Context-Aware Name Recommendation for Field Renaming

Chunhao Dong*
Beijing Institute of Technology
Beijing, China
dongchunhao22@bit.edu.cn

Yanjie Jiang*
Peking University
Beijing, China
yanjiejiang@pku.edu.cn

Nan Niu
University of Cincinnati
United States
nan.niu@uc.edu

Yuxia Zhang
Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Hui Liu†
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

ABSTRACT

Renaming is one of the most popular software refactorings. Although developers may know what the new name should be when they conduct a renaming, it remains valuable for refactoring tools to recommend new names automatically so that developers can simply hit `Enter` and efficiently accept the recommendation to accomplish the refactoring. Consequently, most IDEs automatically recommend new names for renaming refactorings by default. However, the recommendation made by mainstream IDEs is often incorrect. For example, the precision of `IntelliJ IDEA` in recommending names for field renamings is as low as 6.3%. To improve the accuracy, in this paper, we propose a context-aware lightweight approach (called `CARER`) to recommend new names for Java field renamings. Different from mainstream IDEs that rely heavily on initializers and data types of the to-be-renamed fields, `CARER` exploits both dynamic and static contexts of the renamings as well as naming conventions. We evaluate `CARER` on 1.1K real-world field renamings discovered from open-source applications. Our evaluation results suggest that `CARER` can significantly improve the state of the practice in recommending new names for field renamings, improving the precision from 6.30% to 61.15%, and recall from 6.30% to 41.50%. Our evaluation results also suggest that `CARER` is as efficient as `IntelliJ IDEA` is, making it suitable to be integrated into IDEs.

KEYWORDS

Refactoring, Rename, Recommendation, Context-Aware

1 INTRODUCTION

Renaming is by far the most frequently conducted refactoring [17]. Renamings could be conducted for different reasons. First, software entities should be renamed if they are named improperly. Improper names often have a significant negative impact on the readability and maintainability of the source code. Consequently, they should be replaced (renamed) with well-designed names. Another possible reason for renaming is that the roles of some software entities have changed with the evolution of the software applications, and thus they should be renamed to reflect the new roles. Notably, renaming could be applied to various software entities, and the most common renamings include method renaming, field renaming, parameter renaming, local variable renaming, and class renaming.

Most refactoring tools, including mainstream IDEs, provide automated or semi-automated tool support for renaming. Notably, renaming is not easy. It may involve complex software analysis to accurately identify all identifiers that should be replaced, and any incorrect analysis may result in syntax errors and/or semantic errors [41]. Another challenge in renaming is the recommendation of new names. Although it is likely that developers know what the new name should be when they conduct a rename refactoring, it remains valuable to recommend the new name automatically so that developers can simply click the `OK` button (or hit `Enter`) to accomplish the refactoring. However, without a full understanding of the source code and the motivation behind the renaming, it remains challenging to figure out the expected new name. For example, according to the evaluation results in Section 3.4, most (around 93%) new names recommended by `IntelliJ IDEA` for field renamings are incorrect. As a result, the new names recommended by such refactoring tools are often ignored by developers, and they have to coin and type in the new names manually. Because of the low accuracy, some refactoring tools, like `Eclipse`, abandon the functionality of new name recommendations.

To improve the accuracy of name recommendation, in this paper, we propose `CARER`, a context-aware name recommendation approach for field renamings. It leverages a sequence of context-aware heuristics to suggest names that align with the expectations of developers for the to-be-renamed Java fields. Different from existing approaches that rely heavily on the data type and initialization of the to-be-renamed field, `CARER` exploits naming conventions, dynamic contexts of the field, and static contexts of the field. Dynamic contexts of a field renaming refer to field renamings conducted recently within the enclosing project. Recent renamings are useful because the current renaming may follow the same rationale as the recent renamings. Notably, most refactoring tools (e.g., `IntelliJ IDEA` and `Eclipse`) collect all refactorings automatically as a refactoring history so that conducted refactorings can be undone. Consequently, dynamic contexts are inherently available. Static contexts of a field renaming refer to its sibling fields, its initialization, its data

*Chunhao Dong and Yanjie Jiang made equal contributions to this work.
†Corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639195

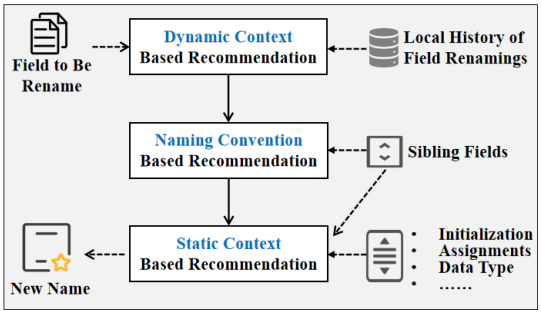

Figure 1: Overview

type, and its assignments. Such static contexts are useful because they often either contain the field name directly or imply some class-specific common structures of field names.

We evaluate CARER with 11,085 real-world field renamings Our evaluation results suggest that CARER substantially improves the state of the art and the state of the practice. Compared to the mainstream IDE IntelliJ IDEA, CARER improves precision and recall by 871% and 559%, respectively. Compared to the state-of-the-art code completion approach Incoder [20], CARER improves precision and recall by 356% and 209%, respectively. Our evaluation results also suggest that CARER is efficient, and its average response time is less than 2 milliseconds.The main contributions are as follows:

- A novel approach (CARER) to recommending new names for Java field renaming. To the best of our knowledge, it is the first approach in this line that exploits the dynamic and static contexts of the renamings as well as naming conventions.
- A public implementation [10] of the proposed approach.
- A benchmark [10] containing 11,085 real-world field renamings.

2 APPROACH

2.1 Overview

Fig. 1 presents the overview of CARER. It employs a sequence of heuristics to suggest a new name for the to-be-renamed field. Overall, it works as follows:

- First, if some field renamings have been applied to the enclosing project recently, CARER selects the most suitable renaming as a reference, analyzes its changing pattern, and applies this pattern to the existing name of the to-be-renamed field. The resulting name is recommended as the new name.
- CARER formats the name of the to-be-renamed field if it conflicts with widely-used naming conventions or project-specific naming conventions. The resulting name, if any, is recommended.
- CARER suggests a new name according to the static contexts of the to-be-renamed field, i.e., its initialization, expressions assigned to it, its data type, and methods that return the field.

2.2 Dynamic Context-based Recommendation

Dynamic contexts of a field renaming refer to recently applied field renamings conducted to the project by the current developer. Notably, most IDEs keep recognizing and recording refactorings

Algorithm 1: Dynamic Context-based Recommendation

```
   Input: f  // the field to be renamed
   Input: RS // recently applied field renamings
   Input: ConflictNames // all names visible at the class level
   Output: newName // recommended name
1  if RS = null then
2  |  return null //not applicable
3  end
4  RS' ← RANKING(RS, f)
5  foreach r ∈ RS' do
6  |  // validate if refactoring r changed naming conventions.
7  |  if Convention(f.name) = Convention(r.oldName) &
   |    Convention(r.oldName) ≠ Convention(r.newName) then
8  |  |  //apply the same change on naming convention
9  |  |  f.name ← ConvertNamingConvention(f.name,
   |  |    Convention(r.newName))
10 |  end
11 |  // infer regular expression for the change.
12 |  rm ← regularMatch(r.oldName, r.newName)
13 |  if !rm.IsEmpty() then
14 |  |  //apply the regular expression to the name.
15 |  |  newName ← rm(f.name)
16 |  end
17 |  // newName should not conflict with other fields
18 |  if newName ≠ null & newName ∉ ConflictNames then
19 |  |  return newName
20 |  end
21 end
22 return null  //fail to make recommendation
23 Ranking(RS, f):
24 |  // renaming histories applied on the same file
25 |  inFileRS=RefactoringsOnGivenFile(RS,f.file)
26 |  //renaming histories applied on other files
27 |  outSideRS=RS-inFileRS
28 |  RankByDataTypeAndDistance(inFileRS,f)
29 |  RankByDataTypeAndDistance(outSideRS,f)
30 |  //Appending refactorings in outSideRS to the end of
   |    inFileRS
31 |  return inFileRS+ outSideRS
32 end
33 RankByDataTypeAndDistance(RS, f):
34 |  // renamings on fields whose data types equal to that of f.
35 |  RS1=RefactoringsOnGivenDataType(RS,f.dataType)
36 |  //renaming refactorings on fields of other data types.
37 |  RS2=RS-RS1
38 |  //refactorings physically closer to f are ranked on the top.
39 |  RS1.RankByDistance(f)
40 |  //refactorings physically closer to f are ranked on the top.
41 |  RS2.RankByDistance(f)
42 |  // appending refactorings in RS2 to the end of RS1
43 |  return RS1+RS2
44 end
```

conducted on the given client and store the refactoring history automatically. CARER employs Algorithm 1 to recommend a new name for the to-be-renamed field f according to recent field renamings.

