Automatically Detecting and Describing High Level Actions within Methods[*]

Giriprasad Sridhara, Lori Pollock and K. Vijay-Shanker
Department of Computer and Information Sciences, University of Delaware
Newark, DE 19716 USA
{gsridhar, pollock, vijay}@cis.udel.edu

ABSTRACT

One approach to easing program comprehension is to reduce the amount of code that a developer has to read. Describing the high level abstract algorithmic actions associated with code fragments using succinct natural language phrases potentially enables a newcomer to focus on fewer and more abstract concepts when trying to understand a given method. Unfortunately, such descriptions are typically missing because it is tedious to create them manually.

We present an automatic technique for identifying code fragments that implement high level abstractions of actions and expressing them as a natural language description. Our studies of 1000 Java programs indicate that our heuristics for identifying code fragments implementing high level actions are widely applicable. Judgements of our generated descriptions by 15 experienced Java programmers strongly suggest that indeed they view the fragments that we identify as representing high level actions and our synthesized descriptions accurately express the abstraction.

Categories and Subject Descriptors:
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Documentation*
General Terms: Algorithms, Documentation
Keywords: Program Comprehension, Documentation

1. INTRODUCTION

It is almost axiomatic that software maintenance is resource intensive and program comprehension required for maintenance is difficult and time-consuming, especially for a newcomer, i.e., a developer faced with an unfamiliar system or an unfamiliar component of a known system. One approach to helping program comprehension is to reduce the amount of code that a human has to read. For a given method, a human could potentially focus on fewer and more abstract actions at a time to understand the code if the code segments that implement higher level algorithmic steps were delineated and described at a more abstract level. However, such internal documentation is often missing or obsolete because it is tedious to manually create and keep current. Additionally, extract method refactoring could also condense a method's code size by replacing the extracted method body statements by a method call; however, this transformation requires tedious analysis to maintain the original semantics.

In this paper, we present the first known technique to automatically identify groupings of statements (i.e., code fragments) that collectively implement high level actions and to then synthesize a succinct natural language description to express each high level abstraction. By a *high level action*, we mean a high level abstract algorithmic step of a method. For example, our algorithm will automatically deduce that the code fragment from lines 9 to 16 in Listing 1 implements a more abstract action and then synthesize the natural language description to represent that action:

```
9  for (int x = 0;  x < vAttacks.size(); x++) {
10   WeaponAttackAction waa=vAttacks.elementAt(x);
11   float fDanger = getExpectedDamage(g, waa);
12   if (fDanger > fHighest) {
13     fHighest = fDanger;
14     waaHighest = waa;
15   }
16 }
17 return waaHighest;
```

Listing 1: Lines 9-16 implement high level action

Synthesized Description: *Get weapon attack action object (in vectorAttacks) with highest expected damage*

The resulting phrase reduces and factors out the details of the set of individual statements that implement the high level algorithmic step being performed. [1]

Our algorithm takes a Java method as input and outputs the method with natural language phrases associated with blocks of statements identified as implementing high level actions. We separately analyze sequences of statements, conditional statement blocks, and loops within the method. To delineate a grouping of statements that collectively implements an identifiable high level action, we leverage both programming language syntax and semantic information as well as linguistic clues embedded in the developers' naming of entities. To synthesize succinct natural language descriptions to express the abstraction, we utilize advanced text

*This material is based upon work supported by the National Science Foundation Grant No. CCF-0702401 and CCF-0915803.

[1]For each of the listings that we use to illustrate challenges and aspects of our technique throughout this paper, we provide the automatically synthesized description.

generation techniques. Our automatic system involves analysis of source code only, requiring no execution information, and thus can be applied to incorrect, incomplete, and unexecutable legacy systems. Since the analysis is local to a method, this capability can be integrated easily into an IDE to provide current descriptions as the software developer begins editing a given method.

Our synthesized descriptions have many uses for software maintenance beyond a tool to help with program understanding. We are able to identify potential candidates for the *ExtractMethod* refactoring [7] and additionally to suggest a good name for the extracted method using the action, theme (i.e., direct object) and other arguments (i.e., indirect objects, prepositional phrases,...) that are identified to synthesize a description. This same information can be used to identify poor method names and suggest better naming that includes both action and theme. The phrases could be used to verify traceability results as well as to generate internal comments for legacy codes. The high level actions that are extracted and expressed as phrases can be used to generate smoother, more succinct method summaries than previous work [21]. We also believe that code search can be improved by using the high level action descriptions to associate higher level actions with the methods containing them, providing new verbs for the queries to match in the code. For example, it might be easier to locate code that performs a Max after identifying and describing a given code fragment within a method that implements this higher level action.

The main contributions of this paper are:

- A set of algorithms to identify code fragments (of sequences, conditionals, and loops) within a method that implement high level abstract actions
- Rules to synthesize succinct, accurate natural language descriptions expressing the high level actions
- Validation of the prevalence and potential reduction in detail with our system applied to 1.2 million methods from 1000 Java programs, and developer-judged precision of the automatically identified high level actions and associated descriptions. 225 independent judgements among 15 experienced Java developers on 75 identified code fragments strongly confirm that the code fragments we identify as high level actions are indeed abstractions and that our synthesized descriptions accurately express these abstractions.
- Illustrations of how the synthesized descriptions can improve four client tools for software maintenance

2. PROBLEM STATEMENT

In this paper, we are addressing the problem:

> Given the signature and body for a method *M*, automatically discover each code fragment that implements a high level abstract action comprising the overall algorithm of *M*, and accurately express each high level action as a succinct natural language description.

