Developing a Model of Loop Actions by Mining Loop Characteristics from a Large Code Corpus

Xiaoran Wang, Lori Pollock, and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
{xiaoran, pollock, vijay}@udel.edu

***Abstract*—Some high level algorithmic steps require more than one statement to implement, but are not large enough to be a method on their own. Specifically, many algorithmic steps (e.g., count, compare pairs of elements, find the maximum) are implemented as loop structures, which lack the higher level abstraction of the action being performed, and can negatively affect both human readers and automatic tools. Additionally, in a study of 14,317 projects, we found that less than 20% of loops are documented to help readers.**

In this paper, we present a novel automatic approach to identify the high level action implemented by a given loop. We leverage the available, large source of high-quality open source projects to mine loop characteristics and develop an action identification model. We use the model and feature vectors extracted from loop code to automatically identify the high level actions implemented by loops. We have evaluated the accuracy of the loop action identification and coverage of the model over 7159 open source programs. The results show great promise for this approach to automatically insert internal comments and provide additional higher level naming for loop actions to be used by tools such as code search.

***Index Terms*—abstraction, mining code patterns, documentation generation**

I. Introduction

A method typically consists of multiple high-level algorithmic steps, where an algorithmic step is too small to be a single method, but requires more than one statement to implement. For example, a code block may "initialize a collection" or "set up a GUI" or "find the maximum in a collection." When internal comments are present, they often describe the actions of these intermediary steps of the method. Unfortunately, descriptive internal comments are rare [1], [2]. In addition, often, neither the method name nor the names in each individual statement capture the main algorithmic steps in a given method.

Thus, information at levels of abstraction between the individual statement and the whole method is not leveraged by current source code analyses, primarily because that information is not easily available beyond any internal comments describing the code blocks implementing them. Instead, current source code analyses driving software maintenance tools today treat methods as either a single unit or a set of individual statements or words. For instance, most existing concern location (or search) tools treat a method as a "bag of words" [3], i.e., a method is viewed as one document during information retrieval. Some documentation generators for method summaries process methods as a set of individual statements and then select a subset of statements for which to generate a method summary [4]. Others use methods as a "bag of words" and select a subset of words for the summary [5].

We define an **action unit** as a code block that consists of a sequence of consecutive statements that logically implement a high level action. The notion of an action unit grew out of the notion of high level actions defined and automatically identified by Sridhara et al. [6] for generating summary comments for methods. While they identified code fragments that implement high level algorithmic steps by using structural and linguistic information, the technique is limited in the kinds of action units it will identify, mostly based on a manually established set of templates for known multi-statement actions such as a loop construct for *compute max*. For example, the evaluation study indicated that only 24% of switch blocks, 40% of if-else blocks, and 15% of iterator loops implemented one of the templates. While it provides a good starting point, a more general, extensible approach is needed.

In this paper, we demonstrate the feasibility of defining a model of action unit stereotypes without manually creating templates, and then use our action identification model to automatically identify action units in projects. Our key insight is to leverage today's available large source of high-quality, open source software projects. We mine patterns of code blocks that we call stereotypes of action units (and their common characteristics) typically implemented at the statement-sequence granularity.

We start with identifying action units that are implemented by loop structures. Loops are a major part of source code, and many algorithmic steps (e.g., count, compare pairs of elements, find the maximum) are implemented as loop structures, which lack the higher level abstraction of the actual action being performed, affecting both human readers and automatic tools. Furthermore, our study of 14,317 projects revealed that internal comments on loops occur less than 20% of the time.

Specifically, to demonstrate feasibility, we focus on loops that contain exactly one conditional statement; we call these structures a loop-if structure. Although this may seem very specific, we found that 26% of loop nests in each Java

This material is based upon work supported by the National Science Foundation Grant No. CCF-1422184.

project that we analyzed are loop-if. From 14,317 open source projects extracted from GitHub [7], we counted 674,800 code fragments with this structure, indicating that each project has on average 48 such code structures. With this restricted syntax, we were able to characterize many different action units based on their syntax, word usage, data flow, and other properties of source code as described in this paper. We found that more general syntactic forms for characterizing action units do not lead to mining single actions, but rather implementing multiple actions.

Overview. Our goal in this paper is to describe the actions performed by a large set of loops. Our hypothesis is that we can automatically identify a few characteristics of a loop that taken together can be used to group loops that perform the same high level action. Consider the following loop which is extracted from an open source project:

```
while (i.hasNext()) {
  I_CmsWidget widget=(I_CmsWidget)i.next();
  if ((widget instanceof CmsCalendarWidget) && !(widget
       instanceof CmsSerialDateWidget)) {
    hasCalendarIncludes=true;
    break;
} }
```

The loop's high level action is to check if there is a certain type of element within a collection. We believe such loops can be characterized by (i) an if statement that checks if elements in a collection satisfy a condition and (ii) by breaking the control flow when such an element is found. The presence or absence of such characteristics of a loop can be captured as feature-value pairs.

Hence, a loop can be represented by a set or sequence of feature values. We call these loop feature vectors. The feature values are stated in terms of structural and data flow elements and linguistic characteristics of the loop. Our hypothesis is that a single feature vector represents many different loops, all of which perform the same action, i.e., form a loop stereotype.