CARER first checks whether the renaming history is empty (Line 1 of Algorithm 1). If yes, it terminates the dynamic context-based recommendation. Otherwise, CARER ranks the recently applied field renamings (Line 4, and Lines 23-44). Renamings applied on the file where f is defined, have higher priority than others. CARER also

```
1 //Recently renamed from "sdiag_family" into "mSdiagFamily
      "
2 private final byte mSdiagFamily;
3 //to be renamed into "mSdiagProtocal"
4 private final byte sdiag_protocol;
5 //Recently renamed from "id" into "mId"
6 private final StructInetDiagSockId mId;
7  ...
```

Listing 1: Renaming Field *sdiag_protocol*

prioritizes field renamings where the renamed fields are of the same data type as f (Lines 34-35). Finally, refactorings that are physically closer to f take precedence over more distant ones (Lines 36-41).

After the ranking of refactoring histories, CARER visits each of them in turn as follows (i.e., the iteration on Lines 5-21):

- If refactoring r changed the naming convention of the associated field i.e., $Convention(r.oldName) \neq Convention(r.newName)$, and the change is applicable to f (i.e., $Convention(f.name) = Convention(r.oldName)$), CARER applies the same change (of naming conventions) to the current name of f (Lines 6-10). Currently, CARER supports the interchanges among the most common naming conventions, i.e., *CamelCase* [12], *Constant naming convention* [18], and *UnderScoreCase* [45].
- CARER infers the regular expression for the change and applies it to the original field name (Lines 11-16). CARER aligns the token sequences in *oldName* and *newName*, validates whether the renaming is to replace (or add) a prefix, to replace (or add) a suffix, or to replace (or remove) a word sequence. All such editions are recorded as a set of regular expressions and are applied to the original name of the to-be-renamed field (Line 15) if they are applicable to the name.
- CARER validates whether *newName* conflicts with other names in the scope. If not, it is recommended (Line 19). Otherwise, CARER begins the next iteration.

We take the real-world example in Listing 1 to explain Algorithm 1. In this example, the field "*sdiag_protocal*" (Line 4) should be renamed into "*mSdiagProtocal*" [4]. For this renaming, we can retrieve two renamings conducted recently within the same closing file [4], i.e., renaming "*sdiag_family*" into "*mSdiagFamily*" (noted as r_1, Line 2), and renaming "*Id*" into "*mId*" (noted as r_2, Line 6). Consequently, $RS = \{r_1\} \cup \{r_2\}$. r_1 is ranked at the top (Line 4 of Algorithm 1) because the field renamed by it has the same data type as the to-be-renamed field f (Lines 23-44). For the top renaming history r_1, the naming convention-based analysis (Lines 7-10) recognizes that r_1 has changed the naming convention from *underscore* to *CamelCase*, and thus it changes the original name "*sdiag_protocal*" (*underscore*) into "*sdiagProtocal*" (*CamelCase*). After that, the algorithm recognizes that r_1 has added the prefix "*m*" to the original name (Line 12), and thus it also adds "*m*" to "*sdiagProtocal*", making *newName* equal to "*mSdiagProtocal*". Note that adding prefixes should not break naming conventions, and thus the initial character "*s*" is capitalized when it is preceded by the prefix. The algorithm returns this name and terminates (Line 19) because this name does not conflict with any other names.

Algorithm 2: Naming Convention-based Recommendation

```
   Input: f  // the field to be renamed
   Input: FS  //all field in the class
   Input: ConflictNames  // all names visible at the class level
   Output: newName  // recommended name
1  if FS ≠ null then
2  |  HEURISTICS1(FS,f)
3  end
4  // in case Heuristic1 does not work
5  if f.newName = null then
6  |   // execute Heuristic 2 or 3 based on the modifier.
7  |  if (IsStatic(f) & IsFinal(f)) || InInterface(f) then
8  |  |  HEURISTICS2(f)
9  |  else
10 |  |  HEURISTICS3(f)
11 |  end
12 end
13 if  FS ≠ null then
14 |   //determine whether to execute Heuristic 4 or 5
15 |  if ! HEURISTICS4(FS, f) then
16 |  |  HEURISTICS5(FS, f)
17 |  end
18 end
19  // newName should not conflict with other fields
20 if  f.newName ∉ ConflictName then
21 |  return f.newName
22 else
23 |  return null // fail to make recommendation
24 end
```

2.3 Naming Convention-based Recommendation

If the old name of the to-be-renamed field does not follow widely-used naming conventions whereas other fields do follow such conventions, it is likely that the developer (who is conducting the renaming) would like to format the original name according to corresponding conventions. Based on this assumption, we propose the following heuristics to recommend new names, and the detailed algorithms are presented in Algorithm 2 and Algorithm 3.

HEURISTICS 1. *Fields within the same class should follow the same naming convention. Consequently, if the selected field does not follow the dominating naming convention,* CARER *recommends a new name by enforcing the dominating naming convention.*

The implementation of this heuristic is presented in Lines 1-9 of Algorithm 3. CARER first retrieves fields within the enclosing class that share the same modifiers (concerning *final* and *static* only) with f. CARER analyzes the dominating naming convention followed by most of the resulting fields (Lines 2-5 of Algorithm 3). If the dominating convention is any of the well-known conventions (i.e., *CamelCase*, *UnderScoreCase*, and *Constant's naming convention*) and the current name of the to-be-refactored field does not follow this dominating naming convention, CARER formats the name to make it consistent with the convention.

HEURISTICS 2. *Static and final fields, as well as fields defined in interfaces, should follow the naming convention for constants (noted as constant's naming convention). Consequently, if their names do not follow the constant's naming convention,* CARER *recommends new*

names by formatting the existing names according to the constant's naming convention.

According to the widely-used naming conventions [19], names of constants should follow the special format where all letters are capitalized and words are connected by underlines. For convenience, we call it *constant's naming convention*. This heuristics is implemented in lines 7-8 of Algorithm 2. Notably, all fields defined within an *interface* are *static* and *final* by default [45], and thus they fit the preconditions of these heuristics.

Heuristics 3. *Java fields (excluding static and final fields as well as fields within interfaces) should follow the CamelCase naming convention. Consequently, if the selected field does not follow the CamelCase naming convention,* CARER *recommends a new name for the selected field by formatting its existing name according to the CamelCase naming convention.*

CamelCase naming convention is by far the most common naming convention for Java [19]. In most cases, it is the default setting for Java identifiers (except for some special elements like static and final fields). This heuristics is implemented on Lines 15-19 of Algorithm 3. If the field to be renamed does not follow this default convention, it is likely that the developers would like to make the selected field follow this default convention. In this case, CARER should capitalize all initial characters (in the original field name) except for the initial character of the first word.

The preceding heuristics focus on the capitalization of letters and the connectors among tokens. After these heuristics, CARER takes Heuristics 4 and Heuristic 5 to recommend a new name.

Heuristics 4. *If most fields in the enclosing class contain a common prefix whereas the to-be-renamed field does not contain the common prefix,* CARER *appends the common prefix to its original name.*

It is quite often that developers would like to use some prefixes (e.g., 'f ' for fields) to identify special identifiers (like fields). If more than half of the sibling fields contain a common prefix that the current name of f does not have (Lines 20-29 of Algorithm 3), CARER appends the prefix to the current name.