Consider the example `makeLine` in Listing 2. This method is composed of three high level actions: (1) create a horizontal box, (2) add the given components to the box, and (3) return the box. One can view any method as representing a high level action itself, especially from the perspective of its callers. Since there is nothing to be gained by labeling individual method calls or statements as a high level action, we instead focus on the problem of grouping multiple statements that collectively implement a high level abstract action. In the example, we seek to automatically capture that the three statements 4-6 together implement the second algorithmic step of the method.

```
1 Box makeLine(JComponent f, JComponent s,
     JComponent t){
2   Box onelineBox = Box.createHorizontalBox();
3   // add components
4   onelineBox.add(f);
5   onelineBox.add(s);
6   onelineBox.add(t);
7
8   return(onelineBox);
9 }
```

Listing 2: High level action as a sequence : Lines 4 to 6: *"add given components to oneline box"*

There may be a myriad of types of statement groups that might represent high level actions. This paper examines three kinds of statement groups corresponding to major control structures within a method:

- a sequence of statements that when taken together represents a single high level action
- a conditional block that performs an action with subtle variations based on the condition
- code patterns that are commonly implemented using loop constructs that constitute a high level action

In this paper, we focus on identifying arbitrary statement sequences in which each statement is performing a similar action. We do not solve the problem of identifying groupings of statements where each statement performs a different kind of sub-action of a given high level action, such as a swap operation. These sequences require looking for a specific known syntactic template or domain knowledge. Instead, we focus on identifying high level actions composed of a set of similar actions with some differences in the actions' arguments, such as in Listing 2, without a predefined set of syntactic templates. We focus on similar kinds of high level actions in conditionals, except the similar actions now occur in different branches, creating a different set of challenges for identification and synthesis. For loops, we examine specific high level actions based on templates; the major challenge for handling loops is synthesizing a good description for the high level action from the loop information. Identification and text representation synthesis have challenges specific to each of these types of statement groups. We describe the specific challenges within each subsection below.

Note that techniques for identifying fragments representing high level actions (this work) and those statements important for summaries [21] are mutually exclusive. Additionally, rules for generating text for a code fragment are different from those for a single statement [21]. Text generation for a sequence builds on single statements with extension for a combined phrase; however, loops and conditionals go beyond combining phrases for component statements, instead having to address control structures used in different ways (ex: getMaximum in Listing 1).

3. DETECTING AND DESCRIBING HIGH LEVEL ACTIONS

Our approach to automatically detecting and documenting high level actions follows the process illustrated in Figure 1. The first phase, a preprocessing phase, involves traditional program analysis to build the program representations that capture program syntax and semantics and linguistic information used by our analysis. We then traverse the method's abstract syntax tree and use both program structure and linguistic information to identify code fragment candidates for high level action abstraction. For each candidate high level action fragment, we analyze the code fragment to identify the action, theme, and secondary arguments to describe the high level action, and then generate the natural language phrase based on the kind of code fragment — sequence, conditional or loop. Each remaining subsection describes the individual phases of our approach.

3.1 Structure and Linguistic Information

Both the analysis to detect code fragments implementing high level actions and analysis to synthesize descriptions use information in the abstract syntax tree and control flow graph representations of the method under analysis. Both analyses also use information from naming conventions and linguistic knowledge gained from observations of thousands of Java programs. In particular, we use information from the control flow graph, and data and control dependences, along with textual clues which we obtain from the Software Word Usage Model (SWUM) [10] of the program.

Before any word usage information can be extracted from names used in the program, identifiers must be split into component words. We use camel case splitting, which splits words based on capital letters, underscores, and numbers (e.g., childXMLElement would be split into "child XML Element"), and aspects of more advanced splitting [6]. As in any system that uses linguistic information, our technique will be hindered if the source code does not include at least some meaningful variable, method, and type names. We believe this requirement is reasonable, given that developers tend to choose long and descriptive names for highly visible program entities such as methods and types [18].

Abbreviations in variable and type names can reduce the readability of the generated description and accuracy of our analysis (e.g., Button butSelectAll, MouseEvent evt). We use techniques from prior work [11] to automatically identify and expand abbreviations in code.

The Software Word Usage Model (SWUM) [10] provides us with the necessary linguistic information beyond individual word frequencies necessary to both identify and express high level actions. SWUM not only captures the occurrences of words in code, but also their linguistic and structural relationships. SWUM has been successfully used in concern location and summary comment generation for Java methods [21].

Particularly, we use SWUM to obtain the action, theme, and optional secondary arguments of a statement grouping to generate succinct and smooth descriptions, and, in conjunction with program structure, we use this information to identify code fragments that implement a high level action. Consider the example method signature list.add(Item i), which can be captured by the phrase, "add item to list." In this example, the action is "add", the theme is "item" and the secondary argument is "(to) list". Further, in this example,

```
7   buildGameMenu();
8   buildViewMenu();
9   buildOrdersMenu();
10  buildReportMenu();
11  buildColopediaMenu();
```

Listing 3: Different method calls can form a fragment in a sequence : Lines 7 - 11: *"build menus"*

```
15        contentPane.add(endedPanel);
16        contentPane.add(bidPanel);
17        contentPane.add(endingPanel);
18        contentPane.add(buttonOK);
```

Listing 4: Same method calls need not form a fragment in a sequence : Fragment is lines 15 - 17 only : *"add panels to content panel"*

the location of the theme is the given parameter while the location of the secondary argument is the receiver object. In addition to these locations, a theme or secondary argument can be the method name itself (e.g., buildMenu()) Lastly, we leverage our previous work in [21] which develops a strategy for variable lexicalization, in which descriptive noun phrases describing variables are generated with modifiers extracted through type information.

3.2 Sequence as Single Action

Challenges. We focus on identifying sequences of statements with similar actions, indicated by similar method calls. The basic identification challenge is to determine whether a given statement is *similar* to its successor statement and thus can be *integrated* with it to build a fragment. To *integrate* two statements, we need to define a notion of similarity between the statements' actions.

From Listing 2, one might think that statements with the same method being called can be integrated to form a fragment. However, as shown in Listing 3, the individual method names are *different*, yet there is a high level action implemented by lines 7-11, synthesized as *build menus* or *build different menus*.

Conversely, a sequence of the same method being called may not be integrable. In Listing 4, we claim that lines 15 to 17 have a high level action of "add panels to content panel". It is not clear if line 18 should be included in this fragment, as the type of the actual parameter to add in line 18 is JButton, while the type of the actual parameter to the add calls on lines 15 to 17 is a JPanel. A description of lines 15 to 18 is likely to mention two different kinds of addition and hence our belief that lines 15 to 18 represent two individual steps. While one could use the formal parameter of add to include 18 with 15 to 17, the resulting abstraction of "add components to content panel" is too vague.