The bulk of our work has focused on developing a model that can associate actions with loops based on their loop feature vectors. We call this the *action identification model.* Given the action identification model, the action for any loop-if in a project can be determined by the process illustrated in Figure 1. The source code representation of the loop-if is analyzed to extract its representative feature vector. The action identification model is then referenced to determine the high level action associated with the loop's feature vector.

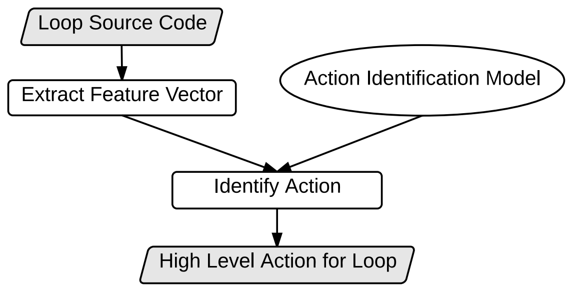


Fig. 1: *Action Identification Process*

Contributions. The main contributions of the paper include:

- a novel automatic technique to identify the high level actions implemented by Java loops based on their structure, data flow and linguistic characteristics,
- demonstration of the feasibility of characterizing loops in terms of certain data flow, structural, and linguistic features learned from a large corpus of open source code. In our model development, this enabled clustering of various loop structures that perform the same action.
- the notion that the actual high level action verb to associate with loop stereotypes can be determined from developers' internal comments on loops and that the distribution of the verbs can be used to cluster the loops into groups that perform similar actions, and
- evaluation results from human judgment studies that indicate strong positive overall opinion of our approach's effectiveness in automatically identifying high level actions for these loop structures.

Automatic identification of the high level actions implemented by code segments such as loop-if has several potential applications. Beyond an approach to generating internal comments for loop structures, tools that rely on the words in the source code and comments for analysis will benefit when the high level action words do not already appear in the loop code (e.g., search tools, comment generator tools). Another application is to help blind programmers grasp a quick high level view of code segments that otherwise are tedious and difficult to understand [8].

II. Characterizing Loops as Feature Vectors

In this section, we describe the features we will extract from a loop and their potential values. We begin with terminology that we use throughout the description.

A. Targeted Loops and Terminology

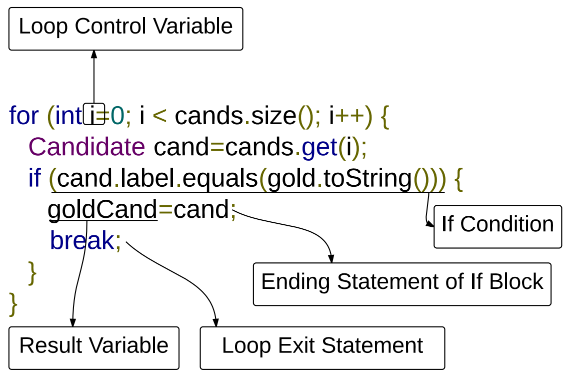
In this paper, our target is a Java loop that uses any one of the loop formats in Java: for, enhanced-for, while or do-while. We require that these loops have a single if-statement that is also the last lexical statement within the loop body. Such restrictions enable us to focus on identification of a single action. Currently, we also do not consider any nested loops. We call such a loop a **loop-if**. We collected all loops from 14,317 open source projects [7] and obtained almost 1.3 million loops. An automatic analysis of these loops revealed that 26% of them fit our loop-if criteria. In the rest of this paper, we use loop and loop-if interchangeably to describe features and the model development process.

We analyzed a large number of loop-ifs and identified all of the available features that can possibly determine the action of a loop, such as structure, data flow, and names. We use the following terminology to describe the features that are used to determine the loop actions.

- **If condition**. The if condition refers to the conditional expression in the if statement of the loop. Typically, the if condition in a loop-if is executed to examine each element of a collection for the condition in the if.

- **Loop exit statement**. Loop exit statements transfer control to another point in the code by exiting when control reaches the loop exit statement, such as a break or return. Since they affect the number of iterations that are executed, we are interested in the existence of the branching statements "break", "return", and "throw" in characterizing loop-ifs.
- **Ending statement of if block**. Since the last statement inside a loop-if is an if, the last executed statement of the loop is the last statement of either the then or else block of the if statement. We are interested in the last statement on the branch that is most frequently executed, thus we approximate this as the then block unless the if condition is the null case, in which case, we will identify the last statement of the else block as the ending statement of the if block. Sometimes the last executed statement of the loop is a branching statement (break, return or throw). In this case, the ending statement is designated to be the statement immediately preceding the branching statement. For the remainder of the paper, we use ending statement to refer to the ending statement of the if block.
- **Loop control variable**. The loop condition determines the maximum number of iterations that will be executed. In `while`, `do`, and `for` loops, the loop control variable is the variable defined in the loop condition. For enhanced-for loops, the loop control variable is each element in a given collection.
- **Result variable**. The intent of the result variable is to capture the resulting value of the loop's action (if one exists). We look for the result variable in the ending statement. If the ending statement is an assignment, the result variable is the left-hand-side variable. If the ending statement is an object method invocation, it is the object that invokes the method.

Figure 2 shows an example loop annotated to demonstrate the terminologies used throughout the paper.



```
for (int i=0; i < cands.size(); i++) {
  Candidate cand=cands.get(i);
  if (cand.label.equals(gold.toString())) {
    goldCand=cand;
    break;
  }
}
```

Fig. 2: Example of terminology

B. Features

We present the features of loop-ifs separated into features related to ending statements and features related to the if condition. Table I details the potential values for each feature.

