How Often Is Necessary Code Missing? — A Controlled Experiment —

Tomoya Ishihara, Yoshiki Higo, and Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, Japan
{t-ishihr,higo,kusumoto}@ist.osaka-u.ac.jp

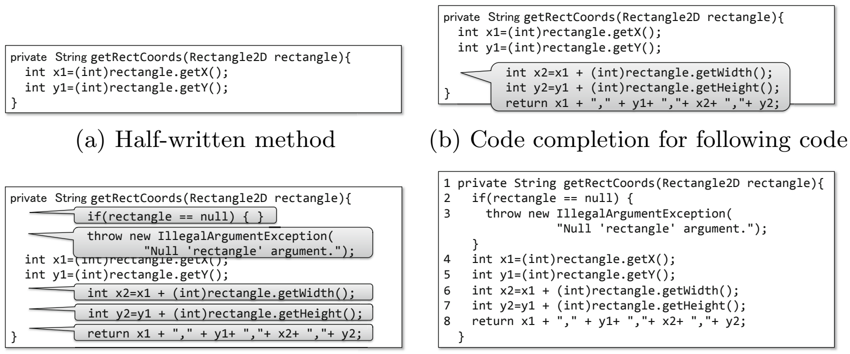
Abstract. Code completion is one of the techniques used for realizing efficient code implementations. Code completion means adding the lacking code required for finishing a given task. Recently, some researchers have proposed code completion techniques that are intended to support code reuse. However, these existing techniques are designed to support the following programming steps. They cannot add necessary code in already-implemented code lines. In this research, we first investigate how often developers forget to write the necessary code in their programming tasks. We also investigate the extent to which opportunities of code reuse are increased by considering middle code completion. To investigate middle code completion, we propose a new technique that leverages type-3 clone detection techniques. We conducted a controlled experiment with nine research participants. As a result, we found that the participants had forgotten to write the necessary code in 41 of 51 (80%) programming tasks. We also found that the proposed technique was able to suggest useful code by middle code completion in 10 of 41 (24%) programming tasks for which the participants had forgotten to write the necessary code.

Keywords: Code completion, Clone detection, Static analysis.

1 Introduction

Code completion is one of the techniques that promote efficient code implementation. Code completion adds the code that developers are going to implement by acting on their triggers. Developers do not need to write all the code they need if they use the code completion functions in their integrated development environments (IDEs). Consequently, the cost of code implementation can be reduced by using code completion appropriately. Currently, IDEs generally have their own code completion functions [1,2,3]. Many developers actually use code completion functions in software development [13].

Recently, several techniques have shown that code completion is useful for promoting code reuse [7,8,14]. If we use existing code completion techniques for code reuse, we can obtain code following the half-written code that we have. However, existing techniques do not consider complementing already-implemented code lines with the necessary code. It often happens that developers forget to



```
private  String getRectCoords(Rectangle2D rectangle){
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
}
```

(a) Half-written method

```
private  String getRectCoords(Rectangle2D rectangle){
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();

      int x2=x1 + (int)rectangle.getWidth();
      int y2=y1 + (int)rectangle.getHeight();
}     return x1 + "," + y1+ ","+ x2+ ","+ y2;
```

(b) Code completion for following code

```
private  String getRectCoords(Rectangle2D rectangle){
      if(rectangle == null) { }
      throw new IllegalArgumentException(
               "Null 'rectangle' argument.");
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
      int x2=x1 + (int)rectangle.getWidth();
      int y2=y1 + (int)rectangle.getHeight();
}     return x1 + "," + y1+ ","+ x2+ ","+ y2;
```

(c) Code completion for middle and following code

```
private String getRectCoords(Rectangle2D rectangle){
  if(rectangle == null) {
    throw new IllegalArgumentException(
              "Null 'rectangle' argument.");
  }
  int x1=(int)rectangle.getX();
  int y1=(int)rectangle.getY();
  int x2=x1 + (int)rectangle.getWidth();
  int y2=y1 + (int)rectangle.getHeight();
  return x1 + "," + y1+ ","+ x2+ ","+ y2;
}
```

(d) Base method for suggesting candidate statements to the method

Fig. 1. Examples of code completion

write the necessary code in their tasks. Consequently, if we consider the middle code in code completion, the opportunities of code completion should be increased and developers can implement more reliable code more efficiently.