As shown above, common actions alone do not suffice in *integrating* two statements. Thus, we take into account the fact that there could be differences among the method calls in the name, parameter or receiver object. Even among the differences, we strive to find some commonality to integrate statements into a fragment.

The next challenge is to synthesize a phrase that represents an abstraction of the fragment. To generate such a phrase, we need to identify the common and different parts

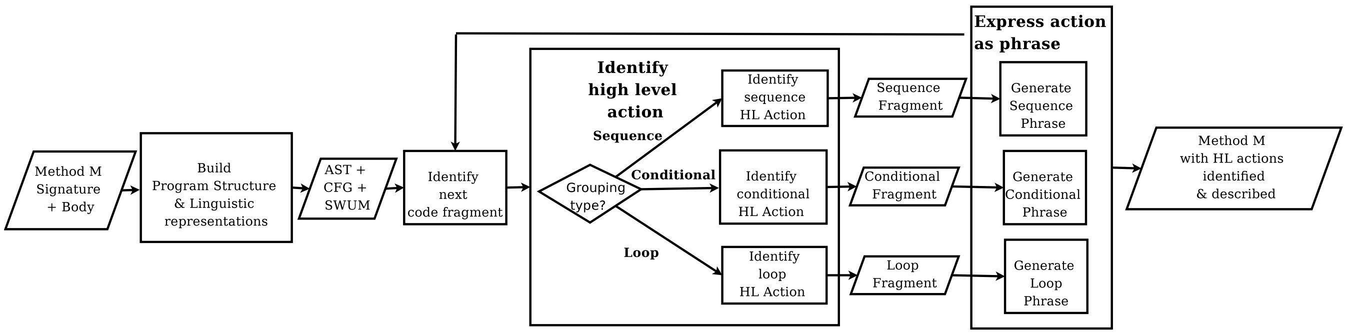
Method M Signature + Body
Build Program Structure & Linguistic representations
AST + CFG + SWUM
Identify next code fragment
Identify high level action
Grouping type?
Sequence
Conditional
Loop
Identify sequence HL Action
Identify conditional HL Action
Identify loop HL Action
Sequence Fragment
Conditional Fragment
Loop Fragment
Express action as phrase
Generate Sequence Phrase
Generate Conditional Phrase
Generate Loop Phrase
Method M with HL actions identified & described

Figure 1: High level view of process: Detect high level action code fragments and synthesize descriptions

among the integrable statements and find a way to express the differences in a concise manner without being overly generic.

Identifying fragments. We focus on integrating statements with one or more method calls, as this is the general case. For a given method, we build a fragment in an incremental manner. We determine if pairs of statements are integrable, starting with the first statement in the method. We build a fragment of such integrable statements with maximum possible length. A given method can have several such fragments.

Figure 2 shows the heuristic for "integrable" used to integrate any two statements in a sequence. We illustrate the execution of the algorithm in Figure 2 with the example in Listing 4. The first task is to generate succinct verb phrases for the given two statements using templates defined in [21]. A *verb phrase* begins with a verb followed by a noun phrase (NP) and optional prepositional phrases (PP). A prepositional phrase begins with a preposition followed by an NP. In an English NP, the last word is typically the *head* word and is important because it bears the type of the object it refers to, whereas preceding words modify or describe the head word.

Consider Line 15 and Line 16 in Listing 4. Our system generates the verb phrases *add ended panel to content panel* and *add bid panel to content panel* for lines 15 and 16, respectively. The verbs in the verb phrases are the same. The PPs are the same – *to content panel.* The NPs have the same head word, *panel.* Thus, we decide that lines 15 and 16 can be integrated. Similarly, lines 16 and 17 can be integrated. However, for line 18, the generated phrase is *add okButton to content panel.* Now the head words of the NP in the phrases for lines 17 and 18 are not the same. Thus, we do not integrate lines 17 and 18, and the fragment only consists of lines 16, 17 and 18.

Synthesizing the description. In an identified fragment, we synthesize the description by beginning with the common verb. To generate a single phrase for NP1 and NP2 (Figure 2), we use the *plural* form of the common head word of NP1 and NP2, if the NPs differ, or use the common Noun Phrase. For the PP, we use the common preposition and use our previous work on lexicalization of variables [21] to synthesize the NP part of the PP. Finally, if the NPs do not share the head word but correspond to fields of the same class, we synthesize *"all attributes"* or *"different attributes"* based on whether all or some fields of the class appear in the different statements of the fragment.

```
boolean areIntegrable(Statement S1,Statement S2)
{
 VP1 = generateVerbPhraseFor(S1);
 VP2 = generateVerbPhraseFor(S2);
 // A verb phrase is of the form,
 // verb noun-phrase [prepositional-phrase]
 // V NP [PP], with PP optional

 if !sameVerb(VP1, VP2) return false;

 if !sameHeadWord(NP1, NP2) AND
  !fieldsOfSameClass(NP1, NP2)
    return false;

 if present(PP1, PP2) AND !samePreposition(PP1, PP2)
  return false;

 return true;
}
```

Figure 2: Integrating statements in a sequence

3.3 Abstracting Conditionals

Challenges. In the case of a sequence, we only focused on a series of statements with method calls. However, for conditionals, we need to integrate additional kinds of statements. Additionally, there can be multiple statements along different branches, which are not integrable as a sequence but some or all of the statements can be integrated with a corresponding statement in the other branch(es). In addition to unifying the statements along the different branches, we need to handle the integration of the conditional expressions guarding the different branches, as shown in Listing 5.

Synthesizing the description poses new challenges due to the new types of statements that can be unified, the handling of multiple statements as described above and the need to describe succinctly the guarding conditional expressions. For example, there may be opportunities to integrate return statements that occur along different paths of a conditional. When integrating return statements, all the return expressions may be literals or similar method calls, in which case, we need strategies to synthesize a more precise phrase, as shown in Listing 6.

