Mining Additions of Method Calls in ArgoUML

Thomas Zimmermann\textsuperscript{1}  
Silvia Breu\textsuperscript{2}  
Christian Lindig\textsuperscript{1}  
Benjamin Livshits\textsuperscript{3}

\textsuperscript{1} Dept. of Computer Science  
Saarland University  
Saarbrücken, Germany  
{tz, cl}@st.cs.uni-sb.de  
\textsuperscript{2} University of Cambridge  
Computer Laboratory, UK  
silvia@ieee.org  
\textsuperscript{3} Dept. of Computer Science  
Stanford University  
Stanford, USA  
livshits@cs.stanford.edu

ABSTRACT

In this paper we refine the classical co-change to the addition of method calls. We use this concept to find usage patterns and to identify cross-cutting concerns for ArgoUML.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—version control; D.2.9 [Management]: Software configuration management

General Terms

Management, Measurement

1. INTRODUCTION

One of the most frequently used techniques for mining version archives is co-change. We specialize this concept to the addition of method calls:

Two method calls that are added together in the same transaction, are related to each other.

We use the concept of co-additions for the following two tasks:

- Find usage patterns, such as “the methods \texttt{containsNode} and \texttt{containsEdge} are frequently called together.”
- Identify cross-cutting concerns, such as “the first statement of every method calls the \texttt{info} method to log the method name.”

In Section 2 we will describe our input data and the tools we used; we present our results for usage patterns in Section 3 and for cross-cutting concerns in Section 4.

2. INPUT DATA AND TOOLS

We applied our mining techniques to the ArgoUML repository that was supplied for the MSR challenge [4]. We restricted our analysis to the \texttt{src} directory that contains the actual source code of ArgoUML. All data was collected with an extended version of the eROSE plug-in [2] for the ECLIPSE environment. For mining, we used SQL queries and the Xelopes data mining library [5].

To reconstruct transactions we use the sliding window approach with a window size of 200 seconds. For each transaction we compute the set of newly added method calls. For this we compare the total set of method calls from the actual and the previous transaction. The total set of method calls is computed for each transaction by traversing the abstract syntax trees of all affected files.

For a call expression \( c_1 . c_2 (\ldots c_n ) \) we only take the final method call \( c_n () \) into account. Since we only analyze one file at a time, the full signature for method \( c_n \) isn’t available. Instead, we augment it with the number of parameters, as shown in Table 1. Analyzing single files rather than complete snapshots makes our preprocessing cheap, as well as platform- and compiler-independent.

3. MINING USAGE PATTERNS

Our approach is based on an observation: Method calls that are added to source code simultaneously often represent a pattern. To identify such patterns, we performed frequent pattern mining on the set of added method calls.

We focused our analysis on intra-procedural patterns: patterns that occur within a single method. In terms of mining this means that we do not use complete transactions as input but group transactions by the method in which a call was added. Furthermore, we ignored calls to frequently used JAVA methods, such as \texttt{iterator.hasNext} and \texttt{toString}, since patterns involving these methods are well-known.

Table 1 shows the patterns we mined, sorted by decreasing frequency. Actual usage patterns are printed in boldface, thus the precision is 40\%. Below we discuss a few examples.

- \texttt{addModelEventListener}.
- \texttt{removeModelEventListener}

This pattern is used when elements are changed. First, the listener is removed for the old element, then the element is changed, and finally the listener is added for the new element.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>localize(2) addField(2)</td>
<td>57</td>
</tr>
<tr>
<td>localize(1) lookupIcon(1)</td>
<td>45</td>
</tr>
<tr>
<td>addCaption(4) addField(4)</td>
<td>43</td>
</tr>
<tr>
<td>addButton(1) lookupIcon(1)</td>
<td>41</td>
</tr>
<tr>
<td>localize(1) addField(2)</td>
<td>28</td>
</tr>
<tr>
<td>findFigsForMember(1) findType(1)</td>
<td>23</td>
</tr>
<tr>
<td>addModelEventListener(2) removeModelEventListener(2)</td>
<td>19</td>
</tr>
<tr>
<td>addModelEventListener(3) removeModelEventListener(3)</td>
<td>13</td>
</tr>
<tr>
<td>addFocusListener(1) addKeyListener(1)</td>
<td>12</td>
</tr>
<tr>
<td>hasMoreElements(0) nextElement(0)</td>
<td>12</td>
</tr>
<tr>
<td>error(2) debug(1)</td>
<td>11</td>
</tr>
<tr>
<td>addSeparator(0) addField(2)</td>
<td>10</td>
</tr>
<tr>
<td>info(1) isEnabled(0)</td>
<td>10</td>
</tr>
<tr>
<td>max(2) isDisplayed(0)</td>
<td>9</td>
</tr>
<tr>
<td>containsNode(1) containsEdge(1)</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1: Usage patterns for ArgoUML.
Illegal arguments. The transaction with the log message "Made the methods look a little more alike. Collected the numerous IllegalArgument calls in methods. [...]" inserted many cross-cutting calls to IllegalArgument or one of its variants. These calls are always last in the method body:

```java
public String getValueOfTag(Object handle) {
    if (handle instanceof MTaggedValue) {
        return ((MTaggedValue) handle).getValue();
    }
    return illegalArgumentString(handle);
}
```

In this case the method illegalArgumentString throws an IllegalArgumentException and returns a null Object. Most of the 262 calls to illegalArgument methods could have been realized as aspects.

Instance of a thing. The transaction with the log message "Replace every single instance of something instanceof MThing with ModelFacade.isAThing(something)" inserted many isA calls to the source code. isA methods look as follows:

```java
public boolean isAClassifier(Object handle) {
    return handle instanceof MClassifier;
}
```

There exist 111 methods of the above form; these methods could have easily been generated with aspects.

In our previous work [1] we showed that mining cross-cutting concerns from version archives has a high precision, for the top 20 aspect candidates of ECLIPSE we reached up to 90%. Measuring recall requires knowing all aspect candidates, which is typically only possible for a few small benchmark projects.

5. CONCLUSION

Co-addition of method calls identifies usage patterns; a usage pattern may be actually a cross-cutting concern when all locations where calls were added call the same set of locations. Both usage pattern and cross-cutting concerns can be identified by mining version archives, as demonstrated by the ones we found in ArgoUML.

Usage patterns and cross-cutting concerns have several benefits. Mining usage patterns can locate defects in software and supports program understanding. Knowing cross-cutting concerns helps to reduce maintenance effort and is the prerequisite for refactoring a legacy system into an aspect-oriented design.

For a more detailed description of our mining approaches, we refer to our publications on finding usage patterns [3] and identifying cross-cutting concerns [1].

6. REFERENCES