

Searching example programs.

The developer searches a program that performs similar operations by using a search engine, and studies subsequent operations. For the above problem, an example Swing program that creates `JFrame` and `JButton` probably tells the operations that should be performed after creation of objects.

Though this approach would work well even for complicated problems, use of a search engine requires the developer's time and attention. The developer has to carefully compose a query (for example, picking the type names `JFrame` and `JButton` from the code), and has to browse search results to filter out suitable programs as results from a general-

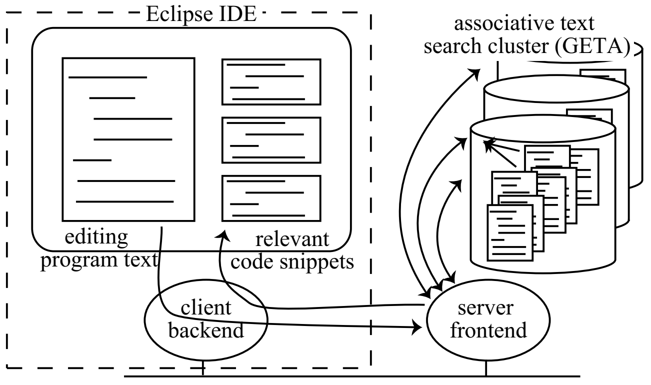

Figure 1: An overview of Selene.

purpose search engine often contain documents other than programs.

Search engines for source code, such Koders and Google Code Search, alleviate the problem, yet need careful selection of queries and manual filtering from the results. For example, the first result from Koders for "`JFrame JButton`" is a sequence of `import` declarations at the beginning of a Java program.

2. SELENE CODE RECOMMENDATION SYSTEM

2.1 Overview

Based on the above observations, we are developing a code recommendation system called Selene that (1) uses the entire editing code as a search query and (2) quickly searches and displays similar program fragments from a repository of example programs. By initiating searches automatically and by putting a large number of source programs of open-source software into the repository, we expect that Selene assists the developers to find usage and idioms of libraries and frameworks suitable to their contexts with few manual operations.

Figure 1 shows an architectural overview of Selene, whose main components are an Eclipse plug-in on the client computer and an associative search engine on the cluster server side. They work in the following steps.

1. The plug-in constantly monitors the active editor pane in Eclipse, and periodically sends the entire text to the sever.

2. The search engine searches, given the program text from the plug-in, the repository for the top N files similar to the text, and sends back pointers to those files.

3. The plug-in, after receiving the pointers of the search results, retrieves contents of the first M files from the repository. It then performs a *local search* for each retrieved file in order to identify the lines that are similar to the text around the cursor position in the editor pane.

4. The plug-in finally shows those lines of the first M retrieved files next to the editor pane.

2.2 Associative Search Engine

As the search engine for looking up similar programs, we use the GETA associative search engine [10]. Unlike keyword-based search engines, whose single search retrieves documents that contains a specified keyword, single search of GETA retrieves, given a set of keywords as a query, top N similar documents that contain similar set of keywords. Its similarity measures, such as a vector cosine similarity, do not require existence of all query keywords in the document. This property is suitable for using the entire editing code as a query.

As indexing terms of the repository and query keywords, we simply use all the alpha-numeric tokens in a program. In other words, we include not only names appearing in API specifications, but also non-public names, local variable names, language constructs, and words in comment texts.

As a similarity measure, we use the cosine similarity of term frequencies normalized by a variant of the term frequency-inverse document frequency (TF-IDF). When building a query vector, we put more weight on the occurrences of tokens near the cursor position so as to find programs that contain similar fragments to the code around the cursor position.

2.3 Repository

The repository contains 2,037,204 Java files that are obtained from the UCI Source Code Data Sets [7]. The index, which is used for associative searching, summarizes frequencies of tokens in each file. It is 4.04 GB large in total, and deployed to back-end nodes after splitting.

2.4 Client-side Plug-in

The user interface is built as an Eclipse plug-in. It consists of four components, namely (1) the one that initiates a search by monitoring the active editor pane, (2) the one that sends a query text to and receives results from the server, (3) the one that performs local search, and (4) the one that displays the result files next to the editor pane.

As for (1) search initiation, our current implementation starts searching whenever the component notices either any issue of an editor command such as cut and paste, or a jump of cursor position. In other words, it ignores minor activities such as insertion of a character and movement of a cursor by one character.