Heuristics 5. *If all of its sibling fields of type $f.dataType$ share a common prefix (and/or suffix) that f does not have,* CARER *appends the common prefix (and/or suffix) to the current name f.*

Some common prefixes and suffixes are employed to highlight the data type of fields. For example, fields of type *List* often have the suffix "List". In case its sibling fields share such data type-specific prefix (and/or suffix), CARER should append such prefix (suffix) to f as well. The implementation is presented on Lines 30-44 of Algorithm 3. The method *FieldsOfGivenType(FS, f.dataType)* on Line 32 retrieves all fields in *FS* whose data types are equal to $f.dataType$. Method *mostCommonPrefix(FS')* on Line 34 finds out the most popular prefix among the fields within *FS'*, and returns the prefix as well as the number of fields (in *FS'*) that contain the prefix. Method *mostCommonSuffix(FS')* is similar to *mostCommonPrefix(FS')*, but returns the most popular suffix.

To explain Algorithm 2 and Algorithm 3, we leverage the real-world field renaming from *Alluxio* [1] (as presented in Listing 2) where the field "*fileType*" is renamed into "*mFileType*". Algorithm 2

Algorithm 3: Auxiliary Functions

```
1  HEURISTICS1(FS, f):
2      //retrieve all fields with the same modifiers as f
3      FS' ← RetrieveFields(IsStatic(f),IsFinal(f))
4      //analyze the dominating naming convention.
5      cvt=DominatingConvention(FS')
6      if cvt ∈ {Camel, underScore, Constant} & cvt ≠
         Convention(f.oldName) then
7          f.newName ← CoverNamingConvention(f, cvt)
8      end
9  end
10 HEURISTICS2(f):
11     if Convention(f.oldName) ≠ Constant then
12         f.newName ← Cover2Constant(f.oldName)
           f.oldName ← f.newName
13     end
14 end
15 HEURISTICS3(f):
16     if Convention(f.oldName) ≠ CamelCase then
17         f.newName ← Cover2CamelCase(f.oldName)
           f.oldName ← f.newName
18     end
19 end
20 HEURISTICS4(FS, f):
21     // collect the most frequent prefix
22     [prefix, frequency] = MostCommonPrefix(FS)
23     if frequency>FS.size*0.5 & !containPrefix(f.oldName,
         prefix) then
24         f.newName ← addPrefix(f.oldName, prefix)
25         return True
26     else
27         return False
28     end
29 end
30 HEURISTICS5(FS, f):
31     // collect fields with same data type as f
32     FS' ← FieldsOfGivenType(FS, f.dataType)
33     if FS' ≠ ∅ then
34         [prefix, frequency]=MostCommonPrefix(FS')
35         if frequency=FS'.size & !containPrefix(f.oldName,
             prefix) then
36             f.newName ← addPrefix(f.oldName, prefix)
               f.oldName=f.newName
37         end
38         // collect the common suffix
39         [suffix, frequency]=MostCommonSuffix(FS')
40         if frequency=FS'.size & !containSuffix( f.oldName,
             suffix) then
41             f.newName ← addSuffix(f.oldName, suffix)
42         end
43     end
44 end
```

first invokes *HEURISTICS1* (Line 2), recognizes that fields within the same class follow *CamelCase* convention, and skips the *if-statement* (Lines 6-9 of Algorithm 3) because f follows the same naming convention. Algorithm 2 skips *HEURISTICS2* (Line 8) but invokes *HEURISTICS3* (Line 10) because f is not *static* or *final*. *HEURISTICS3* fails because the old name of f does follow *CamelCase* (Line 16 of Algorithm 3). Consequently, Algorithm 2 invokes *HEURISTICS4*

```
1 public  TemporaryFolder mFolder = new TemporaryFolder();
2 //To be renamed into "mFileType"
3 public  String fileType;
4 public  Pair<PropertyKey, String> mTestConf;
5 private String mLocation;
```

Listing 2: Renaming Field *fileType*

(Line 15). The latter recognizes that all other fields within the class have a common prefix "*m*" that f does not have (Lines 22-23 of Algorithm 3). Consequently, it adds the prefix to the old name "*fileType*", updates *newName* with the resulting string (i.e., "*mFileType*") on Line 24, and returns *true*. Since *HEURISTICS4* on Line 15 returns *true*, Algorithm 2 skips Line 16, and returns *newName* on Line 21.

2.4 Static Context-based Recommendation

If neither the dynamic context-based recommendation in Algorithm 1 nor the naming convention-based recommendation succeeds, CARER leverages static contexts of the to-be-renamed field to recommend a new name as presented in Algorithm 4. Notably, the static contexts of f include the right assignments to f (e.g., $this.address = address$), the initialization of f, the data type of f, and the names of methods that contain the statement "return f".

Heuristics 6. *If field f is initialized or assigned with a method invocation,* CARER *copies the simple name of the invoked method, removes all special tokens from it (if there is any), and recommends the resulting string as the new name for f. If field f is initialized or assigned with a parameter or a field inherited from the superclass, CARER retrieves the simple name of the parameter (or field), and recommends it as the new name for f.*

The implementation of this heuristic is presented in Lines 1-8 and 23-44. If the field to be renamed has initialization, CARER calls the method *RecWithInitOrAssignment* (Line 3) to recommend the field name according to the initialization. Otherwise, it calls the method *RecWithInitOrAssignment* (Line 7) to recommend the field name according to its assignments. Notably, assignments and initializations are handled in the same way by the same method *RecWithInitOrAssignment* (Lines 23-44). Initializations are exploited first (Lines 1-4), and CARER will terminate if it succeeds in recommending names with initializations. In this case, assignments have no chance to be exploited (Line 7).

For each of the method invocations used as assignment or initialization to f, CARER retries the simple name of the method (Line 28) and calls the method *RemoveSpecialTokens* to remove special tokens from the name. If the method name begins with a verb (in its base form), *RemoveSpecialTokens* removes the verb. Note that method names often begin with verbs, e.g., "get", "split", "create", "set", and "retrieve". However, fields are rarely named with such verbs. For example, in the field declaration "*private String name=getName()*", the simple method name is "*getName*" whereas the field name is "*name*" that does not contain the verb "*get*".

If the field is initialized or assigned with a parameter or a field inherited from the superclass, CARER simply retrieves the simple name of the parameter or the field and recommends the simple name as the new name for the to-be-renamed filed f (Lines 34-41).

Algorithm 4: Static Context-based Recommendation

```
   Input: f // the field to be renamed
   Input: RSMs //right assignments assigned to f
   Input: INE //initialization of f
   Input: DT //data type of f
   Input: MNs // methods contain statement "return f"
   Input: ConflictNames // all names visible at the class level.
   Output: newName // recommended name.
1  if INE ≠ ∅ then
2  |  // recommend newName based on initialization
3  |  newName ← RecWithInitOrAssignment(INE,f)
4  end
5  // recommend newName based on right assignment
6  if newName= null & RSMs ≠ ∅ then
7  |  newName ← RecWithInitOrAssignment(RSMs,f)
8  end
9  // recommend newName based on method name
10 if newName= null & MNs ≠ ∅ then
11 |  newName ← RecWithMethodNames(MNs)
12 end
13 // recommend newName based on its data type
14 if newName= null & IsCustomType(DT) then
15 |  newName ← RecWithDataType(DT)
16 end
17 // newName should not conflict with other fields
18 if newName≠ null && newName ∉ ConflictNames then
19 |  return newName
20 else
21 |  return null // fail to make recommendation
22 end
23 RecWithInitOrAssignment(exps, f):
24 |  // traverse all assignments or initialization
25 |  foreach exp ∈ exps do
26 |  |  // method invocation as f initialization or assignment
27 |  |  if IsMethodInvocation(exp) then
28 |  |  |  name=getMethodName(exp)
   |  |  |    newName=RemoveSpecialTokens(name)
29 |  |  |  // newName should not conflict with other fields
30 |  |  |  if newName ∉ ConflictNames then
31 |  |  |  |  return newName
32 |  |  |  end
33 |  |  end
34 |  |  //parameter(superField) as f initialization(assignment)
35 |  |  if IsParameter(exp) or IsSuperField(exp) then
36 |  |  |  newName=getSimpleName(exp)
37 |  |  |  // newName should not conflict with other fields
38 |  |  |  if newName ∉ ConflictNames then
39 |  |  |  |  return newName
40 |  |  |  end
41 |  |  end
42 |  end
43 |  return null
44 end
```

Heuristics 7. *If a method returns f,* CARER *retrieves the simple name of the method, removes all special tokens from it, and returns the resulting string as the new name.*

This heuristic is implemented on Lines 9-12 where *MNs* represents all methods that contain the statement "*return f*". Method *RecWithMethodNames(MNs)* on Line 11 retrieves the first method

```
1 //to be renamed into "result"
2 private final CheckResult checkResult;
3 public LotteryTicketCheckResult(CheckResult result) {
4   checkResult = result;
5   ...
6 }
```

Listing 3: Renaming Field *checkResult*

in *MNs*, calls *RemoveSpecialTokens* to remove special tokens from the method name, and returns the resulting string as the new name.