In this research, we examine how often developers forget to write the necessary code in their implementations. We also investigate the extent that complementing with middle code increases code reuse. Then, we propose a new code completion technique, which can add both the middle and the following code. The proposed technique leverages type-3 clone detection techniques to identify code to be used to complement.

The aforementioned items were investigated by conducting a controlled experiment with research participants. In the experiment, all the participants were given tasks of code implementations, and their PC screens were captured as video. By using the videos, we found that the developers had forgotten to write the necessary code in 80% (41/51) tasks. We also found that the proposed technique was able to suggest reusable code in 63% (32/51) tasks and 31% (10/32) of them were for the middle code.

The remainder of this paper is organized as follows. Section 2 shows a motivating example of this research, introduces two research questions (RQs) for investigation, and explains some terms used in this paper. Section 3 explains our proposed technique. Section 4 describes the experiment conducted for answering the RQs. Lastly, Section 5 concludes this paper.

2 Preliminaries

This section plays a preliminary role in this paper. First, we explain the motivation of this research. Then, we introduce some terms used in this paper.

2.1 Motivating Example

Figure 1(a) shows a half-written method. In this case, the developer is writing code to obtain the X-Y coordinates of the upper left and lower right corners of a given rectangle and to return them in the determined form. If she/he uses existing code completion techniques (e.g., [15]), the code following the half-written code complements it as shown in Fig. 1(b). As a result, she/he can obtain two program statements that retrieve the lower right corner and another statement that translates the coordinates into the *String* form. She/he can avoid writing all the statements in the method by applying the code completion technique.

However, complementing with code following the half-written code is not sufficient to implement reliable code efficiently because it is possible that developers occasionally forget to write some necessary code. For example, developers often notice that they forgot to write error-checking code for the formal parameters of methods when they see *NullPointerException*. If code completion techniques consider not only the code following the half-written code but also the middle code in the half-written code, developers can implement more reliable code more efficiently. Figure 1(c) shows an example. If code completion techniques consider the middle code, too, we obtain null checking code for the formal parameter.

In this research, we investigate the following research questions related to code completion for the middle code.

RQ1. How often do developers forget to write the necessary code?
RQ2. To what extent is code reuse promoted by middle code completion?

2.2 Terms

In this paper, a half-written method where code completion is performed is called a **target method**, program statements suggested as completion candidates are called **candidate statements** and methods used to identify candidate statements are called **base methods**.

If we apply the definitions to Fig. 1, the method `getRectCoords` is a target method. In Figs. 1(b) and 1(c), some program statements, which are in the balloons, are suggested to the developers. These are called candidate statements. To suggest candidate statements to the developers, the proposed technique leverages the information of the other methods. In the example, the method shown in Fig. 1(d) was used. Thus, this method is called a base method.

As shown in Figs. 1(a) and 1(d), base methods include all the program statements in the half-written method. In other words, half-written code is a type-3 clone[1] of the base methods and some extra statements are in the base methods. In this research, base methods are called the **super-clones** of the target method. The proposed technique complements half-written code with both the middle and following code by detecting its super-clones.