1) Features related to Ending Statement: Sridhara et al. [4] observed that methods often perform a set of actions to accomplish a final action, which is the main purpose of the method. Similarly, in our analysis, the ending statement also plays an important role toward indicating the action of a loop. We have identified five loop features that are related to the ending statement.

F1: Type of ending statement. The syntactic type of ending statement can be a strong indicator of what the overall loop does. We distinguish several types of ending statements for this purpose: assignment, increment, decrement, method invocation, or object method invocation. Further, we separately distinguish assignments that are boolean assignments. The type of ending statement is important in the perspective of determining actions. For example, an increment ending statement is a strong indicator of counting.

F2: Method name of ending statement method call. When the ending statement is a method invocation, the verbs comprising the method name often reflect the loop's actions. An eye-tracking study conducted by Rodeghero et al. [9] indicated that method invocations are heavily read when a developer wants to understand and summarize code. Two loops are not likely to do the same high level action if one has add method and the other has remove method at the end.

We process all loop-ifs to extract the method names in ending method calls. We use Java method naming conventions to extract verbs by splitting with camel case and extracting the first word. Although many verbs are found in method names of ending statements from our data set, Table I only shows verbs eventually used in our action identification model.

F3: Elements in collection get updated. Consider the following example:

```
for (VirtualDiskDescType vDiskDesc : disks) {
  if (diskFileId.equalsIgnoreCase(vDiskDesc.getFileRef()))
      {
    vDiskDesc.setCapacity(String.valueOf(bundleFileSize));
  }}
```

The `set` method is invoked on qualified elements in the collection `disks`, which is the loop control variable of the loop. Since the result variable is the loop control variable, the method is invoked on every element that satisfies the criteria. But if the result variable is not the loop control variable, that is not the case. So this feature has the potential to differentiate between different actions. We set F3=1, when the result variable is the loop control variable; otherwise, F3=0.

F4: Usage of loop control variable in ending statement. Normally, we expect the loop control variable to appear in the ending statement, as the loop goes through a collection and uses elements in some way. We have already considered whether the loop control variable is the result variable or not (F3). Here we consider whether it is directly used or some variable derived from it is used in the ending statement. We also consider if it never appears in the ending statement. Therefore, we set F4=0, when the loop control variable never appears in the ending statement; F4=1, when the loop control variable is directly used in the ending statement; F4=2, when

TABLE I: Semantics of Feature Values

Label	Feature	Possible Values and Their Semantics
F1	Type of ending statement	0: none 1: assignment 2:increment 3:decrement 4:method invocation 5:object method invocation 6: boolean assignment
F2	Method name of ending statement method call	0:none 1:add 2:addX 3:put 4:setX 5:remove
F3	Elements in collection get updated	0: false 1: true
F4	Usage of loop control variable in ending statement	0: not used 1:directly used 2:used indirectly through data flow
F5	Type of loop exit statement	0:none 1:break 2:return 3:return boolean 4:return object 5:throw
F6	Multiple collections in if condition	0: false 1: true
F7	Result variable used in if condition	0: false 1: true
F8	Type of if condition	1: >/</>=/<= 2: others

the loop control variable is on a def-use chain to a use in the ending statement.

F5: Type of loop exit statement. It is important to know whether there is a control flow disruption (i.e., all elements in the collection are considered or not). For example, in a find action, there has to be a disruption of the loop. In addition, the type of disruption might be related to different actions and we want to know the type. Intuitively, a return of boolean indicates the loop checks if there is any element in the collection that satisfied the condition, while a return of an object indicates finding the first qualified element in the collection. So values for F5 are none, break, return, return boolean, return object, and throw.

2) Features related to the if Condition: We consider three features that are related to the if condition. We examine whether multiple collections are compared in the if condition, the role of result variable in the If condition, and the type of if condition expression.

F6: Multiple collections in if condition. This feature is a boolean that indicates whether multiple collections are compared in the if condition. We believe loops that manipulate multiple collections are likely to accomplish very different actions than those that manipulate only one. We set F6=1 if there are two synchronized collections in the if condition; otherwise, F6=0.

F7: Result variable used in if condition. Within the if statement, the result variable will be updated. So determining whether the result variable is part of the if condition is equivalent to determining whether the current value of the result variable needs updating. For example, in finding the maximum or minimum, the current value determines whether it needs to be updated. We set F7=1, if the result variable appears in the if condition; Otherwise, we set F7=0.

F8: Type of if condition. This feature indicates the type of if condition. The if condition can be a numeric value comparison (e.g., "<" and ">"), or a user-defined method that returns a boolean value. For example, a numeric value comparison combined with the feature that the result variable is used in the if condition is a strong indicator of finding the maximum/minimum element in a collection.

3) Other Considered Features: We started with loop control variable, result variable, if condition, and ending statement, and considered their various usages as different features. Many features turned out to be not helpful in determining high level actions. This determination is based on the empirical method we use for developing the action identification model. We just described those that are useful; however, many others (such as the type of loop, whether the if statement contains any else blocks, whether the loop control variable appears in the if condition, and the number of statements in the if block) are omitted from the above description.