In case CARER cannot recommend a new name according to any of the preceding heuristics, it should leverage the following heuristics to recommend a new name according to the data type of f.

Heuristics 8. *If the data type of f is a custom type and its existing name is different from the name of the data type,* CARER *returns the name of the data type as the new field name.*

The implementation of the heuristics is presented in Lines 13-16. According to our experience, developers often name fields, variables, and parameters with the names of their corresponding data types. For example, in the well-known "*openjdk/jdk*" project [3], the developer renamed the field "*srv*" to "*serverSocket*" where "*serverSocket*" is the name of its type.

We take the real-word field renaming from *LotteryTicketCheckResult.java* [2] (as presented in Listing 3) to explain Algorithm 4. In this example, a field is renamed from "*checkResult*" to "*result*". The algorithm skips the first *if-statement* (Lines 1-4) because the field f does not have any initialization. Since f has been assigned by the statement "*checkResult=result*" (Line 4 in Listing 3), Algorithm 4 invokes *RecWithInitOrAssignment* (Line 7). The latter skips the first *if-statement* (Lines 26-33) but activates the second one (Lines 34-41) because f is assigned by a parameter. It then assigns a *newName* with the simple name of the parameter (i.e., "*result*"). The algorithm skips the following two *if-statements* (Lines 9-16) because *newName* is not *null*, and returns *newName* as its recommendation (Line 19).

3 EVALUATION

3.1 Research Questions

Our evaluation investigates the following research questions:

- **RQ1:** Can CARER improve the state of the art or the state of the practice in recommending new names for field renamings?
- **RQ2:** How well do the evaluated approaches work in recommending names for different types of fields?
- **RQ3:** How accurate are the heuristics proposed in Section 2 and how often are they employed?
- **RQ4:** Is CARER efficient?
- **RQ5:** How does the heuristics overlap?
- **RQ6:** How prevalent is each heuristic in the dataset?

RQ1 investigates the performance of CARER by comparing it against the state-of-the-art approaches. We compare CARER against IntelliJ IDEA (called IDEA for short) [5], Incoder [20] and Zhang's [49]. IDEA is selected because it presents the state of the practice. IDEA is one of the most popular IDEs for Java, and it is well-known for its *intelligence*. Incoder [20] is selected because it represents the state of the art. Given a piece of code with some blanks (called masks [21]), Incoder can automatically fill in the blanks with automatically generated source code. If the occurrence of a field name is replaced with a mask, the Incoder will generate a name to replace the mask. Note that for each of the evaluated approaches, we only consider its first suggestion for each renaming because both CARER and Incoder make only a single suggestion for each refactoring. Notably, some well-known IDEs, like Eclipse and Visual Studio, are not selected for comparison although they do support refactorings. We notice that such IDEs do not suggest new names for field renamings, and thus we cannot compare them against the proposed approach. jSparrow [9] does support some special renamings of fields. However, it only renames fields whose names contain underscores or "$" by switching the naming convention to CamelCase. Consequently, it looks more like a fixing of naming conventions instead of a generic filed renaming. More than 95% of the field renamings collected in Section 3.2 are out of the scope of jSparrow. Consequently, jSparrow is not selected for comparison Zhang's [49] is selected because it represents the state of the art.

RQ2 investigates how the data types of fields may influence the performance. **RQ3** investigates the effect of the employed heuristics. CARER is essentially a sequence of heuristics, and thus it is highly valuable to figure out which heuristics are more important. **RQ4** concerns CARER's efficiency. It is critical because developers cannot tolerate any substantial delay during a rename refactoring. **RQ5** and **RQ6** concern the impact of each component on CARER and its prevalence in the data, respectively.

3.2 Dataset

To facilitate the evaluation, we collect real-world field renamings from open-source applications. We select the top 1,000 Java projects from GitHub according to their stars. Such subjects come from different domains and are maintained by different teams, and thus they have great diversity. We apply *RefactoringMiner* [42] to all of the projects to discover field renamings. *RefactoringMiner* is selected because it represents the state of the art in refactoring discovery. It reports various categories of refactorings (including field rename) by comparing two successive versions of the same applications. It should be noted that *RefactoringMiner* reports many false positives, making it impractical to use directly the raw data from *RefactoringMiner*. To minimize the false positives, we take the following measurements:

- We exclude all potential field renamings that changed the data type of the associated fields. According to our experience with *RefactoringMiner*, most false positives belong to this category. The major reason for such false positives is that *RefactoringMiner* often mismatches a field in the old version to another field in the new version although these two fields have different names and different data types.
- If a renaming renamed a field from *oldName* to *newName*, and another renaming later renamed the same field from *newName* to *oldName*, we exclude both of them.
- If a renaming (noted as R_1) renamed a field from *oldName* to *newName*, and another renaming (noted as R_2) later renamed the same field from *newName* to *anotherName*, we exclude R_1.

Finally, we successfully collected 11,085 real-world field renamings from 388 open-source applications.

Table 1: Comparison against Existing Approaches

Approaches	Precision	Recall	F1
CARER	61.15%	41.50%	49.44%
IDEA	6.30%	6.30%	6.30%
Incoder	13.41%	13.41%	13.41%
Zhang's[49]	20.54%	19.81%	20.17%

3.3 Process and Metrics

Note that CARER exploits renaming histories to make suggestions for the current renaming. Consequently, the order of the renamings could influence the results of the evaluations. However, if multiple renamings in the dataset are applied on the same commit, we cannot distinguish the order of the refactorings. To this end, we rank such renamings in the same order as they are discovered by *RefactoringMiner*. According to such an order, we apply CARER and the baseline approaches to make recommendations for each of the renamings. Since we only leverage the field renamings within the same commit as the dynamic contexts, if a commit contains only a single refactoring, the evaluation order of it does not matter. The evaluation order of refactorings belonging to different commits does not matter, either.

For each of the renamings, the recommendation (of any of the evaluated approaches) is correct if and only if the suggested name is identical to that coined by the original developers. To measure the performance of the evaluated approaches, we compute common metrics, including precision, recall, and F1. $Precision = \frac{\#Correct\ Suggestions}{\#Suggestions}$ where $\#Suggestions$ is the number of suggestions made by the evaluated approach, and $\#Correct\ Suggestions$ is the number of correct suggestions made by the approach. $Recall = \frac{\#Correct\ Suggestions}{\#Tested\ Refactorings}$ where $\#Tested\ Refactorings$ is the total number of evaluated refactorings. $F1 = \frac{2*Precision*Recall}{Precision+Recall}$.

3.4 RQ1:Improving the State of the Art

To answer RQ1, we compare CARER against the state of the art (`Incoder, Zhang's [49]`) and the state of the practice (`IntelliJ IDEA`). The evaluation results are presented in Table 1. Notably, for `IntelliJ IDEA` (and `Incoder` as well), the precision always equals recall because it makes recommendations for all field renamings. In contrast, CARER may refuse to make recommendations when it lacks confidence. As a result, its precision is likely to be greater than its recall.