Identifying and describing conditionals. Here, we describe our approach for integrating statements in the then and else branches of an if else statement. We handle if . . . else if . . . else . . . by integrating the statements of the second if-else and then recursively integrating with the parent if. This strategy also generalizes to switch statements.

We compare each statement in the then block with each statement in the else branch to build a fragment. When the statements involve only method calls, the fragment identi-

```
37    if (nothingJrb.isSelected()){
38        ConfigurationManager.setProperty(...);
39    } else if (lastJrb.isSelected()){
41        ConfigurationManager.setProperty(...);
42    } else if (LastKeepPosJrb.isSelected()){
44        ConfigurationManager.setProperty(...);
45    }
```

Listing 5: Higher level action in a conditional : Lines 37 to 45 : *"set property based on which radio button is selected"*

```
6  switch(movementType) {
7  case IEntityMovementType.MOVE_NONE: return "N";
8  case IEntityMovementType.MOVE_WALK: return "W";
9  case IEntityMovementType.MOVE_RUN: return "R";
10 }
```

Listing 6: Theme Inference for return: Lines 6 to 10: *"Return movement abbreviation based on movement type"*

fication and abstraction follows the same algorithm as for statement sequences. The crucial difference is in synthesizing a description, where instead of the plural used in the sequence case, we use the corresponding singular form. Listing 5, lines 38, 41, 44 demonstrate this basic case.

Additionally, we can also detect potential opportunities to integrate return statements and assignments to the same variable along the different branches. These two types do not manifest in the sequence abstraction case where the integrable actions will occur along the same path; a programmer would not correctly have two exact same assignments to the same variable along the same path.

Here, we describe the integration of the different statement types we could encounter in identifying and describing high level actions in different conditional branches. For return statements, we integrate based on the actual return expressions. If each return expression is a literal (e.g., return 1;), we infer a name for the *theme* as shown in Listing 6, based on the theme of the enclosing method. Our strategy enables the synthesis of *movement abbreviation* in place of the generic *string* in the abstraction. For variables, we first generate an appropriate noun phrase for each variable *v*, via the process called *lexicalize(v)* defined in [21]. We then check if the lexicalized variables can be integrated. For method calls, we check if the generated phrases for the calls can be integrated and if so, use the synthesized phrase for the abstraction.

For assignments to the *same* variable, *v*, if the assignment is not due to a method call, we abstract the fragment as, *update lexicalize(v)*. When the assignment is due to a *create or get* method, we abstract the assignment statements in the different branches as *"create or get lexicalize(v)"* as shown in Listing 7.

Describing conditional expressions. In addition to integrating the statements in the different branches, we also need to integrate and synthesize a phrase for the guarding conditional expressions. Our strategy is similar to the strategy we follow for statements except for the fact that we now need to consider clauses of the form *subject predicate object* generated for the conditional expressions (instead of verb

```
3 if ( type == PT_HTTP ){
4   tester = new NetworkAdminHTTPTester(core, l);
5 } else if ( type == PT_TCP ){
6   tester = new NetworkAdminTCPTester(core, l);
7 }else{
8   tester = new NetworkAdminUDPTester(core, l);
9 }
```

Listing 7: Similar assignment : Lines 3-9 : *"create network administration protocol tester based on type"*

phrases). The clauses are generated using [21]. We compare the generated clause for each conditional expression to see if the expressions are integrable.

If the *subject* and *predicate* of each clause is the same, then we integrate the expressions and synthesize an abstraction as:

based on what subject predicate

An example is shown in Figure 4(a) in Section 5, lines 5, 7 and 10. The synthesized abstraction is *based on what os name starts with.*

Notice the use of *what* in the abstraction. The individual clauses for each expression are propositions of the form *"osName starts with ... "*. We use *what* to transform these propositions into corresponding questions, *what does osName start with?* The heuristic uses *what* because the predicate *startsWith* begins with a verb in the third person singular.

If only the *head word* of the *subject* in each clause is the same, then we integrate the conditional expressions and synthesize the abstraction as:

based on which head word predicate object

For example, in Listing 5, the abstraction for 37, 39, 42 is *based on which* **RadioButton** *is selected.* Again notice the use of *which* in the abstraction, to help transform the proposition into a corresponding question. Also, observe the use of *RadioButton* in the synthesized phrase obtained from the type name of the receiver object in the expression method calls. Finally, the heuristic uses *which* because the predicate *isSelected* begins with the auxiliary verb, *is.*

In the default case, we integrate the expressions and synthesize an abstraction, *based on different conditions.*

3.4 Finding Traceable Patterns in Loops

Challenges. Our first challenge is to identify loops that perform common algorithmic steps like *finding, counting, copying* (e.g., Listing 1). This involves developing identification templates for each common algorithmic step in a loop, which are general enough to capture a range of code fragments that carry out these algorithmic steps. The second challenge is to develop heuristics to synthesize a smooth and succinct phrase for each identified type.

Strategy: Loop fragment identification and abstraction. We examined many loops across diverse open-source programs and developed heuristics to identify common high level actions performed by loops. As a proof of concept, we have developed and implemented rules for five different common high level actions implemented by loops. These patterns are:

max-min: Get an item in a collection with a maximum (minimum) value computed using some specified criteria (e.g., Listing 1)

count: Count items in a collection that meet a specified criteria

contains: Check if a collection contains an item that has some specified property

find: Find an item in a collection that meets some specified criteria

copy: Copy one or more items in a collection that have a desired attribute(s) to another collection

Some other algorithmic loop patterns follow immediately from the above. For example, *sum* can be seen an extension of *count*. We can extend our work by looking for other algorithmic patterns in sources like the C++ Standard Template Library and the Java library. We can also add to the catalog of common patterns by looking at introductory data structure or programming courses.