C. From Loop to Feature Vector: An Example

A feature vector for a given loop-if is constructed by extracting the features F1 through F8 from the loop's source code representation using simple static analysis. The feature vector for the example code fragment in Figure 2 is:

(F1:1, F2:0, F3:0, F4:2, F5:1, F6:0, F7:0, F8:2)

F1 indicates that the ending statement is an assignment. F2 indicates there is no method name from an ending method call. F3 indicates that not every element in the collection is updated. F4 indicates that the loop control variable is on the def-use chain to a use in the ending statement. F5 indicates that the type of loop exit statement is a break. F6 indicates that there is only one collection in if condition. F7 indicates that the result variable is not used in the if condition. F8 indicates that the type of the if condition is not numeric comparison.

III. Developing an Action Identification Model

Our goal is to identify the action of a loop from its feature vector representation. More than one feature vector can correspond to the same action. Some of the features may be relevant to a specific action, while others may not be relevant to the same action. Therefore, our goal is to determine the combinations of features that are relevant to a specific high level action and to find the groups of vectors that perform the same actions.

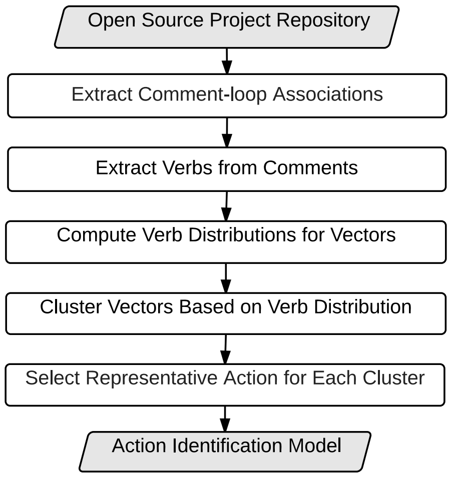
To characterize the high level action performed by a specific feature vector, we could examine several loops corresponding to that loop feature vector that have comments associated with them. If these comments describe what the loop code is accomplishing, we can analyze the action descriptions in these comments, and associate them with the feature vector.

Thus, our first task was to automatically find loops for each feature vector that have descriptive comments associated with them. We call these comment-loop associations. As is known from natural language processing, we found that the verbs in the comments are not sufficient to characterize the actions. Verb phrases with the same verb can describe different actions based on their arguments. Furthermore, verb phrases with different verbs can describe the same action. For example, an action of "finding" is often described by verbs "search", "check", etc. While verbs alone from these descriptive comments are not enough, our further analysis revealed that the distribution of verbs can be a good indicator of the action of a specific loop feature vector.

Our hypothesis is that while different verbs may be used in comments to describe the action of loops with the same loop feature vector, the verbs associated with a loop feature vector are related to each other and can not be any arbitrary subset (of verbs). This hypothesis yields a new opportunity. The distribution of verbs associated with individual loop feature vectors can be the basis of clustering loop feature vectors that perform similar actions. Different sets of verbs associated with a pair of loop feature vectors clearly indicate that the actions performed by the pair are very different. For example, some vectors have associated comment verbs "check", "find", "search", "look", "try", "return", "get", "see", while some have verbs "remove", "filter", "clean", "check", "clear", "prune", "delete". Clearly, they perform different actions. However, two loop feature vectors may just correspond to two different ways of programming the same action, and in this case, we should expect a similar distribution of verbs associated with them.

Thus, we cluster loop feature vectors by the distribution of verbs. Indeed, for our data set, we found that the top 100 most frequently occurring loop feature vectors cluster into just 12 groups on this basis.

However, it is desirable to use a phrase to capture the action rather than using the associated set of verbs. While every step thus far is completely automated, choosing the representative verb phrase to describe the 12 groups was done manually using the procedure described in Section III-E. To summarize, our approach to developing the action identification model follows the process illustrated in Figure 3.



Open Source Project Repository
Extract Comment-loop Associations
Extract Verbs from Comments
Compute Verb Distributions for Vectors
Cluster Vectors Based on Verb Distribution
Select Representative Action for Each Cluster
Action Identification Model

Fig. 3: *Process of developing the Action Identification Model*

A. Collecting Comment-loop Associations

We extract our comment-loop associations from the corpus of 14,317 open source Java projects originating from GitHub [7] . We use half of the repository (7,158 open source projects) to develop our action identification model, so we can use the remaining half for evaluation of the model.

To extract all available comment-loop associations from the corpus, we traverse the abstract syntax tree of each project and collect all loop-ifs with any associated comments. We identify the internal comments associated with loop-ifs by following the general convention that blank lines indicate separation of blocks of code that logically fit together and the comment immediately preceding the block is describing that code block [10], [11]. The end of a block could also be the end of a method or the end of a statement block where the comment resides. Thus, we extract any loop-if that has a preceding internal comment and ends with a blank line or end of method.

B. Extracting Verbs from Associated Comments

Previous researchers have developed classifications of comments based on the information conveyed by the comments [11]–[14]. Comments can be descriptive, explanatory, evolutionary, and conditional. We are only interested in descriptive comments as they describe what the code following the comment is doing, at a higher level of abstraction than the code. We begin by filtering out all other kinds of comments using heuristics similar to others. From our observations, descriptive comments tend to be verb phrases, while non-descriptive comments are more likely to be sentences. However, a descriptive comment does not necessarily start with a verb; it may start with “for”, “now”, “if”, “the”, “first”, etc. In each case, we still need heuristics for identifying verbs in the natural language text of the comment. For example, the comments starting with the word “for” are often in the format: “for each/all . . . , verb . . . ”. We extract the verb by selecting the first word in the second clause. Similarly, we extract verbs for other cases. After applying the heuristics, we also check if the ‘extracted ‘verb” can be found in the verb dictionary [15]. We base the verb extraction process on the work of Howard et al. [16].