From Table 1, we make the following observation:

- CARER substantially outperforms the state of the art and the state of the practice. It improves F1 by 685% = (49.44%-6.3%) / 6.3% (compared against `IntelliJ IDEA`) and 268% = (49.44%-13.41%) / 13.41% (compared against `Incoder`).
- Compared to the state of the art and the state of the practice, CARER improves both precision and recall at the same time. The improvement in precision is 871% = (61.15% - 6.3%) / 6.3% (compared against `IntelliJ IDEA`) and 356% = (61.15%-13.41%) / 13.41% (compared against `Incoder`). Whereas the improvement in recall is 559% = (41.50% - 6.30%) / 6.3%(compared against `IntelliJ IDEA`)) and 209% =(41.50% - 13.41%) / 13.4% (compared against `Incoder`).

```
/**"ClassGen.java" of openjdk/jdk
(commit:1a0ff28ea10aaba53c5fbeb59800d3bcb1d228bc)*/

//Recently renamed from "field_vec" to "fieldList"
private final List<Field> fieldList = new ArrayList<>();

//Recently renamed from "method_vec" to "methodList"
private final List<Method> methodList = new ArrayList<>()
    ;

//Recently renamed from "attribute_vec" to "attributeList
    "
private final List<Attribute> attributeList = new
    ArrayList<>();

//Selected to-be-renamed field
private final List<String> interface_vec = new ArrayList
    <>();
//Recommendation: CARER("interfaceList"),  IDEA("
    arrayList"), Incoder("methodNameList")
```

Listing 4: Typical Example Where Dynamic Contexts Help

```
/** "HybridScanBasedCommandLineTestRunner.java" of
    apache/pinot
(commit:31a6b95200cc5845706d27304fc2ed4767ec2aab)*/

private static List<String> _invIndexCols = new ArrayList
    <>(4);
...
private static String _sortedColumn;
private static String _logFieleName;
private static boolean _inCdLine = false;
private static boolean _recordScanResponses = false;
private static boolean _compareWithRspFile = true;
private static boolean _scanRspFilePath;

// Selected to-be-renamed field
private static boolean multiThreaded = true;
//Recommendation: CARER("_multiThreaded"), IDEA("aBoolean
    "), Incoder("_sortOrderAscending")
private FileWriter _scanRspFileWriter;
private LineNumberReader _scanRspFileReader;
```

Listing 5: Typical Example Where Naming Conventions Help

- CARER outperforms Zhang's approach [49] in suggesting field names. The improvement in precision and recall are 40.61% = (61.15% - 20.54%) pp and 21.69% = (41.50% - 19.81%) pp, respectively.
- The precision (61.15%) of CARER is substantially greater than its recall (41.50%). It may suggest that ignoring some challenging cases helps improve the precision of CARER.

To illustrate why CARER can substantially outperform baseline approaches, we present here some typical examples. The first example is presented in Listing 4. This example is extracted from the well-known application *openjdk/jdk*. When developers select the field *interface_vec*, and click *rename* button, `IntelliJ IDEA` recommends to rename it into "*arrayList*" according to its initialization expression (i.e., "*new ArrayList < >()* "). `Incoder` leverages a deep network to make a recommendation for the field, which results in a new name "*methodNameList*". However, both of the recommended new names are different from the final field name (i.e., "*interfaceList*") that was manually coined by the developers. CARER succeeds in recommending the new name ("*interfaceList*") in this case because it takes full advantage of the field renamings conducted

recently. We notice that the field *fieldList* on Line 5, *methodList* on Line 8, and *attributeList* on Line 11, have been renamed recently (within the same commit). CARER analyzes such renamings, and realizes that the renamings have changed the naming convention from *UnderScoreCase* [45] to *CamelCase* [12], and thus it turns the initial field name "*interface_vec*" into "*interfaceVec*". CARER also realizes that the field renamings have replaced the common suffix "*Vec*" with "*List*", and thus it further changes the field name from "*interfaceVec*" into "*interfaceList*". The latter is then recommended as the new field name, which results in a perfect recommendation.

Listing 5 presents another example to illustrate how CARER outperforms baseline approaches. The example is extracted from open-source applications *apache/pinot*. We present all static fields defined within the class *HybridScanBasedCommandLineTestRunner*. When developers select the field "*multiThreaded*" (Line 14) for renaming, CARER recommends a new name "*_multiThreaded*" that is exactly the same as what the original developers finally used. CARER succeeds because it realizes that all static fields within the enclosing class share the same common prefix "_" whereas the to-be-renamed field does not. Consequently, CARER realizes that the purpose of the renaming is to make it follow the naming convention (i.e., names of static fields should begin with "_"). As a result, it appends the prefix "_" to the original field name, which results in the recommended new name "*_multiThreaded*". In contrast, both IntelliJ IDEA and Incoder fail to make the correct recommendation for this example. IDEA recommends "*aBoolean*" because of the field's data type (Boolean) whereas Incoder recommends "*_sortOrderAscending*" (we do not exactly know the rationale for the recommendation). The new names, i.e., "*aBoolean*" and "*_sortOrderAscending*", are substantially different from the expected one (i.e., "*_multiThreaded*").

Listing 6 presents another typical example to illustrate how CARER outperforms baseline approaches. This example is extracted from the open-source application *elastic/elasticsearch*. When developers select the field "*ops*" (Line 8) for renaming, CARER recommends a new name "*interestOps*". CARER notices that the method *interestOps* (Lines 8-10) contains the statement "*return ops*". Consequently, according to Heuristics 6, it recommends to copy the method name (i.e., "*interestOps*") as the recommended field name. The recommendation is correct because the recommended new name is identical to that coined by the original developers of the program. However, both IDEA and Incoder fail in this case. IDEA recommends "*anInt*" because the data type of the field is "*int*". Incoder recommends "*readyOps*" although it conflicts with another field on Line 7.

We employ the example in Listing 7 to explain why CARER sometimes fails. In this case, the field *multiServerRegistry* on Line 7 is selected for rename. CARER recommends "*multiServerUserRegistry*" as the new name according to Heuristics 6 because this field is assigned with a new instance of *MultiServerUserRegistry* (Line 12). However, the recommended name is different from the manually coined name "*registry*", and thus the recommendation is incorrect. We also notice that the recommendation of IDEA("*multiServerUserRegistry*") and Incoder ("*muitiRegistry*") are also incorrect. The challenge here is to guess the motivation for the renaming (i.e., to shorten the field name by removing some determiners) and to figure out which tokens should be removed.

```
1  /** "TestSelectionKey.java" of   elastic/elasticsearch
2  (commit:1d3b096374a10dadc8d2bd9b2b7088ea8e1136cb)*/
3
4  // Selected to-be-renamed field
5   private int ops = 0;
6  //Recommendation: CARER("interestOps"), IDEA("anInt"),
       Incoder("readyOps")
7   private int readyOps;
8   public int interestOps() {
9      return ops;
10  }
```

Listing 6: Typical Example Where Static Contexts Help

```
1  /** "MultiServerUserRegistryTests.java" of spring-
       projects/spring-framework
2  (commit:6aa216afb6600582a36451c26427faa8e8262132)*/
3
4  private SimpUserRegistry localRegistry;
5
6  // Selected to-be-renamed field
7  private MultiServerUserRegistry multiServerRegistry;
8  //Recommendation: CARER("multiServerUserRegistry"),IDEA("
       multiServerUserRegistry"),Incoder("muitiRegistry")
9  private MessageConverter converter;
10 public void setUp() throws Exception {
11   this.localRegistry = Mockito.mock(SimpUserRegistry.
         class);
12   this.multiServerRegistry = new MultiServerUserRegistry(
         this.localRegistry);
13   this.converter = new MappingJackson2MessageConverter();
14 }
```

Listing 7: Typical Example Where CARER Fails

3.5 RQ2: Influence of Data Types

To answer RQ2, we classify the to-be-renamed fields into three categories according to their data types. The first category (noted as *primitive types*) is composed of fields whose data types are primitive types predefined by Java [6] (e.g., *int* and *Boolean*). The second category, noted as *custom types*, is composed of fields whose data types are defined by the enclosing applications or included libraries [7]. The third category (noted as *others*) is composed of fields whose data types are predefined Java reference types [8], e.g., *String*, *List*, and *Map*. Table 2 presents the performance of the evaluated approaches on different categories of fields, P and R denote precision and racall, respectively. From this table, we make the following observations:

- CARER substantially outperforms baseline approaches in each of the three categories of fields. The minimal improvement in precision and recall are 22.82% = (59.09% - 36.27%) pp and 5.58% = (41.85% - 36.27%) pp, respectively.
- The performance of the baseline approaches (i.e., IntilliJ IDEA and Incoder) is substantially better on customer types than on others (i.e., primitive types and others). One possible reason is that fields of customer types are often named with the names of their data types. As a result, all of the evaluated approaches have a good chance to make correct suggestions.