Identifying fragments with any of the five patterns. We have developed templates representing the general code structure of the five patterns described above. For example, Figure 3 is the general form of Listing 1. For a given loop within a method, we check if the loop can map to any of the algorithm templates.

```
ALGORITHM TEMPLATE:

initialize variable, max to 0
Intialize variable, maxItem to null

loop over each item i in a Collection c
 {
   set variable, current, with the return value
   from a method-call which uses i as a parameter
   if (current > max) //also: max < current
   {
    update max to current
    update maxItem to i
   }
 }
```

```
SYNTHESIS TEMPLATE:

get <item> (in <collection>) with <adjective> <criteria>
```

Figure 3: Template for the Get Maximum Algorithm Pattern

Synthesizing description for an identified pattern. For each of the five patterns, we developed synthesis templates to express the abstraction in a succinct manner. These synthesis templates rely upon the corresponding algorithmic template for the pattern. For example, the synthesis template for the max-min pattern is shown in Figure 3.

Once a loop maps to a particular algorithmic template, we generate succinct phrases for each statement in the actual loop using rules from [21]. We then look at the corresponding synthesis template for the pattern and extract elements of the synthesis template from the mapping of the actual loop values to the algorithm template. We have defined *extraction rules* for each algorithm pattern.

Extraction rules. For the max-min pattern, the extraction rules are: extract <*item*> and <*collection*> from the actual loop, using rules defined in [21]. Rules in [21] handle different ways of looping over each item in a collection, such as, using an iterator or an index from 0 to N-1, where N is the size of the collection. To extract <adjective>, scan the individual words in the words of the variable in the actual loop that corresponds to *maxItem* in the algorithm template. To extract <criteria>, scan the verb phrase generated for the

```
2  for (DrawingView v : views) {
3   if (v.getComponent() == c)
4        return v;
5  }
```

Listing 8: Use of relative clause: Lines 2 to 5: *"Find (return) drawing view whose component equals given container"*

statement that assigns to the variable in the actual loop corresponding to *current* in the algorithm template. The noun phrase in this generated phrase supplies the <*criteria*>.

Putting it all together. Consider Figure 3 and Listing 1. Our first step of mapping the actual loop to an algorithm template succeeds for the loop in Listing 1, which maps to the algorithm template for *get maximum*. To synthesize an abstraction for the loop, we extract the following based on the extraction rules:

- <*item*> is weaponAttackAction object
- <*collection*> is vectorAttacks,
- <*adjective*> is *highest* (from the variable on Line 14 in Listing 1),
- *current* is fDanger,
- *phrase* generated for the assignment to fDanger is *get expected damage*
- <*criteria*> is *expected damage* (i.e., *Noun phrase* in the phrase generated for the assignment to fDanger).

Combining these extracted entities yields the phrase shown in Listing 1.

Slight variations in the synthesis template. We have a slightly different synthesis template in cases where we can use a relative clause to synthesize a more succinct phrase. Consider the snippet shown in Listing 8. The loop maps to the algorithm template for *find*. The synthesis template for *find* is: *find item (in collection) whose/which/such that* <*criteria*>.
The relative pronoun *whose/which* is decided as follows: Given an *item* and a clause for the *criteria* of the form *subject predicate object*, if *item* is the *subject*, then the relative pronoun is *which*. If *an attribute of item* is the *subject*, then the relative pronoun is *whose*. The default uses *such that*.

For Listing 8, *item* is *drawing view* and the clause generated for the *criteria* (line 3) is *component of drawing view equals given container*. Thus, in this clause, an attribute of *item* (i.e., *component*) is the *subject*, hence the relative pronoun is *whose*, as shown in Listing 8.

4. EVALUATION

We have implemented the algorithm described in Section 3.4 in a prototype tool for identifying and describing high level actions in Java methods. In this section, we evaluate the output of our tool and examine how often our tool actually identifies high level actions when executed on 1.2 million methods across 1000 Java programs.

Specifically, we designed studies to answer the questions:

Prevalence: How prevalent are the high level actions implemented as sequence, conditional, and loops identified by our algorithm in Java software?

Detail Reduction: What is the reduction in the number of statements that need to be read by abstracting code fragments into succinct descriptions with our technique?

Precision: How precisely do we identify code fragments that implement a high level action, and how well does the synthesized description represent the high level action?

4.1 Prevalence of Identified High Level Actions

We selected a random sample of 1000 open-source Java programs from Sourceforge. Cumulatively, these programs contain over 1.2 million methods, with a median of 314 and maximum of 144,604 methods per project. We executed our prototype tool on the 1.2 million methods and gathered statistics on the frequency that the sequence, conditional, and loop high level actions were automatically detected.

Sequence. Since high level actions implemented as a simple sequence occur infrequently in small methods, we examined the frequency of detecting these sequences in methods with at least 10 statements. 155,289 (12.5%) of the 1.2 million methods contain at least 10 statements. Among these 155,289 methods, our prototype detected 17,205 (11%) methods with at least one high level action implemented as a simple sequence.

Conditional. The corpus contains 144,562 if-else statements and 17,940 switch statements. Our prototype automatically discovered 58,439 (40%) of these if-else blocks and 4,319 (24%) of the switch blocks as high level actions.

Loop. There are 162,535 loops in the corpus, of which we automatically classify 82,402 (51%) as *iterating over all items in a collection*. 12,524 (15%) of these *iterator* loops were detected as implementing an algorithmic pattern.

For this large corpus of Java methods, the frequency of the identified code fragments ranges from 11% for sequences within methods larger than 10 statements to 40% of if-else blocks. We believe these numbers are high enough to demonstrate that our automatic identification algorithm for high level actions has wide applicability.

4.2 Potential Reduction in Reading Detail

One measure of the effectiveness of our technique is to quantify how much code a human can potentially avoid reading by instead reading our synthesized high level descriptions. We focus here on how much detail is reduced or collapsed into our descriptions for each kind of statement grouping — sequence, conditional, and loop. Thus, each time we synthesize a description, we count the number of statements that were captured by our description and the number of phrases in the resulting description. While each sequence is reduced to a single phrase, conditionals are typically described by one phrase for the body (then and else parts combined) and one phrase for the set of guarding conditions. The number of phrases for the patterns implemented by loops varies with the pattern.