As for (3) local searching, we currently use a naive algorithm to identify the lines to be displayed. It first splits a result file received from the server into segments of 20 lines, calculates similarity score between each segment and the editing code, and finds the segment that gives the maximum score.

2.5 Performance

The search engine cluster currently consists of one front-end node with a 1.86 G Hz dual-core Intel Xeon 3040 processor and four back-end nodes, each of which has a 3.0 G Hz dual-core Intel Xeon E3110 processor with 2 GB memory.

Our experiments with the repository of two million Java files showed that the average time for a single search is 2.70 seconds with a standard deviation of 1.25 for 50 times trials. The actual latency for displaying search results will include network latency and local searches performed by the plug-in. We believe that the latency is not so different from the search time at the server when we use a quick local search algorithm, though we have not yet measured.

3. DISCUSSION

3.1 Performance and Scalability

As mentioned above, the average search time of our current system is 2.7 seconds, which we believe reasonable for interactive use. In particular, our strategy to spontaneously initiate a search makes the search latency less noticeable to the developer.

Though we are satisfied with the current response time with respect to the repository size, we should be careful with scalability of the repository. Roughly speaking, a search time by GETA is proportional to the number of the files (and the number of different indexed terms in each file). At the same time, a search time can be easily reduced by increasing the number of processor nodes. We therefore presume that availability of low-cost multi-core processors will support larger repository without difficulty.

3.2 Repository Management

When building the repository from the archives of open source projects, we found many duplicated files. Majority of those files are from common libraries bundled with project archives. We removed identical files before constructing the index.

The repository still contains files that are not identical, yet very close to each other, which are very difficult to be eliminated in advance. We observe that those are typically (1) the files in different versions of the same library, and (2) a file copied from a library, which is slightly modified for adaptation purposes (like changing the package name).

From our experience, those almost-identical files degrade search quality. When one of those files gets high score to the query, others usually have the similar score as well. Hence, the system sometimes recommends those almost-identical files at the same time, which is waste of space.

Since it is computationally expensive to remove those sets of files at the time of index construction, we plan to filter out those almost-identical files after obtained search results. An ad-hoc filtering algorithm would be sufficient for the top N files when N is small.

3.3 Similarity Measures

There are several possible directions to improve similarity measures, as we will briefly discuss below.

Associating semantic information.

A preprocess can associate semantic information to tokens in programs (both in the repository and in the editing code) before calculating similarity. For example, it can expand an abbreviated type name (e.g., `String`) to the fully qualified one (e.g., `java.lang.String`) so that the semantically same yet differently represented tokens can match each other. It can also associate static type information to occurrences of variables so that avoid matching semantically different, yet having the same name variable occurrences.

A problem of this approach is that it cannot be applied to incomplete programs, which are often the case during development. The preprocess would also be expensive for a large-scale repository.

Semantic-based weighting.

Semantic information can also be useful to give different weights on different kind of program elements. For example, names of public types and public methods could have more weights than private or local ones; tokens in comments could have less weights than those in the code; and language constructs such as `if` and `int` would have less weight than others. Even though we already use TF-IDF weighting, customized weighting that takes semantics into account might work better.

Using ontology.

When the associative engine compares tokens in the repository and in the editing code, we can use ontology-based techniques instead of simple string equality. A simple example is to split a multi-word identifier (e.g., `DirectedEdgeWithWeight`) to its constituents. The subtype relation can used for matching tokens that are semantically related. Moreover, an ontology database for text processing could match different names that refer similar concepts (e.g., `Edge` and `Arc` classes in different graph manipulation programs).

3.4 Local Search

Our experience so far suggests that quality of the local search is a crucial factor to determine overall usefulness of the tool. In fact, we found that our current algorithm often displays inappropriate lines.

We are currently investigating several algorithms for improving file local searching. One possibility is to use an algorithm to find the longest common subsequence, or to use Levenshtein distances. The algorithms that uses program structure [3, 8, 9] or ontology [1] would also be usable as the number of files is small at this point.

3.5 User Interface

Search latency and frequency.

We found that, through our experiences with different server configurations, the search latency has a dramatic impact on usefulness. Under our spontaneous search model, the search latency that the developer feels is the period from when the developer wants for help until display of search results. Because a search might have been started before the developer's desire, a noticed latency can be shorter than an actual one. On the other hand, our current initiation policy often misses an appropriate timing by ignoring minor editor activities. When the search latencies were much longer (around 10 seconds), we felt that the search results are updated almost at random timing with less related contents.

