Code Recommendation Based on a Degree-of-Interest Model

Naoya Murakami
The University of Tokyo, Japan
murakami@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology, Japan
aotani@is.titech.ac.jp

ABSTRACT

Code recommendation systems predict and present what the user is likely to write next by using the user's editing context, namely textual and semantic information about the programs being edited in a programming editor. Most existing systems however use information merely around the cursor position—i.e., the class/method definition at the cursor position—as the editing context. By including the code related to the current method/class, like the callers and callees of the method, recommendation could become more appropriate. We propose to use the user's editing activity for identifying code relevant to the current method/class. Specifically, we use a modified degree-of-interest model in the Mylyn task management tool, and incorporated the model in our repository-based code recommendation system, Selene. This paper reports the design of the system and the results of our initial experiments.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated environments, Interactive environments

General Terms

Design, Human Factors, Measurement

Keywords

Keyword search, similarity

1. INTRODUCTION

Source code recommendation systems recommend fragments of sample code that are relevant to the program being edited. It is useful when a developer wants to know names and usages of unfamiliar API, for example.

Assume a developer is writing programs under the following situation as shown in Listing 1. *The developer is creating a visual text editor in Java. After created several classes, he/she is defining an* `actionPerformed` *method in the* `OpenFileMB` *class (Listing 1, top). The method corresponds to the "open file" command in the editor's menu bar. Since it needs to read texts from a file, he/she has decided to implement it in a separate method, namely the* `readFile` *method in the* `FileUtils` *class (Listing 1, bottom). After writing the first line that opens a specified file, he/she tries to remember how to read text lines from the file.*

Listing 1: Caller and Callee Classes

```
//------------------------------OpenFileMB.java
class OpenFileMB extends JMenuBar {
  ...
  void actionPerformed(ActionEvent event) {
    anEditorPane.setContentType("text/html");
    File file = chooser.getSelectedFile();
    String t = FileUtils.readFile(file);
    anEditorPane.setText(t.trim());
} }
//------------------------------FileUtils.java
class FileUtils {
  //to read internal data
  static Object readObject(File file){
    FileInputStream inFile =
      new FileInputStream(file);
    ObjectInputStream in=new ObjectInputStream(
        new BufferedInputStream(inFile));
    Object obj=in.readObject();
    ...}
  //to read text file
  static String readFile(File file){
    InputStream is = new FileInputStream(file);
    (cursor position)
```

If the developer sees the following code fragment, he or she quickly remembers the class (`BufferedReader`), the method (`readLine`) for reading lines from a text file, and the return value at the end of the file.