Across the 1.2 million methods of the 1000 programs, we computed the following *Detail Reduction* information about the reduction in reading detail achieved by our prototype:

sequence. Of the sequences implementing a high level action, there were 159,114 statements originally, which were synthesized to 35,130 phrases (22% of the original size).

conditional. The conditionals identified as a high level action contained 246,703 statements. These were reduced to 70,730 phrases (29% of original size).

loop. Loops with patterns originally contained 49,387 statements which were synthesized to 12,524 phrases (25% of original size.

These results demonstrate that a *significant* reduction in detail can be obtained with our technique.

4.3 Precision of Identification and Description

Procedure. We asked 15 human evaluators to judge the precision of our prototype on identifying and describing high level actions targeted by our technique. The programming experience of this group ranges from 2 to 20 years, with a median of 8 years. Eleven of the evaluators consider themselves as expert or advanced programmers. Seven evaluators have software industry experience ranging from 2 to 7 years.

Project	#m	kloc	Project	#m	kloc
Freecol	5971	110	Freemind	6110	94
GanttProject	4956	60	Hibernate	12793	148
HsqlDB	5150	136	Jabref	5368	100
Jajuk	2139	44	JavaHMO	1737	32
JBidwatcher	1877	30	JFtp	2379	45
JHotDraw	4267	63	MegaMek	9256	200
PlanetaMessenger	1142	22	Vuze	36372	700
SweetHome3D	4083	73			

Table 1: Subject programs in precision study. #m: # methods; kloc: 1000 lines of code.

From the 1000 open-source Java programs, we selected methods from 15 projects of different domains and executed our prototype on these methods (see selection method below). Table 1 shows the characteristics of these 15 projects.

We gave each evaluator 15 code fragments identified as a high level action by our prototype. These code fragments were drawn randomly across the methods analyzed by our tool. Each human evaluated 5 sequences, 5 conditionals, and 5 loops. To account for variation in human opinion, we gathered three separate judgements for each code fragment. Thus, we obtained 225 independent judgements on 75 identified code fragments, by 15 developers evaluating independently in groups of 3. To control for learning effects, the evaluators in a group did not examine the code fragments in the same order.

Although we are identifying and abstracting code fragments, an evaluator might need to read the entire method to judge the tool's identification and abstraction. Thus, to avoid burdening the evaluators, we selected code fragments from methods with at most 20 statements. Thus, we executed our tool on each method with up to 20 statements across the 15 programs, and collected the methods in which our tool identified a high level action. We then randomly chose 25 fragments each of sequence, conditional, and loop from this collection of methods. For loops, we randomly chose 5 fragments from each of the 5 patterns. Each group of evaluators had one fragment of each loop pattern.

We showed evaluators each code fragment assigned to them and asked them first to write an abstraction (i.e., English description) of the code fragment. To avoid bias, we deliberately did not provide an explicit definition or examples of an *abstraction*. They were allowed to use any resource to help write a description, such as the method containing the fragment, any existing comments, or the signatures of called methods. We then asked for their opinion on the following two propositions:

P1: *The fragment of statements from lines = X to Y reflects a high level action that could be expressed as a succinct phrase by a human. i.e.,*

- *There are no other statement(s) in the method that you would include in the fragment, AND*
- *there are no statement(s) in the fragment that you would exclude from the fragment.*

P2: *The description represents an abstraction of the block.*

To reduce bias, we showed our synthesized description to the evaluators after they answered *P1* and before they responded to *P2*. The evaluators were asked to respond to the above two propositions via the widely used five-point Likert scale, (1) Strongly Disagree, (2) Disagree, (3) Unsure (Neither agree nor disagree), (4) Agree, (5) Strongly Agree.

Results and Discussion. Table 2 shows the number of individual developer responses along the Likert scale. The results quite strongly suggest that the code fragments that we automatically identify as a high level action are indeed viewed as high level actions by humans. Similarly, the results indicate that we are able to synthesize descriptions that accurately represent the high level action. In 192 of 225 responses, humans *strongly agreed* that the identified code fragment represented an abstractable high level action. Only in 12 cases did humans not agree with our proposition. Similarly, 165 of 225 human responses *strongly agreed* that the synthesized description represented an abstraction of the fragment. In only 23 of 225 responses, developers did not agree or strongly agree with this proposition. When considering the majority of the three opinions for a given code fragment, for all 75 fragments examined, the majority agreed or strongly agreed that the fragment reflected a high level action (P1). For 73 of 75 fragments, the majority agreed or strongly agreed with P2, that the synthesized description represented the high level action.

	Identification				Description			
Response	**Sq**	**Co**	**Lp**	**All**	**Sq**	**Co**	**Lp**	**All**
1:S Disagree	0	1	1	2	1	0	2	3
2:Disagree	3	2	0	5	4	3	3	10
3:Neutral	0	2	3	5	1	6	3	10
4:Agree	8	5	8	21	13	13	11	37
5:S Agree	64	65	63	192	56	53	56	165
Total	75	75	75	225	75	75	75	225

Table 2: Precision results: Distribution of human judgements of high level action identification and description. Sq: sequence; Co: conditional; Lp: loop.

While the results are very encouraging, we analyzed the few fragments for which a majority of the evaluators did not agree or strongly agree with the proposition P2. In one loop, we identified the count pattern, but there is an additional object creation action which we did not include in the description. In a conditional fragment, the `then` has multiple statements each of which has a corresponding similar statement in the `else` branch. This is challenging to our current tool when attempting to produce a succinct description, and two evaluators thus responded with the *unsure* response.

We also examined the fragments where any one evaluator did not agree or strongly agree with the propositions. For identification, P1, there were 12 such cases. The interesting cases were those fragments where the evaluator wanted us to include the declaration of the variable used in a conditional or loop expression in our description.

For description, P2, 23 responses had an evaluator who did not agree or strongly agree with the proposition. 18 responses were when an evaluator did not agree or strongly agree with our proposition. Among these, the *unsure* or *disagree* was mainly due to the evaluator wanting additional information in the description. Usually this information was found in the parameter of a method call in the statement.