More frequent initiation of searching will reduce (subjective) latency, but could also annoy the developer. We experienced that the tool unexpectedly updates the search results while we are inspecting the results. When a fragment of a search result file displayed on the screen provides the exact information that the developer wants to know, the developer would read the contents without touching any input device, which is difficult to be distinguished from a situation when the developer is uninterested in the current search results. One possible approach is to smoothly relocate result windows by using animation like Code Bubbles [2] does.

Showing search results.

We also feel that the number of the results shown on the screen and the number of lines shown in each result are crucially important. With our preliminary experiences, we often needed to go through the results, and to scroll within a result to find a suitable example fragment. It is however difficult to increase both the number of results and the number of lines in each result, given a limited size of the screen.

One possible remedy is to show results in bubbles [2], which consume less space around the contents. Also, techniques that elide unimportant parts of the code, zooming interfaces, and contour views would be useful.

4. RELATED WORK

CCFinder is a tool that, given a set of program files, find pairs of program fragments that are almost identical [5]. Since its main usage is to find candidates of program refactoring and to detect plagiarism, its similarity measure is more strict and more computationally expensive than ours.

CodeBroker is one of the seminal systems that suggest usage of libraries and frameworks by using information in the editing code as a search query [11]. As it primarily suggests method signatures in a library, the developer needs to infer usage from documentation. The experiment in the paper used 673 classes in the repository.

Prospector [8], XSnippet [9] and RECOS [1] are tools that find a sequence of method calls to obtain an object of a type (e.g., `BufferedReader`) from another type (e.g., `InputStream`). Those tools use call graphs, code repositories, ontology-based representation and static program analysis for improving precision. They are limited to situations when the developer knows the type of an object that he or she wants to obtain and require computationally expensive algorithms. An experiment with XSnippet to find example code fragments uses a repository of 2,000 classes.

Strathcona is a tool that recommends example programs that would be useful to problems in the editing code [3]. It exploits semantic and structural information of programs in order to find program fragments that are written in contexts similar to the editing code's. The paper reports that the response time to a query varies between 4 and 12 seconds over a repository of 17,456 classes, though their implementation is claimed to be an unoptimized prototype.

Sourcerer is a code search engine with a large-scale software repository consisting of open source software project archives [6]. In addition to techniques for text-based search engines, it can use structural information for searching. Such a large-scale repository is also useful to code recommendation tools[1].

Code Conjurer is a test-based code search engine over a repository of 8 million Java files [4]. It can find class definitions that satisfy given test cases in between 3 seconds to 26 minutes, when it searches classes that exactly have the designated class and method names. When it searches by ignoring class and method names in candidate definitions, the search times go up between 47 seconds to more than 20 hours.

5. SUMMARY

Selene is a source code recommendation tool based on a text-based search engine over a source code repository. A fast parallel associative search engine allows it to use a large-scale repository, currently containing two million files with maintaining query response time within a few seconds. While it exhibits promising basic performance numbers, its effectiveness in practical software development yet to be investigated. We also pointed out several interesting research issues that we found so far, especially of performance, similarity measures, and user interface.

[1]In fact, our repository is built from the archives collected by the Sourcerer project [7].

Acknowledgement

We would like to thank Sushil Bajracharya and Cristina Lopes for their help to access the UCI Source Code Data Sets, and to thank Akihiko Tanako for his comments on the usage of the GETA search engine. This work was partly supported as a Microsoft Research CORE Project. We to thank the members of the PPP research group at the University of Tokyo for their valuable comments on the paper.

6. REFERENCES

[1] A. Alnusair, T. Zhao, and E. Bodden. Effective API navigation and reuse. In *Information Reuse and Integration (IEEE IRI)*, pp.7–12. 2010.

[2] A. Bragdon, et. al. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of International Conference on Software Engineering (ICSE'10)*, pp.455–464, 2010.

[3] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of International Conference on Software Engineering (ICSE'05)*, pp.117–125, 2005.

[4] O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25:45–52, 2008.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic tokenbased code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[6] E. Linstead, et. al. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[7] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets (SDS_source-repo-18k), Apr. 2010. `http://www.ics.uci.edu/~lopes/datasets/`.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of Programming Language Design and Implementation (PLDI'05)*, pp.48–61, 2005.

[9] N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pp.413–430, 2006.

[10] A. Takano. Association computation for information access. In *Proceedings of International Conference on Discovery Science*, LNCS 2843, pp.33–44, 2003.

[11] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, pp.513–523, 2002.