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(is));
  while((tmp=br.readLine()) != null){ ...
```

A code recommendation system extracts the developer's intent from the program text being edited, looks for similar program fragments in a code repository, and then displays those fragments. By using a large-scale code repository (for example 2 million files in Selene [4]), it becomes possible to recommend appropriate code fragments matching various situations that the developers are facing.

However, existing systems cannot always correctly extract an intention of the developers. Since most systems simply use the information in the file being edited, it might extract information not directly related to the current context, but

RSSE'14, June 3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2845-6/14/06...$15.00
http://dx.doi.org/10.1145/2593822.2593828

the one that happens to be in the same file.

For example, if Selene used the terms in `FileUtils.java` (Listing 1, top), it would recommend a code fragment that deserializes objects from a file, which does not match the developer's intention. This is due to the terms in the neighboring method, namely `readObject`.

If we knew the *context* of the current method being edited, we could improve recommendation. The context here means the code fragments related to the module being edited. In the above case, `actionPerformed` in `OpenFileMB` is a context of `readFile`. If Selene used the terms in `actionPerformed` as well, it could recommend a code fragment that reads text lines from a file by reflecting the related terms such as `setContentType` and `trim`.

2. RECOMMENDATION BASED ON A DEGREE-OF-INTEREST MODEL

We propose to use a *degree-of-interest* (DOI) model originally proposed by Kersten and Murphy [1] as the context information for recommendation. A DOI model keeps track of the developer's activities in the editor, and assigns each program element (e.g., method, field, function, and class) the relevance to the task that the developer is currently working on. Under the situation in the previous section, the model observes the developer's activities of selecting the `actionPerformed` method and of editing the `readFile` method, and increases relevance scores of those two methods.

We believe that the DOI model can be a better approximation of the developer's intent than the information obtained by a program analysis. This is supported by the following reasons. (1)Even when a program is not completed, the DOI model works well. On the other hand, analyzing such a program is not easy, and gives less imprecise information. (2) A program analysis would give less focused information than the DOI model as it equally treats relationships between program elements. For example, if a method being edited has many callers, it is difficult for a program analysis to discriminate the callers. By using the DOI model, the developer should have browsed only the callers that are relevant to his/her current interest. (3)Relationships between program elements that can be found by a program analysis tend to be more indirect than the ones in the developer's mind. For example, when the developer is editing a method of a GUI component, he/she often wants to know the available fields and methods in the respective model object. However, the classes of those objects are usually connected indirectly on most GUI frameworks. By using the DOI model, since the developer knows the actual class of the model object and is likely to browse it, we could get information more relevant to the developer's intention.

Based on above observations, we extended our source code recommendation system, Selene [4], to include more context information by using the DOI model. The extension consists of the following components, namely the one for monitoring editor activities, the one for keeping track of DOI, and the one for repository search using DOI. We implemented a DOI-based recommendation system by extending Selene, which uses Eclipse as an underlying development environment, GETA [3] as a search engine for relevant files, and UCI Source Code Data Sets 18k as a code repository. The monitoring and DOI calculation parts are extended from the Mylyn's implementation.

Since performance of recommendation depends on various parameters in the system, we adjusted them by using a machine learning technique. For adjusting parameters, we employed a machine learning approach used in our previous study [2], but extended to deal with the DOI model. Since our recommendation needs a history of editing activities to calculate the DOI values, we developed an Eclipse plug-in that records developers editing activities. From the collected histories, we generated a set of problems (a snapshot of program texts with DOI values) and answers (a fragment of program text inserted just after the problem point), and optimized the parameters to maximize recall rates.

3. PRELIMINARY EVALUATION

We carried out 10-fold cross-validation for evaluating our proposal. We collected development histories of 8 programs written by 3 subjects totaling to 2430 increased lines of code. We then generated ten sets of problems and answers from the collected histories. For each set, we optimize the parameters by using the rest of nine sets, applied the original Selene and the extended one with the DOI model to the selected set, and compared the recall rates of recommendations.

Our proposed approach outperformed the original one for 4 out of 10 cases. From this result, we cannot conclude advantages of the proposed approach. We however found the following interesting observations.

When the cursor jumps from one file to another, the two systems tend to give different recommendations. In particular, when the developer's context changed (i.e., he/she started writing code on a different concern), the original Selene gives better recommendations. Ours gives better ones when the concern is not changed.

When we evaluated the systems after they are configured to recommend one code fragment, our proposed approach outperforms 6 out of 10 sets with a higher average recall rate. This suggests that if we used the DOI values for the later stage of recommendation, our approach could give better recommendations.

4. CONCLUSION

We proposed an approach for a code recommendation system to better estimate the developer's intent by using a history of editing activities. The key idea is to use a modified degree-of-interest model as the context of the code being edited. We implemented our approach by extending our code repository based recommendation system, Selene. From our preliminary evaluation, we found several interesting observations where our approach can work better.

5. REFERENCES

[1] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. *AOSD '05*, pp.159–168, 2005.

[2] N. Murakami and H. Masuhara. Optimizing a search-based code recommendation system. *RSSE'12*, pp.68–72, 2012.

[3] A. Takano, et al. Information access based on associative calculation. *SOFSEM'00*, pp.15–35, 2000.

[4] T. Watanabe and H. Masuhara. A spontaneous code recommendation tool based on associative search. *(SUITE'11)*, pp.17–20, 2011.