4.4 Summary of Results

Our study of the prevalence of detected high level actions in over 2.1 million methods indicates that our algorithm for automatically identifying code fragments that implement high level actions has wide applicability. Measuring the size of the statement groupings when we synthesize descriptions suggests a significant reduction in the details is obtained. Finally, human judgements by 15 developers strongly suggest indeed they view the code fragments that we identify as high level actions and our synthesized descriptions accurately express the abstraction.

4.5 Threats to Validity

Our results may not generalize to other Java programs or languages. To mitigate this, we downloaded 9000 most popular projects from Sourceforge, and the 1000 programs were drawn from this set. We chose our samples for study from across 15 diverse projects. In the precision study, the code fragments were drawn from methods with 20 or fewer statements for human judgement, thus our results might vary on larger programs. All our evaluators were non-novices, so our results might not hold with novices. Our reduction in reading detail provides one way to measure the amount of code that a developer may be able to avoid reading, given a high level description. It is still possible a developer would want to read at that level of detail.

5. IMPROVING CLIENT TOOLS

Another measure of this work's contribution is the impact on client tools for software maintenance. While we briefly mention other uses in the introduction, this section details some examples of how our technique can contribute to refactoring, automatic internal documentation, method renaming, and improving on our previous work in generating summary comments for methods.

Extract Method Refactoring. Fowler [7], states: *"You have a code fragment that can be grouped together, turn the fragment into a method whose name explains the purpose of the method".* The key steps are:

1. Identify code fragment to extract into a separate method
2. Identify input parameters and return type for extracted method, create a new method and replace fragment in the original method by a call to the extracted method
3. Provide a descriptive name for the extracted method

While IDEs like Eclipse support the middle step (i.e., the *actual* extraction process), there are no proven automated techniques to achieve the other two tasks. We believe that the fragments identified by our technique will broadly *correspond* to some of the *fragments* described by Fowler. More-

```
1 /** Creates a new instance. */
2 public static void main(String[] args) {
3   Application app;
4   String os = System.getProperty("os.name").toLowerCase();
5   if (os.startsWith("mac")) {
6      app = new DefaultOSXApplication();
7   } else if (os.startsWith("win")) {
8     //  app = new DefaultMDIApplication();
9      app = new DefaultSDIApplication();
10  } else {
11     app = new DefaultSDIApplication();
12  }
13
14  SVGApplicationModel model = new SVGApplicationModel();
15  model.setName("JHotDraw SVG");
16  model.setVersion("7.0.8");
17  model.setCopyright("Copyright 2006-2007 (c) ... +
18          "This software is licensed ... BY");
19  model.setProjectClassName("org.jhotdraw.samples....");
20  app.setModel(model);
21  app.launch(args);
22 }
```

(a) Original Method

```
1 /** Creates a new instance. */
2 public static void main(String[] args) {
3   Application app;
4   String os = System.getProperty("os.name").toLowerCase();
5   app = createApplication(os);
6
7   SVGApplicationModel model = new SVGApplicationModel();
8   setDifferentAttributes(model);
9   app.setModel(model);
10  app.launch(args);
11 }
```

(b) After Extract Method Refactoring

Figure 4: Using our system for refactoring

over, the abstraction phrase generated by our tool for the fragment can be used to name the extracted method.

Figure 4 shows a method main from the open-source project JHotDraw. Our technique identifies two abstractable fragments in this method. The first fragment is the conditional from line 5 to 12, while the second fragment is the statement sequence from line 15 to 19. A developer could use this output to select the two fragments to extract into methods. Our system generates the description *"create application based on what os starts with"* for the first fragment, and produces the abstraction *"set different attributes of SVGApplicationModel"* for the second fragment. The extracted method can be named using the *action* and the *theme* in the generated abstraction for the fragment from which the method was extracted. Thus, lines 5 to 12 can be replaced by a method call createApplication, while lines 15 to 19 can be replaced by a method call setDifferentAttributes. The method main after refactoring is also shown in Figure 4.

Internal Comment Generation. Studies have shown the utility of comments (including internal comments) for understanding software [22,23]. However, few software projects adequately document the code to reduce future maintenance costs [15]. Fowler recommends *ExtractMethod* refactoring over the usage of comments within a method body. However, if a developer does not want too many small methods as a result of the *ExtractMethod* refactoring, he can use our tool to automatically identify fragments and use the generated abstraction as a comment within the body of the method for the fragment. This is particularly useful for inserting internal comments into infrequently commented legacy code. Our tool could also be used to add empty lines between fragments to delineate the fragments and enhance readability, by achieving a text document paragraph-like structure.

Suggesting more informative method names. The benefits of having informative method names is well-known. Unfortunately, it is also well-known that many methods are not suitably named. We believe that our technique could be used to identify methods whose name could be made more informative. More importantly, our technique can be used to *provide* a *better* name for some methods. Our work could be used to make the *theme* of a method more descriptive. Instead of changing the method name, a developer could simply use the suggested name by our system as a comment at the call sites. From the project, *xml-cml*, consider the method with the signature, getAtom(List<CMLAtom> newAtomVector). Our system identifies the high-level action *get maximum* in this method. Thus, the system synthesizes, *get atom with heaviest atomic number* for the high-level action. From this description, one can automatically derive a better name, getHeaviestAtom for the original method.

Improving automatically generated summary comments for a method. Previously, we presented a novel technique to automatically generate comments that summarize the major actions of a Java method [21]. Our current work can be used to *significantly* improve the summarization. Consider Listing 2. Using the high-level action identification in this paper, we can produce the summary *"create oneline box. add the given components to it and return it"*.

Recognizing the high-level actions in a loop can also help generate succinct summaries. In the method getConnectionSize from project Vuze(Azureus), by recognizing the *count* pattern and using our synthesis process, a summary comment generator can produce the summary *"count transfers that have started and return count"*. We believe that this summary is a very good adjunct to the method name.

6. RELATED WORK

To our knowledge, this is the first automatic technique to identify source code fragments that implement a high level action and synthesize a natural language description to express that abstraction. However, there is work on automatically extracting topic words and phrases from source code [19,20] and clustering program elements that share similar phrases [17]. Host et al. [13] automatically extracted a verb lexicon from source code to help new programmers understand typical verb usage in practice. Hill et al. [12] developed an algorithm to automatically extract and generate verb, noun, and prepositional phrases from method and field signatures. The phrases are used to capture word context of natural language queries for software search. In this paper, we go beyond this by generating phrases for various length code fragments involving different types of statements. Additionally, these techniques do not identify code fragments that collectively implement a particular high level action within a given method.