From our repository that contains 7,158 open source Java projects, we extracted 30,089 distinct qualified comment-loop associations from which we could extract verbs. Each comment-loop association is related to a single feature vector based on the feature vector representing the loop in the comment-loop association.

C. Computing Verb Distributions for Feature Vectors

At this point, each feature vector has an associated set of comment-loop associations composed of the commented loop-ifs in the development set that are represented by that feature vector. Each comment-loop association has a comment verb, so each feature vector has an associated set of comment verbs.

While different verbs may be used to describe the action of a loop feature vector, the verbs for a loop feature vector are related to each other and can not be any arbitrary subset (of verbs). Certain verbs frequently appear together for some feature vectors. For example, the set of {"check", "find", "get", "search", "look for"} and the set of {"set", "update", "make", "add", "change", "reset", "replace"} contain verbs that have strong correlation. Those groups of verbs have similar meaning and are together naturally. On the other hand, some verbs are less likely to appear together, because they have very different or conflicting meanings. For example, "add" and "remove" rarely appear together.

Our hypothesis is that the distributions of comment verbs associated with feature vectors correspond to different high level actions, and thus can be used to cluster loop feature vectors that perform similar actions. We cluster the feature vectors with similar actions because we want a generalized model that can identify the action for any given feature vector. The top 100 most frequent feature vectors cover more than 59% of loops in the data set. Thus, to give us good coverage, we generalize by performing the clustering on the top 100 most frequent feature vectors. In addition, the number of loops associated with a given feature vector starts dropping after the top 100 feature vectors, such that clustering on the basis of verb distributions would become less reliable. Most of the feature vectors have 5-6 verbs that occur repeatedly in the comment-loop associations for that feature vector. In addition, the distribution of verbs associated with each feature vector has a long tail. Thus, the verb distribution for a given feature vector is based on the distribution of the top 10 verbs most frequently associated with that feature vector.

Specifically, we extract all unique verbs that are associated with the top 100 feature vectors. There are 230 unique verbs in total, which becomes the dimension of the verb probability distribution. For each feature vector, the verbs are transferred to a normalized probability distribution of verbs associated with that feature vector.

D. Clustering Feature Vectors Based on Verb Distribution

To help us group together the same actions, we use hierarchical clustering analysis [17]. Hierarchical clustering does not require us to specify the number of clusters. We use the complete linkage method for hierarchical clustering. This particular clustering method defines the cluster distance between two clusters to be the maximum distance between their individual components. At every stage of the clustering process, the two nearest clusters are merged into a new cluster. The process is usually repeated until the whole data set (100 loop vectors) is agglomerated into one cluster. However, since the cluster quality drops as we create larger clusters, we chose to stop the process when the distance between clusters to be merged reached a threshold value of 0.5. This resulted in 10 clusters.

Given two feature vectors, let p and q be their corresponding verb probability distributions, respectively. Each distribution will be the assignment of each verb $verb_i$, from 1 to the total number $m = 230$ unique verbs. The Euclidean distance between verb distributions p and q is:

$$d(p,q) = \sqrt{\sum_{i=1}^{m}(p(verb_i) - q(verb_i))^2}$$

Kullback-Leibler divergence and Jensen-Shannon divergence are two additional popular methods of measuring the similarity between two probability distributions. For our data set, we found that Euclidean distance performs very well for our clustering purpose.

E. Creating Action Identification Model

The last step is to create the action identification model based on the clusters. This step is manual because we want to discover and verify what action each cluster performs. This can not be automated.

Every cluster corresponds to a group of feature vectors that have some common feature values and some values different for a few features. In our development set, we manually analyzed up to 33 vectors per cluster. For any single cluster, we began by examining the common feature values and expressed them as a loop-if template. If a difference in feature value indicated a possible difference in action, we chose to divide the cluster on the basis of that feature.

TABLE II: Action Information for 100 Most Frequent Loop Feature Vectors in Development Set

Action	# of Feature Vectors	# of Loops
find	33	5312
get	16	2967
determine	8	2192
add	11	940
remove	9	818
copy	2	714
count	3	512
max/min	4	500
ensure	2	395
set_all	4	330
set_one	2	223
compare	1	157

For example, the major differences between members of a particular cluster were the value for feature F5 (type of loop exit statement), F7 (result variable used in if condition) and F8 (type of if condition). Since the presence or absence of loop exit is critical in the action computed by a loop (because it determines whether or not all elements in the collection will be processed), we chose to divide the cluster on the basis of whether or not the loop-if has a loop exit statement. Immediately, this resulted in two clusters that suggest the following two loop-if templates. These templates correspond to the "min/max" and "find" actions in Table IV, respectively.

```
for (loop_control_variable for a collection) {
  if (numeric comparison) {
   //result variable used in if condition
    result_variable = loop_control_variable;
   // no loop exit statement
}  }

for (loop_control_variable for a collection) {
  if (numeric / non-numeric comparison ) {
    result_variable = loop_control_variable;
    break/return/return result_variable;
}  }
```