3.6 RQ3: Effects of Heuristics

To answer research question RQ3, we first investigate the effect of the three categories of heuristics, i.e., the dynamic context-based recommendation (noted as DCR for short) in Section 2.2, the naming

Table 2: Performance on Different Types of Fields

	Primitive Types		Custom Types		Others	
	P	R	P	R	P	R
CARER	55.54%	45.15%	59.09%	41.85%	64.11%	39.11%
IDEA	0.46%	0.46%	24.92%	24.92%	2.46%	2.46%
Incoder	10.24%	10.24%	36.27%	36.27%	7.41%	7.41%

Table 3: Disabling Components of CARER

Setting	Precision	Recall	F1
Default Setting	61.15%	41.50%	49.44%
Disabling DCR*	49.55%	25.59%	33.75%
Disabling NCR†	59.37%	34.40%	43.56%
Disabling SCR‡	77.34%	30.31%	43.55%

Table 4: Performance of Heuristics

	Heuristic	#Recommendation	#Correct	Precision
DCR	DCR	1,999	1,758	87.94%
NCR	H1	481	315	65.49%
	H2	300	251	83.66%
	H3	47	38	80.85%
	H4	190	79	41.58%
	H5	224	128	57.15%
SCR	H6	3,126	1,617	51.73%
	H7	456	206	45.18%
	H8	586	182	31.06%

convention-based recommendation (noted as NCR for short) in Section 2.3, and the static context-based recommendation (noted as SCR for short) in Section 2.4. To this end, we disable one of them at once, and repeat the evaluation. The evaluation results are presented in Table 3. From this table, we make the following observations:

- All of the three components of CARER are useful and indispensable. Disabling any of them results in a substantial reduction in performance. The reduction in F1 varies from 5.88 percentage points (pp) = 49.44% - 43.56% to 15.69 pp = 49.44% - 33.75%.
- Disabling the dynamic context-based recommendation (DCR) results in the greatest reduction in performance. The precision is reduced by 18.97% =(61.15% - 49.55%) / 61.15% and the reduction in recall is 38.34% = (41.50% - 25.59%) / 41.50%. As a result, it results in a substantial reduction in F1 of 31.74% = (49.44%-33.75%) / 49.44%. The results may suggest that DCR is critical for the success of CARER.
- Disabling static context-based recommendation (SCR) results in a reduction in F1 and recall although it increases the precision. It reduces F1 by 5.89 pp = 49.44%- 43.55%, and recall by 11.19 pp = 41.5%-30.31%. However, it also increases precision by 16.19 pp = 77.34%-61.15%. The results may suggest that SCR helps improve recall and F1 because it rescues many cases where DCR and NCR fail. However, SCR as a whole is less accurate (than DCR and NCR), and thus employing SCR reduces the overall precision of the proposed approach.

Since some components (e.g., SCR) are composed of multiple heuristics, we further investigate the influence of such heuristics. First, we count how many names each of the heuristics have recommended and how many of them are correct. The results are presented in Table 4. Note that we do not disable any of the components or heuristics while collecting the performance in Table 4: We simply run CARER on the data set collected in Section 3.2.

From Table 4, we make the following observations:

- Each of the heuristics is useful. All of them have made some correct recommendations, contributing to the overall recall of the proposed approach. The number of correct suggestions made by them varies from 38 (H3) to 1,758 (DCR).
- Some heuristics are more accurate than others. The precision of DCR, H2, and H3 is substantially higher than that of others. H8 (recommending the name of the field's data type as the field name) has the lowest precision (31.06%), suggesting that recommending data types as field names often results in rejection.
- DCR and H6 have the greatest numbers of correct recommendations. It may suggest that dynamic contexts, assignments, and initialization of the field contribute the most to the success of CARER.

3.7 RQ4: Efficiency

To answer RQ4, we measure how long it takes the evaluated approaches to make recommendations for a single field renaming. The evaluation results suggest that CARER is efficient. The average response time is 1.32 milliseconds. It is almost as efficient as the state-of-the-practice approach (i.e., IntelliJ IDEA) whose average response time is 1.13 milliseconds. We also notice that Incoder is much more time-consuming than CARER and IntelliJ IDEA. Its average response time is 32.62 seconds, substantially larger than those of CARER and IntelliJ IDEA (1.32 and 1.13 milliseconds, respectively).

3.8 RQ5: Overlapping

To investigate whether there is any overlap among the different components of the proposed approach and how such overlap influences the overall performance, we enabled a single component at a time. The evaluation results are presented in Table 5. From this table, we make the following observations:

- The DCR has the highest precision. That is one of the reasons why CARER prioritizes the DCR component to improve the precision of the suggested name.
- SCR component has the highest recall although it results in the lowest precision. It may suggest that SCR should be employed only if other components fail.
- The sum of the recalls in Table 5 (i.e., 44.11%=15.86%+8.95%+19.30%) is slightly higher than that (41.5%) of CARER when all components work together. It may suggest that the components do have overlap, but the overlap is rather small, confirming the great complementarity among the components in CARER.

3.9 RQ6: Prevalence

To answer research question RQ6, we evaluated the popularity of each heuristic algorithm in the dataset. Our evaluation results are presented in Table 6. From this table, we make the following observation:

Table 5: RQ5: Effect of the Heuristics When Used Separately

Working Component	Precision	Recall
DCR	87.94%	15.86%
NCR	62.94%	8.95%
SCR	40.28%	19.30%

Table 6: Prevalence of Heuristics

	Heuristic	#Prevalence	#Precision	#Recall
DCR	DCR	18.03 %	87.94%	15.86%
	H1	5.14%	69.82%	3.59%
	H2	4.04%	78.79%	3.18%
NCR	H3	10.61%	63.01%	6.68%
	H4	2.02%	49.11%	1.00%
	H5	2.30%	55.29%	1.27%
	H6	40.58%	36.85%	14.96%
SCR	H7	4.19%	43.30%	1.81%
	H8	28.12%	24.67%	6.94%

- The prevalence of different heuristics varies significantly, ranging from 2.02% to 40.58%.
- H8 and H6 have significantly higher prevalence than others. However, they also result in noticeably lower precision.

3.10 Threats to Validity

A threat to the external validity is that only a limited number of field renamings are involved in the evaluation. The limited data size may threaten the generality of the conclusions. To reduce the threat, we select top open-source Java projects from GitHub according to their #stars, and automatically discover all field renamings for the evaluation. However, refactoring discovery is time and resource-consuming, which makes it challenging to significantly increase the size of the dataset.

A threat to constructive validity is that the automated discovery of field renamings could be inaccurate. We employ the state-of-the-art refactoring miner RefactoringMiner [42] to discover field renamings by comparing two successive versions of the same applications. However, it may miss some refactorings and/or report some false positives, which may result in an inaccurate benchmark for the evaluation. To reduce the threat, we employ the state-of-the-art tool RefactoringMiner, which is evaluated to be more accurate than alternatives [40]. We also employ a sequence of heuristics (as specified in Section 3.2) to exclude false positives.

Another threat to constructive validity is that a recommendation is counted as *correct* if and only if the recommended name is identical to the one discovered in the version history. However, it is likely that the recommended name could be even better than the one in the version history, and thus the recommendation should have been taken as *correct* even if the recommended name is different from the manually coined one. However, to the best of our knowledge, there are no automated tools that can distinguish such cases automatically. It is also impractical to distinguish such cases manually because it is tedious and time-consuming.