3 Code Completion for Middle Code

In this research, we propose a new code completion technique that complements with both the middle and the following code. Figure 2 shows an overview of

[1] Bellon et al. defined type-3 clones as follows: *"type-3 is a copy with further modifications; statements were changed, added, or removed."* [4]

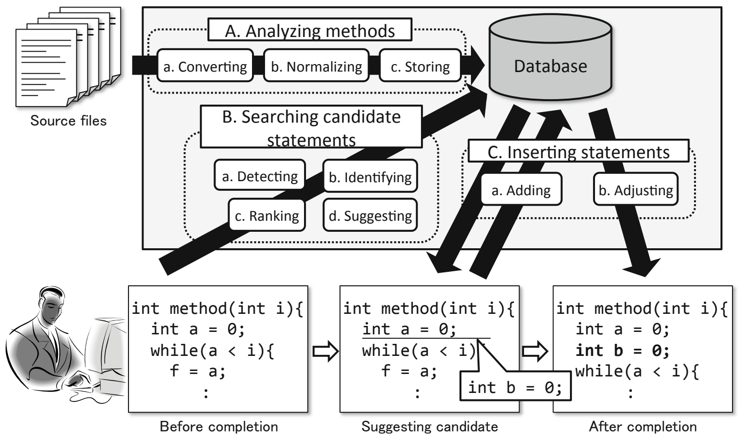


Fig. 2. Overview of the proposed technique

the proposed technique. The proposed technique extracts methods from given source files and the information obtained by analyzing the methods is stored in a database. Once the database is created, the proposed technique can suggest candidate statements for every completion position in a given half-written method. When a developer chooses a candidate statement, the statement is added at the suggested position in the half-written method. The proposed technique continues to suggest candidate statements as long as not all of the candidate statements are chosen by the developer.

As shown in Fig. 2, the proposed technique consists of the following three procedures. PROCEDURE-A extracts methods from a target set of source files and converts each of the methods into a list of statements. Next, it generates information such as a hash value from each statement and stores the information in a database. The information is used to detect super-clones.

PROCEDURE-B first obtains the information of statements included in a given target method by analyzing it. Second, it detects super-clones from the database. Third, it finds candidate statements in the super-clones (base methods). Statements of a base method are classified into two types: one is a set of statements that are common to the statements in the target method; the other is a set of statements that do not correspond to any statement in the target method. The latter statements are regarded as candidate statements. Finally, these candidate statements are ranked and suggested to the developers in the order of their rank.

PROCEDURE-C inserts a candidate statement selected by the developer. After inserting the statement, the proposed technique continues to suggest other candidate statements in the target methods. This allows developers to continue to insert multiple candidate statements as long as they want. If they do operations other than selecting suggested candidate statements, the proposed technique stops suggesting candidate statements.

To perform PROCEDURE-B based on the developer's demands, the database needs to be created by performing PROCEDURE-A in advance. PROCEDURE-A is performed only once to create the database. In contrast, PROCEDURE-B is performed every time a developer looks for candidate statements. Similarly, PROCEDURE-C is performed every time a candidate statement is selected.

4 Experiment

To provide answers to the RQs, we conducted a controlled experiment with nine research participants. In this experiment, we used *UCI source code data set*[2] for creating a database for code completion. The *UCI* dataset is a collection of open source software that is open to the public on the Web. The size of the *UCI* dataset is huge: it includes 13,192 projects, 2,127,877 Java source files, and 20,449,896 methods.

We need to identify the code that the research participants forgot to write to answer the RQs. In this experiment, forgotten code was defined as follows: a chunk of program statements written into the front or the middle of already-written program statements.

Research participants were one research associate, two PhD candidates, and six master's course students. All of them belonged to the same department as the authors. All of them had at least a half-year experience in Java programming. Their Java experiences were gained from their classes and research activities.

Each research participant was provided six tasks and she/he implemented a Java method to meet the specification. For each task, the participants were given a method signature and Javadoc comments. The method signature consisted of modifiers, a return type, a method name, parameters, and a throws clause. The Javadoc comments contained descriptions on the specification of the method.

4.1 Procedure

Firstly, the authors performed PROCEDURE-A of the proposed technique to create a database for code completion. It took approximately 2 hours to complete these operations.