Existing research into design recovery and reuse has also used information from identifiers [1, 5, 9]. However, all of these approaches require an expert-defined domain model or knowledge base, which is not available for all software systems or domains.

Our work might sound similar to finding "beacons" [2, 4]. A beacon can be a well known coding pattern (e.g., 3 lines for swapping array elements), meaningful identifiers, program structure, or comment statements, that signal something to the code reader wants to know about the code segment's functionality. Beacons only represent a name given to a visually recognizable entity or pattern. Our work automatically finds some of these fragments in code and generates an abstraction for the fragment. Gil and Maman [8] define the notion of traceable patterns, which are similar to design patterns, except that they are mechanically recognizable and represent lower level abstraction, (e.g., data manager or pool). Particularly, they present a catalog of 27 micro patterns, that is, class-level traceable patterns, for Java, intended for design assessment. In contrast, our work focuses on identifying high level actions within methods, providing support for different client tools.

Method extraction is primarily based on slicing; block-based slicing [24] or program transformations with slicing [16] to make the dependence-related statements contiguous for extract method refactoring. Our approach would potentially identify candidate fragments not identified through dependencies, does not require code transformations to make contiguous, and can identify good names for the extracted method using the phrases we generate.

Host and Ostvold [14] developed name-specific implementation rules mined from a large corpus of Java programs, applied the rules to identify method names that do not match their implementation, and then suggest better method names. To the best of our understanding, their work is restricted to checking if the *action (verb)* in the method name is appropriate. Our work can help in identifying if the the *theme* of the *action* is appropriate and can make suggestions to improve the *theme* part of the method name.

Lastly, this work enables automatically generated comments well beyond our previous work in summary comment generation [21] in several important ways. Our previous comment generation focused on identifying important content for the method summary and then generating text for each selected statement in isolation. This paper addresses the problem of identifying which blocks of statements form a single high level step, which is a different problem from identifying the important content for a summary. Additionally, our text generation in this work takes into account the set of statements to generate a single cohesive phrase for that step. Each statement in the identified code fragment is not analyzed in isolation but instead the set is analyzed as a group for text generation. Thus, we can now generate internal comments for statement blocks as well as use the phrases generated for high level steps to improve the abstraction level of summary comments. Other documentation generation work has focused on specific aspects of source code [3].

7. CONCLUSIONS AND FUTURE WORK

To our knowledge, we have presented the first technique for identifying code fragments of statement sequences, conditionals and loops that can be abstracted as a high level action, with the capability of also automatically synthesizing a natural language description of the abstraction. Based on 15 experienced Java programmers' opinions, we are quite encouraged by both our success in accurately identifying a widely applicable set of code fragments, and in synthesizing descriptions that humans believe accurately express the high level action.

In the future, we will continue to augment our system by examining additional potential code patterns in loops and non-loop constructs with different characteristics observed in our corpus, with attention to both identification and synthesis of succinct, informative descriptions. We plan to integrate our work in this paper into our summary comment generator to investigate the improvements made possible. We will also integrate the techniques into Eclipse and investigate usefulness for refactoring with human study. There are several other client tools that we believe we can build upon the descriptions that we synthesize for code fragments.

8. REFERENCES

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. *ICSE '93: Intl. Conf. on Softw. Engg.*, 1993.

[2] R. Brooks. Towards a theory of the comprehension of computer programs. *Intl. Journal. Man-Machine Studies*, 18, 1983.

[3] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. *ISSTA '08: Intl. Symposium on Softw. Testing and Analysis*, 2008.

[4] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. *Workshop of the Psychology of Programming Interest Group (PPIG)*, 2002.

[5] P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie—a knowledge-based Softw. information system. *ICSE '90: Intl. Conf. on Softw. Engg.*, 1990.

[6] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. *Intl. Working Conf on Mining Softw. Repositories (MSR)*, 2009.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] J. Y. Gil and I. Maman. Micro patterns in Java code. *OOPSLA '05: Conf. on Object-Oriented Prog., Systems, Languages, and Applications*, 2005.

[9] S. Henninger. Using Iterative Refinement to Find Reusable Software. *IEEE Softw.*, 11(5):48–59, 1994.

[10] E. Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration.* PhD Dissertation, University of Delaware, 2010.

[11] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools. *Intl. Working Conf on Mining Softw. Repositories*, 2008.

[12] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse. *Intl. Conf on Softw. Engg. (ICSE)*, 2009.

[13] E. W. Høst and B. M. Østvold. The programmer's lexicon, Volume I: The verbs. *SCAM '07: Intl. Working Conf. on Source Code Analysis and Manipulation*, 2007.

[14] E. W. Høst and B. M. Østvold. Debugging method names. *ECOOP: European Conf. on Object-Oriented Prog.*, 2009.

[15] M. Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Softw. Engg.*, 10(1):31–55, 2005.

[16] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. *Intl. Workshop on Program Comprehension*, 2003.

[17] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3), 2007.

[18] B. Liblit, A. Begel, and E. Sweetser. Cognitive Perspectives on the Role of Naming in Computer Programs. *Psychology of Programming Workshop (PPIG)*, 2006.

[19] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. *ISEC '08: India Softw. Engg. Conf.*, 2008.

[20] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. *MSR '05: Intl. Workshop on Mining Softw. Repositories*, 2005.

[21] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards Automatically Generating Summary Comments for Java Methods. *Intl. Conf on Automated Softw. Engg. (ASE'10)*, 2010.

[22] A. A. Takang, P. A. Grubb, and R. D. Macredie. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. *J. Prog. Lang.*, 4(3), 1996.

[23] T. Tenny. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.*, 14(9), 1988.

[24] N. Tsantalis and A. Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities. *European Conf. on Softw. Maintenance and Reengineering (CSMR)*, 2009.