4 RELATED WORK

4.1 Suggesting Entity Names

Considering the difficulty in naming complex software entities, a few approaches have been proposed to suggest names for software entities. For example, Kashiwabara et al. [26] utilized association rules [11] to mine the relationship between verbs in method names and identifiers in the method bodies. With such association rules, the approach recommends candidate verbs that may be used as part of the method name. Hiroshi et al. [47] proposed *Mercem* to suggest method names according to method bodies. It leverages the sets of callees as the embedding representation of a method, believing that methods with similar callees should have similar embeddings. *Mercem* recommends a method name by searching for methods whose embedding is similar to that of the given method. Liu et al. [29]proposed an automated iterative algorithm ReMapper to match software entities between two successive versions. Kurimoto et al. [27] suggest class names by employing heterogeneous graphs to quantitatively represent the properties and behaviors of classes. They first construct a graph of relationships between program elements (classes, methods, and fields) in a corpus. They then employ a deep learning network to learn the mapping between program element relationships to class names, and the trained network is used to suggest class names. Liu et al. [30] conducted empirical research to analyze potential factors related to method names. The research findings suggest that method names are closely related to project-specific contexts, in addition to the factors previously identified [28, 29, 33]. Consequently, they improve the deep-learning-based method name recommendation by exploiting such program-specific contexts. CARER differs from such approaches in that CARER focuses on field names whereas such approaches focus on method names and class names. Another difference is that CARER takes full advantage of the existing name of the to-be-renamed entity as well as field renamings conducted recently.

Recently, Zhang et al. [49] proposed a novel approach to identify should-be-renamed identifiers and to suggest new identifiers to replace the original ones. The approach exploits various information to predict whether a given identifier should be renamed. If yes, it recommends a new identifier according to the to-be-renamed identifier (noted as *odlID*) as well as the renaming histories (noted as *H*) collected from the enclosing project. The recommendation is composed of two parts. In the first part, it discovers token-level revision patterns by mining the renaming histories, and generates candidate identifiers by applying such patterns to *odlID*. In the second part, it retrieves renamings (from *H*) where the renamed identifiers are similar to *odlID*, and recommends the new identifiers in the renamings as candidate identifiers. All such candidate identifiers are sorted according to the frequencies of the applied revision patterns and the lexical similarity between *odlID* and the identifiers renamed by retrieved renaming histories. Our approach differs from Zhang's approach [49] in that they have different scopes, exploit different contexts, and employ different algorithms. First, our approach focuses on field renamings only whereas Zhang's approach considers renamings of all identifiers. Second, our approach exploits various contexts, e.g., sibling fields, naming conventions, field initialization, and data type of the field that are not exploited by Zhang's approach

in the recommendation of new names. Finally, Zhang's approach exploits all renaming histories in the enclosing project, and thus has enough data for revision pattern identification (mining), and for direct reuse of identifiers. In contrast, our approach leverages histories of field renamings conducted recently by the given client only, and thus the dataset (histories) is too small for pattern mining. Consequently, we design a sequence of fine-grained heuristics to recommend a new identifier with a single field renaming selected from the histories.

4.2 Suggesting Renaming Opportunities

It is often difficult for developers to figure out which entities should be renamed. For example, Liu et al. [33] proposed a novel approach to recommending renaming opportunities according to the consistency between method names and method bodies. It also suggests a new method name by searching a corpus of methods for method names whose corresponding method bodies are highly similar to the body of the to-be-rename method. *DeepName* [28] improved this approach by exploiting additional contexts, e.g., methods called by the current method. Another difference is that it leverages a deep neural network to generate new method names instead of reusing existing method names in corpus.

Butler et al. [14] investigated the correlation between different granularity identifiers and low-quality source code identified with static analysis. They employed medical diagnostic test techniques to identify which particular identifier naming flaws could be used as a light-weight diagnostic of potentially problematic. In addition, they subsequently proposed [15] a naming convention checking library (called NOMINAL) for Java that allows the declarative specification of conventions regarding typography and the use of abbreviations and phrases. Peruma et al. [36] presented a tool (called IDEAL) that utilizes well-known tools and libraries used for natural language and static analysis, to detect linguistic anti-patterns. Zhang et al. [48] focused on splitting of identifiers and expansion of abbreviations, and they proposed an effective and efficient identifier normalization approach (called BEQAIN) to split identifiers into their composing words and to expand the enclosed abbreviations. In addition, Jiang et al. [23–25] also explored the expansion of abbreviations, and their approaches significantly improve the state of the art. Liu et al. [16, 32] proposed a novel approach to suggest renaming opportunities and to optimize existing IDEs. Whenever a renaming is conducted by developers, it analyzes the rationale for the naming and validates whether the same renaming rationale holds for software entities that are closely related to the recently renamed entity. if yes, it recommends a new name by re-applying the renaming pattern to the original names of the recommended entities.

Our approach differs from such approaches in that it depends on developers to determine which entities (fields) should be renamed. Another difference is that it exploits contexts (e.g., sibling fields, assignments, and initialization), and naming conventions of the to-be-renamed fields that are often ignored by existing approaches.

4.3 Empirical Study on Renamings

Anthony et al. [37] explored the evolution of identifiers through an empirical study to understand the motivation for the evolution of identifiers. They found that most renaming refactorings narrowed the meaning of the identifiers. They also found that renamings of software entities often occur with the changes in data types [38]. Osumi et al. [35] investigated the relationships both co-renamed identifiers, and revealed several relationships between identifiers often found in common renaming identifiers.

Mastropaolo et al. [34] quantitatively and qualitatively evaluated the potential of three data-driven methods (i.e., *N-gram* [22], *T5* [39], and *CugLM* [31]) in supporting automated variable renaming. The results demonstrated that *CugLM* significantly outperformed other methods. However, the author also pointed out the limitations of existing generation models. To the best of our knowledge, `Incoder` [20] proposed by Fried et al. is the latest advance in this line. It is the first large-scale generative code model capable of populating arbitrary area codes, and supporting various tasks, e.g., variable re-naming, comment generation, and type inference. Different from other models, `Incoder` adopts code infilling with bidirectional context. According to their evaluation [20] , `Incoder` significantly outperforms its alternatives (such as GPT-J [44], CodeParrot [43], GPT-NeoX [13], PolyCoder [46]). This is one of the reasons why `Incoder` was selected as a baseline in our evaluation.

5 CONCLUSIONS AND FUTURE WORK

Software renaming is widely employed, and automated support from IDEs (or professional refactoring tools) is critical for its success. To this end, in this paper, we propose `CARER`, a context-aware lightweight approach to recommending a new name for the to-be-renamed field. It takes full advantage of dynamic and static contexts of the field renaming as well as naming conventions. In contrast, mainstream IDEs rely heavily on the data types and initialization of the to-be-renamed fields, ignoring most contexts exploited by `CARER`. As a result, `CARER` significantly improves the precision and recall by 356% and 209%, respectively.

We would like to investigate how the proposed approach could be adapted to renaming other software entities, like methods and classes. This paper focuses on name recommendations for field renaming because it is common, challenging, and less investigated. The name recommendation for fields often results in low performance (see Table 1), and most renaming recommendation approaches (See Section 4.1) focus on methods/classes with long implementations instead of fields that do not have any associated implementations (bodies).

The most critical future work, is to integrate the proposed approach into mainstream IDEs. We have submitted some issues and pull requests to `Eclipse` concerning the proposed approach, and some heuristics have already been merged into and distributed with the standard distribution of Eclipse. Integrating the proposed approach into mainstream IDEs (like Eclipse, IDEA, and Visual Studio) could significantly increase its impact in the industry.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers from ICSE 2024 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037) and China Postdoctoral Science Foundation (2023M740078).

REFERENCES

[1] 2023. Commit History from *Alluxio*. https://github.com/Alluxio/alluxio/commit/99ac10a1dd47176538af0d7824d97291d7fa305e

[2] 2023. Commit History from *Iluwatar*. https://github.com/iluwatar/java-design-patterns/commit/5cf2fe009bc2c04dab737d41523576c6149e605c

[3] 2023. Commit History from *OpenJDK/JDK*. https://github.com/openjdk/jdk/commit/5caa77b043ae490c3d7d56d181d0e07e6b859b9e

[4] 2023. Commit History from *platform_frameworks_base*. https://github.com/aosp-mirror/platform_frameworks_base/commit/5d62167efc4b58f7d319bd11e85c34b42d7dc6ac

[5] 2023. IntelliJ IDEA. https://www.jetbrains.com/zh-cn/idea/.