Secondly, the research participants implemented Java methods according to the specifications of given tasks. They used a workstation equipped with an environment for recording motion captures. We used *CamStudio*[3] for recording the participants' screens. Windows Server 2008 R2 was installed in the workstation, and each participant logged in by using the *Remote Desktop* function.

We imposed no restrictions for the participants' implementations. They implemented Java methods as always. Some test cases had been prepared for each task. If a participant's implementation passed all the test cases, we judged that the implementation met the specification.

Thirdly, we investigated the motion pictures. In the investigation for RQ1, we watched all of each motion picture to check the order of program statement implementations. In the investigation of RQ2, we obtained the half-written code

[2] `http://www.ics.uci.edu/~lopes/datasets/`
[3] `http://camstudio.org`

from the motion pictures. Then, we input the obtained half-written code into our tool for checking whether the tool was able to suggest any code for completion and compared the suggested code and the participants' full-written code. In the investigation, half-written code was obtained every time that a new program statement was implemented.

4.2 Investigation for RQ1

Table 1 shows the participants who forgot to write the necessary code on their tasks. Each circle ("○") means there were one or more program statements that the participant forgot to write, and a hyphen ("-") means the recording of the motion picture failed for that the participant's task. The number of total tasks was 54 (6 tasks × 9 participants) but the recording motion pictures of three participant's tasks failed. We were not able to obtain any data from the failed tasks.

Table 1 shows that the participants forgot to write the necessary code in 41 out of 51 tasks. Such tasks are approximately 80% of the total number of tasks. In addition, we found the following phenomena.

– All the participants forgot to write the necessary code on at least two tasks.
– At least four participants forgot to write the necessary code on all the tasks.

Consequently, our answer to RQ1 is as follows: research participants forgot to write the necessary code in approximately 80% of the implemented methods.

4.3 Investigation for RQ2

Table 2 shows the participants and tasks for which the proposed technique was able to suggest code equal to each participant's final implementations. We judged

Table 1. Tasks and participants having forgotten code ("○" means one or more program statements were forgotten, and "-" means recording of the motion picture failed)

	T1	T2	T3	T4	T5	T6	Total
P1	-	○		○		○	3
P2	○	○	○	○	○	○	6
P3	-	○	○	○		○	4
P4	○	○	○	○	○	○	6
P5	-	○	○	○		○	4
P6	○	○	○	○	○	○	6
P7	○	○	○	○	○	○	6
P8	○	○	○	○			4
P9				○		○	2
Total	5	8	7	9	4	8	41

Table 2. Tasks and participants for which the tool was able to suggest correct candidates ("○" and "●" mean correct following code and correct middle code were suggested, respectively.)

	T1	T2	T3	T4	T5	T6	Total
P1	-		○	○	○	●○	4
P2	●○		●○	●○	○	●○	5
P3	-				○	●	2
P4	○	○		●○		●○	4
P5	-	○				○	2
P6	○					○	2
P7	●○	●○	○	○	○	○	6
P8	○					○	2
P9	○	○	○	○		○	5
Total	6	4	4	5	4	9	32

the semantically equal code to also be correct. "*Semantically equal*" means suggested code that has different syntax from the participant's final code but has the same semantics. For example, if the tool suggests "`if(i - 1 > j)`" and the participant's final code is "`if(i > j + 1)`", their semantics are the same. White circles ("○") and black circles ("●") mean the tool was able to suggest correct code following the half-written code or correct code in the middle of the half-written code, respectively.

The table shows that the tool was able to suggest correct code for 32 tasks. Those tasks are 63% (32/51) of all the participants' tasks. In addition, the tool was able to suggest correct code in the middle of the half-written code for 10 participants' tasks, which are 24% (10/41) of the tasks where participants forgot to write the necessary code. The tasks where the tool suggested the correct middle code are 31% (10/32) of the tasks where the tool suggested the correct code.