TABLE III: Identified actions with their verb phrase descriptions

Label	**Action Phrase**
count	count the number of elements in a collection that satisfy some condition
determine	determine if an element of a collection satisfies some condition
max/min	find the maximum/minimum element in a collection
find	find an element that satisfies some condition (other than max/min)
copy	copy elements that satisfy some condition from one collection to another
ensure	ensure that all elements in the collection satisfy some condition
compare	compare all pairs of corresponding elements from two collections
remove	remove elements when some condition is satisfied
get	get all elements that satisfy some condition
add	add a property to an object
set_one	set properties of an object using objects in a collection that satisfy some condition
set_all	set a property for all objects in a collection that satisfy some condition

The manual analysis of the 10 feature vector clusters led to splitting some clusters such that there are 12 distinct actions identified from the original 10 clusters associated with top 100 most frequent feature vectors. To give an insight into the amount of manual analysis, Table II shows the number of top 100 most frequent feature vectors and the total number of loops in the development set associated with each action. The manual analysis was performed on feature vectors, not on individual loops. For each cluster, we analyzed the template and created a verb phrase description. Table III shows the 12 identified action stereotypes and their verb phrase descriptions. Table IV constitutes a model where each row shows an action and its corresponding combination of feature values. Some actions appear in multiple rows because the different feature value combinations could not be merged into one. For example, if a loop has combination of value 0 for F1 and 2 or 3 for F5, or value 6 for F1 and 0 or 1 for F5, then the model will label this loop with action determine.

IV. Example of Automatic Action Identification

To exemplify the automatic action identification process shown in Figure 1, consider the following source code:

```
for (Subunit s : subunits) {
  if (s instanceof Department) {
    Department subDepartment=(Department)s;
    Department result=subDepartment.findDepartment(id);
    if (result != null) {
      return result;
} } }
```

With the action identification model and the given loop-if source code, the automatic identifier first extracts the features as described in Section II. The extracted feature vector is (F1:1, F2:0, F3:0, F4:2, F5:2, F6:0, F7:0, F8:2). By mapping the feature vector against the action identification model in Table IV, the identified action is "find". Thus, we can identify the action as "find an element that satisfies some condition".

V. Evaluation

We implemented the automatic action identifier, and designed our evaluation to answer the two main questions:

- *RQ1 - Effectiveness.* How effective is the automatic action identifier?
- *RQ2 - Prevalence.* How prevalent are the actions we are able to identify in Java software?

TABLE IV: Action Identification Model

Feature / Action	F1	F2	F3	F4	F5	F6	F7	F8
count	2				0			
determine	0				2,3			
determine	6				0,1			
max/min	1			1,2	0	0	1	1
find	1			1,2	1,2,4			
find	0				4			
copy	5	1	0	1	0			
ensure					5			
compare					3	1		
remove	5	5	1	1	0			
get	5	1,3	0	2	0			
add	5	2	0	1,2	0			
set_one	5	4	0	1,2				
set_all	5	4	1	1,2	0			

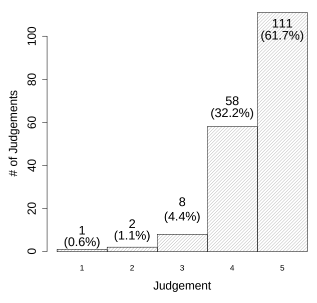
A. *Effectiveness of Automatic Action Identification*

Procedure. We asked 15 human evaluators to judge the output of our prototype in identifying and describing high level actions targeted by our technique. To reduce potential bias, we told the judges that the action descriptions were identified from different systems, and that our goal was to know which system is better.

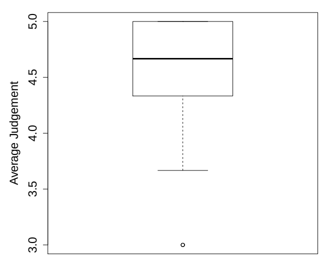
The programming experience of this group ranges from 5 to 15 years, with a median of 9 years; 13 of the evaluators consider themselves to be expert or advanced programmers, and 4 evaluators have software industry experience ranging from 1 to 3 years. None of the authors of this paper participated in the evaluation.

For evaluation, we randomly selected 5 loops for each of the 12 actions that we are able to identify. To account for variation in human opinion, we gathered 3 separate judgments for each loop. Hence, each human evaluator evaluated 12 loops. In total, we obtained 180 independent judgments on 60 loops, by 15 developers working independently with 3 judges per loop.

In a web interface, we showed evaluators a code fragment and asked them to read the code. Then, they were instructed to click a button to answer questions. This process let the evaluator read the code first and avoid their opinion being affected by the provided options. We asked evaluators two



(a) Individual judgements



(b) Average judgements per loop

Fig. 4: Human judgements of identified actions (1:Strongly disagree, 2:Disagree, 3:Neither agree or disagree, 4:Agree, 5:Strongly agree)

questions about the high level action of the code:

1) How much do you agree that the loop code implements this action?
2) How confident are you in your assessment?

The evaluators were asked to respond to the first question via the widely used five-point Likert scale: 1:Strongly disagree, 2:Disagree, 3:Neither agree or disagree, 4:Agree, 5:Strongly agree. Similarly, evaluators were asked to respond to the second question with values 1:Not very confident, 2:Somewhat confident, or 3:Very confident.