[6] 2023. Java Oracle. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

[7] 2023. Java Oracle. https://docs.oracle.com/javase/tutorial/jdbc/basics/sqlcustommapping.html.

[8] 2023. Java Oracle. https://docs.oracle.com/javase/8/docs/api/java/lang/ref/Reference.html.

[9] 2023. JSparrow. https://jsparrow.io/.

[10] 2023. Replication Package. https://github.com/DongChunHao/CARER

[11] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 207–216. https://doi.org/10.1145/170035.170072

[12] Reem S. Alsuhaibani, Christian D. Newman, Michael J. Decker, Michael L. Collard, and Jonathan I. Maletic. 2021. A Survey on Method Naming Standards: Questions and Responses Artifact. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 242–243. https://doi.org/10.1109/ICSE-Companion52605.2021.00112

[13] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *CoRR* abs/2204.06745 (2022). https://doi.org/10.48550/arXiv.2204.06745 arXiv:2204.06745

[14] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. Investigating naming convention adherence in Java references. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 41–50. https://doi.org/10.1109/ICSM.2015.7332450

[15] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas (Eds.). IEEE Computer Society, 156–165. https://doi.org/10.1109/CSMR.2010.27

[16] Xiaye Chi, Hui Liu, Guangjie Li, Weixiao Wang, Yunni Xia, Yanjie Jiang, Yuxia Zhang, and Weixing Ji. 2023. An Automated Approach to Extracting Local Variables. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023). https://api.semanticscholar.org/CorpusID:265509666

[17] Florian Deissenboeck and Markus Pizka. 2006. Concise and Consistent Naming. *Softw. Qual. J.* 14, 3 (2006), 261–282. https://doi.org/10.1007/s11219-006-9219-1

[18] Shouki A. Ebad and Danish Manzoor. 2016. An Empirical Comparison of Java and C# Programs in Following Naming Conventions. *Int. J. People Oriented Program.* 5, 1 (2016), 39–60. https://doi.org/10.4018/IJPOP.2016010103

[19] Shouki A. Ebad and Danish Manzoor. 2016. An Empirical Comparison of Java and C# Programs in Following Naming Conventions. *Int. J. People Oriented Program.* 5 (2016), 39–60.

[20] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=hQwb-lbM6EL

[21] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2980–2988. https://doi.org/10.1109/ICCV.2017.322

[22] Vincent J. Hellendoorn and Premkumar T. Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 763–773. https://doi.org/10.1145/3106237.3106290

[23] Yanjie Jiang, Hui Liu, Jiahao Jin, and Lu Zhang. 2022. Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion. *IEEE Transactions on Software Engineering* 48, 2 (2022), 519–537. https://doi.org/10.1109/TSE.2020.2995736

[24] Yanjie Jiang, Hui Liu, and Lu Zhang. 2019. Semantic Relation Based Expansion of Abbreviations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3338906.3338929

[25] Yanjie Jiang, Hui Liu, Yuxia Zhang, Nan Niu, Yuhai Zhao, and Lu Zhang. 2021. Which Abbreviations Should Be Expanded? *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 578–589. https://doi.org/10.1145/3468264.3468616

[26] Yuki Kashiwabara, Yuya Onizuka, Takashi Ishio, Yasuhiro Hayase, Tetsuo Yamamoto, and Katsuro Inoue. 2014. Recommending Verbs for Rename Method using Association Rule Mining. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, Serge Demeyer, Dave W. Binkley, and Filippo Ricca (Eds.). IEEE Computer Society, 323–327. https://doi.org/10.1109/CSMR-WCRE.2014.6747186

[27] Shintaro Kurimoto, Yasuhiro Hayase, Hiroshi Yonai, Hiroyoshi Ito, and Hiroyuki Kitagawa. 2019. Class Name Recommendation Based on Graph Embedding of Program Elements. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. IEEE, 498–505. https://doi.org/10.1109/APSEC48747.2019.00073

[28] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 574–586. https://doi.org/10.1109/ICSE43902.2021.00060

[29] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, and Yanjie Jiang. 2023. Automated Software Entity Matching Between Successive Versions. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1615–1627. https://doi.org/10.1109/ASE56229.2023.00132

[30] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1294–1306. https://doi.org/10.1145/3510003.3510154

[31] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 473–485. https://doi.org/10.1145/3324884.3416591

[32] Hui Liu, Qiurong Liu, Yang Liu, and Zhouding Wang. 2015. Identifying Renaming Opportunities by Expanding Conducted Rename Refactorings. *IEEE Trans. Software Eng.* 41, 9 (2015), 887–900. https://doi.org/10.1109/TSE.2015.2427831

[33] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[34] Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota. 2023. Automated Variable Renaming: Are We There Yet? *Empir. Softw. Eng.* 28, 2 (2023), 45. https://doi.org/10.1007/s10664-022-10274-8

[35] Yuki Osumi, Naotaka Umekawa, Hitomi Komata, and Shinpei Hayashi. 2022. Empirical Study of Co-Renamed Identifiers. In *29th Asia-Pacific Software Engineering Conference, APSEC 2022, Virtual Event, Japan, December 6-9, 2022*. IEEE, 71–80. https://doi.org/10.1109/APSEC57359.2022.00019

[36] Anthony Peruma, Venera Arnaoudova, and Christian D. Newman. 2021. IDEAL: An Open-Source Identifier Name Appraisal Tool. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*. IEEE, 599–603. https://doi.org/10.1109/ICSME52107.2021.00064

[37] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian D. Newman. 2018. An Empirical Investigation of How and Why Developers Rename Identifiers. In *Proceedings of the 2nd International Workshop on Refactoring, IWoR@ASE 2018, Montpellier, France, September 4, 2018*, Ali Ouni, Marouane Kessentini, and Mel Ó Cinnéide (Eds.). IWoR@ACM, 26–33. https://doi.org/10.1145/3242163.3242169

[38] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian D. Newman. 2020. Contextualizing Rename Decisions using Refactorings, Commit Messages, and Data Types. *J. Syst. Softw.* 169 (2020), 110704. https://doi.org/10.1016/j.jss.2020.110704

[39] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

[40] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.

[41] Andreas Thies and Christian Roth. 2010. Recommending Rename Refactorings. In *Proceedings of the 2nd International Workshop on Recommendation Systems*

for Software Engineering, RSSE 2010, Cape Town, South Africa, May 4, 2010, Reid Holmes, Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann (Eds.). ACM, 1–5. https://doi.org/10.1145/1808920.1808921

[42] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[43] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022. Natural Language Processing with Tranformers. (2022).

[44] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. (2021). https://github.com/kingoflolz/mesh-transformer-jax

[45] Yanqing Wang, Shengbin Wang, Xiaojie Li, Hang Li, and Jin Du. 2010. Identifier Naming Conventions and Software Coding Standards: A Case Study in One School of Software. In *2010 International Conference on Computational Intelligence and Software Engineering*. 1–4. https://doi.org/10.1109/CISE.2010.5676869

[46] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming, San Diego, CA, USA, 13 June 2022*, Swarat Chaudhuri and Charles Sutton (Eds.). ACM, 1–10. https://doi.org/10.1145/3520312.3534862

[47] Hiroshi Yonai, Yasuhiro Hayase, and Hiroyuki Kitagawa. 2019. Mercem: Method Name Recommendation Based on Call Graph Embedding. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019, Putrajaya, Malaysia, December 2-5, 2019*. IEEE, 134–141. https://doi.org/10.1109/APSEC48747.2019.00027

[48] Jingxuan Zhang, Siyuan Liu, Lina Gong, Haoxiang Zhang, Zhiqiu Huang, and He Jiang. 2023. BEQAIN: An Effective and Efficient Identifier Normalization Approach With BERT and the Question Answering System. *IEEE Trans. Software Eng.* 49, 4 (2023), 2597–2620. https://doi.org/10.1109/TSE.2022.3227559

[49] Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 148 (sep 2023), 51 pages. https://doi.org/10.1145/3603109