We can see that the tool was able to suggest the correct code for at least two tasks for all of the participants. Especially, for participant P7, correct candidates were suggested for all tasks. Also, for task T6, the tool was able to suggest the correct code for all of the participants.

Our answer to RQ2 is as follows: for 24% (10/41) of the users' tasks, the proposed method was able to suggest correct code for the forgotten code.

4.4 Discussion

The proposed technique suggests code included in the database, which means that code completion by the proposed technique makes code clones between the code in the database and the user's development system. The presence of code clones is said to be one of the factors that makes software maintenance more difficult. Some research studies have shown that code clones are harmless [6,10,12]. On the other hand, other studies report that only a small part of code clones are harmful to software maintenance [5,9,11]. In the proposed technique, the source code used for creating the database is very important. If we make reliable code that is well tested or executed for a long time for creating the database, the proposed technique can suggest code based on such reliable code. Clones of reliable code are not harmful because clones only become harmful when they require simultaneous modifications. Identifying all the code fragments to be modified simultaneously is a costly and error-prone task, and unintentional inconsistencies occur if we forget to modify some of these code fragments [6].

Consequently, we can say that the proposed technique promotes the generation of code clones without problems if we use reliable code for creating the database.

5 Conclusion

In this paper, we investigated the following: *(RQ1) How often developers forget to write the necessary code in their implementation tasks* and *(RQ2) To what extent* *code completion for such forgotten code is useful.* To conduct the investigation, we developed a new technique that can complement half-written code with the following code and the middle code. The investigation was conducted with nine research participants. Each participant performed six implementation tasks and their PC screens were recorded as motion pictures. By using the motion pictures, we investigated the RQs. As a result, we provided these answers: participants forgot to write the necessary code in 80% (41/51) of their tasks and the proposed technique was able to suggest correct statements in 24% (10/41) of such tasks.

Acknowledgments. This work was supported by JSPS KAKENHI Grant Numbers 25220003, 24650011, and 24680002.

References

1. Eclipse, `http://www.eclipse.org`
2. Intellij idea, `http://www.jetbrains.com/idea/`
3. Intellisense, `http://msdn.microsoft.com/en-us/library/hcw1s69b.aspx`
4. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and Evaluation of Clone Detection Tools. IEEE TSE 33(9), 577–591 (2007)
5. Göde, N., Koschke, R.: Frequency and risks of changes to clones. In: Proc. of ICSE, pp. 311–320 (2011)
6. Higo, Y., Kusumoto, S.: How Often Do Unintended Inconsistencies Happened? – Deriving Modification Patterns and Detecting Overlooked Code Fragments–. In: Proc. of ICSM, pp. 222–231 (2012)
7. Hill, R., Rideout, J.: Automatic method completion. In: Proc. of ASE, pp. 228–235 (2004)
8. Holmes, R., Walker, R.J.: Systematizing pragmatic software reuse. ACM TOSEM 21(4), 1–44 (2012)
9. Hotta, K., Sano, Y., Higo, Y., Kusumoto, S.: Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In: Proc. of ERCIM/IWPSE, pp. 73–82 (2010)
10. Inoue, K., Higo, Y., Yoshida, N., Choi, E., Kusumoto, S., Kim, K., Park, W., Lee, E.: Experience of finding inconsistently-changed bugs in code clones of mobile software. In: Proc. of IWSC, pp. 94–95 (2012)
11. Kapser, C.J., Godfrey, M.W.: "cloning considered harmful" considered harmful: Patterns of cloning in software. ESE 13(6), 645–692 (2008)
12. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE TSE, 176–192 (2006)
13. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software 23(4), 76–83 (2006)
14. Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J., Nguyen, T.N.: Graph-based pattern-oriented, context-sensitive source code completion. In: Proc. of ICSE, pp. 69–79 (2012)
15. Yamamoto, T., Yoshida, N., Higo, Y.: Seamless Code Reuse Using Source Code Corpus. In: Proc. of IWESEP, pp. 31–36 (2013)