Results and Discussion. Figure 4a shows the number of individual developers' responses for each point in the Likert scale for the first question. The results strongly indicate that the code fragments that we automatically identify as high level actions are indeed viewed as high level actions by developers. In 93.9% (169 out of 180) of responses, judges strongly agree or agree that the identified actions represent the high level actions of the loops, within which 61.9% correspond to strong agreement. Furthermore, out of 180 responses, 176 judges are confident or very confident about their opinions.

For each of the 60 loops given to evaluators, we also computed the average opinions given by 3 evaluators for each loop. Figure 4b shows the box plot of average opinions for the 60 loops. The average opinions per loop range from 3 to 5, with their median being 4.33. Also 91.7% (55 of 60) of the average opinions were 4.0 or above, where 4.0 indicates agreement that the identified action represents the high level action of the loop.

We analyzed the three loops with the average opinion below 3: Neither agree or disagree. One interesting case is that when an if block contains many statements and there are multiple actions expressed by the statement block, the last statement itself is not sufficient to summarize the major action. Similarly, when there are multiple then clauses in the if statement and each of them does a different action, the ending statement we select from the first clause is not sufficient to represent all of them. In another case, when the last statement is an object method invocation and the object is a class, the method is then a static method. In such cases, the class that the static method is invoked on is not the object that gets updated. Instead, the static method updates the object passed into the method. We can improve our implementation to capture this case.

B. Prevalence of Identifiable Actions

Procedure. To determine the potential impact of automatically identifying the high level actions of loop-ifs, we ran the action identifier on all 7,159 open source projects that we had saved as our test data set. Cumulatively, these programs contain 9,358,179 methods, with a median of 150 and maximum of 206,175 methods per project. We gathered data on the frequency of each high level action that was automatically identified.

Results. In the test data set, there are 1.3 million loops, of which 337,294 are loop-ifs. For those loop-ifs, we were able to automatically identify 195,277 high level actions (i.e., 57.9%). The frequency distribution of each identified high level action is shown in Figure 5.

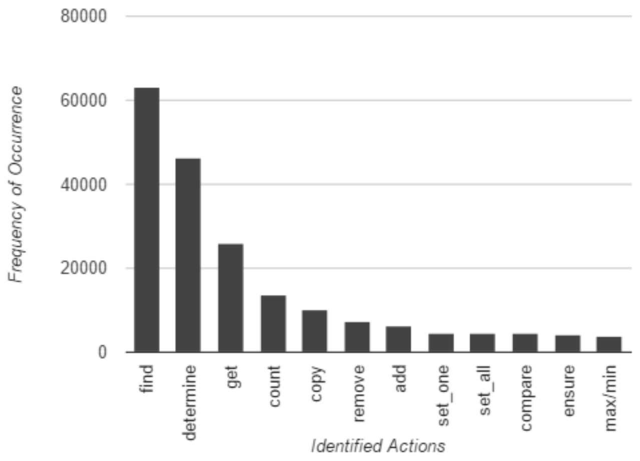


Fig. 5: *Identified high level action distribution over 7,159 projects*

For this large corpus of Java projects, we believe these numbers are high enough to demonstrate that our action identification technique has wide applicability.

C. Summary of Evaluation Results

Human judgments by 15 developers strongly suggest indeed that they view our automatically identified descriptions as accurately expressing the high level actions of loops. Our study of the prevalence of detected high level actions in over 7000 Java open-source projects indicates that our algorithm for automatically identifying loop-ifs that implement high level actions has wide applicability.

D. Threats to Validity

Our results may not generalize to other Java programs. To mitigate this, we used a repository that contains 14,317 projects from GitHub. We use half of the repository to develop our technique and the other half for testing. Although our extracted features are not specific to the Java programming language, they may not generalize to other programming languages. Our evaluation relies on human judges, which could be subjective or biased; to mitigate this threat, each action description was judged by three evaluators, and the evaluators were told that the output they were judging was from two different tools that we were comparing. The reported prevalence numbers could be slightly overestimated because we did not manually check all 195,277 high level actions identified for accuracy; however, our accuracy results show that it should be a good approximation.

VI. Improving Client Tools

Another measure of this work's contribution is the potential impact on client tools for software maintenance. Here, we discuss refactoring, code search, internal comment generation, and automatic code completion.

Our technique can help developers identify loops that can be refactored to a method or migrated to using the new Java 8 Stream APIs. For example, the automatic action identifier can identify loops as implementing the "find" action, which the developer could then migrate to the Java 8 Stream API.

The automatic action identifier has the potential to increase the effectiveness of code search tools by providing the action phrase with the associated loop. To illustrate, in our data set of 63,265 loops that we identified automatically as "find", the word "find" appears in the loop bodies only 1442 times (2% of the "find" loops).

Studies have shown the utility of comments for understanding software [18], [19]. However, few software projects contain many internal comments [2]. In our data set, less than 20% of loop-ifs are commented. Considering that some of the comments are poor or nondescriptive, the actual percentage is even less. Developers can use the automatic action identifier to generate internal comments by customizing the verb phrases using identifiers from the specific loops.

Major IDEs [20], [21] currently allow programmers to define code snippets and easily reuse them. Manually defining the templates is tedious and time consuming. When a developer wants to perform a specific action that we have identified as a high level action in our action identification model, we can provide the code template.

VII. Related Work

To our knowledge, this is the first general, extensible approach to automatically identifying action units and abstracting them as high level action phrases, additionally demonstrating the feasibility of defining a model of action unit stereotypes without manually creating templates. The closest work is by Sridhara et al. who automatically generates high-level actions within methods [6], by using a small set of templates that were developed by manually examining code. By using our automated approach, we were able to handle loops from 7 more stereotype actions just by considering the loop-if's. We believe the same process can be extended even more.

Others extract topic words or verb phrases from source code [22]–[24] to identify code fragments that are related to a given action or topic, and sometimes cluster program elements that share similar phrases [25]. These approaches rely solely on the linguistic information to determine the topic of the code segment, which we found is not adequate for many action units where the action is not expressed as a word within the source code explicitly.

Since we are able to generate internal comments for the identified high level actions, our work is related to comment generation. Most comment generation work is focused on creating summaries for methods or classes [4], [26]. However, Wong et al. mine question and answer sites for automatic comment generation [27]. They extract code-description mappings from the question title and text, use heuristics to refine the descriptions, and use code clone detection to find source code snippets that are almost identical to the code-description mapping. Their technique does not leverage an abstraction of source code, which leads to generating only 102 comments for 27 large projects. In contrast, we abstract the features that are related with an action, and only focus on related features.

Many others have been mining code fragments across projects [28]–[33]. Most of the work focuses on mining statement sequence examples of using APIs to help with learning APIs or code reuse, not identifying high level actions. For those problems, there is no need for an action identification model.

Our development of the action identification model through clustering feature vectors characterizing loop stereotypes is related to finding code clones, where clones are similar code fragments typically caused by copy-paste operations [34], [35]. Unlike code clones, code segments that implement the same high level action in many places across projects have a broader set of characteristics that are related among the code segments than code clones, and are rarely verbatim copies like with code clones. Consequently, we found that existing code clone detection techniques based on text [36], token [37], abstract syntax tree [38], and program dependence graph [39] do not capture the kinds of characteristics in code segments implementing the same action. Action units also differ from code idioms which tend to be either library specific or general idioms such as looping over an array or defining a string constant, which are not action specific.

VIII. Conclusion and Future Work

We presented a novel technique that uses loop structure, data flow and word usage to automatically identify action units. Based on 15 experienced developers' opinions in which 93.9% of responses indicate that they strongly agree or agree that the identified actions represent the high level actions of the loops, we are quite encouraged that we have demonstrated the feasibility of characterizing loops in terms of their features learned from a large corpus of open source code enough to accurately identify high level actions. The results also show that the technique needs only a few comment-loop associations to exist in a large corpus to support the approach.

Based on the success of this approach, we are examining how to apply the overall approach to other kinds of potential action units beyond loop-ifs. We will also explore the applicability of the approach to other programming languages and incorporate the resulting action phrases into client tools.

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd Annual International Conference on Design of Communication*. New York, NY, USA: ACM, 2005, pp. 68–75.

[2] M. Kajko-Mattsson, "The state of documentation practice within corrective maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2001, p. 354.

[3] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[4] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 43–52.

[5] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 17th Working Conference Reverse Engineering (WCRE)*, 2010, pp. 35–44.

[6] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[7] M. Allamanis and S. Charles, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.

[8] A. M. Stefik, C. Hundhausen, and D. Smith, "On the design of an educational infrastructure for the blind and visually impaired in computer science," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE)*. New York, NY, USA: ACM, 2011, pp. 571–576.

[9] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2014, pp. 390–401.

[10] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 35–44.

[11] S. McConnell, *Code Complete, Second Edition*. Microsoft Press, 2004.

[12] P.-N. Robillard, "Automating comments," *SIGPLAN Not.*, vol. 24, no. 5, pp. 66–70, 1989.

[13] L. Etzkorn, C. Davis., and L. Bowen, "The language of comments in computer software: A sublanguage of English," *Journal of Pragmatics*, vol. 33, pp. 1731–1756(26), November 2001.

[14] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers - Taxonomies and characteristics of comments in operating system code," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, May 2009.

[15] Wordnet. https://wordnet.princeton.edu/wordnet/.

[16] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 377–386.

[17] L. Rokach and O. Maimon, "Clustering methods," in *Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, Eds. Springer US, 2005, pp. 321–352.

[18] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[19] T. Tenny, "Program Readability: Procedures Versus Comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, 1988.

[20] SnipMatch. http://marketplace.eclipse.org/content/snipmatch.

[21] JetBrains. https://www.jetbrains.com/idea/help/live-templates.html.

[22] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proceedings of the 1st India Software Engineering Conference (ISEC)*. New York, NY, USA: ACM, 2008, pp. 113–120.

[23] M. Ohba and K. Gondow, "Toward mining "concept keywords" from identifiers in large software projects," in *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2005, pp. 1–5.

[24] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 232–242.

[25] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, 2007.

[26] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of the 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 23–32.

[27] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 562–567.

[28] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 782–792.

[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2009, pp. 383–392.

[30] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007, pp. 35–44.

[31] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *Proceedings of the 23rd European Conference on ECOOP*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[32] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2014, pp. 653–663.

[33] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*. New York, NY, USA: ACM, 2014, pp. 472–483.

[34] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 1996, pp. 244–253.

[35] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 61–82, Mar. 2006.

[36] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 1999, pp. 109–118.

[37] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[38] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 1998, pp. 368–377.

[39] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of 8th Working Conference on Reverse Engineering (WCRE)*, 2001, pp. 301–309.