# Mining Additions of Method Calls in ArgoUML

Thomas Zimmermann[1] Silvia Breu[2] Christian Lindig[1] Benjamin Livshits[3]

[1] Dept. of Computer Science, Saarland University, Saarbrücken, Germany
{tz, cl}@st.cs.uni-sb.de

[2] University of Cambridge, Computer Laboratory, Cambridge, UK
silvia@ieee.org

[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 `containsNode` and `containsEdge` are frequently called together."
- *Identify cross-cutting concerns,* such as "the first statement of every method calls the `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 `src_new` 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.

| Pattern | Count |
|---|---|
| localize(2) addField(2) | 57 |
| localize(1) lookupIcon(1) | 45 |
| addCaption(4) addField(4) | 43 |
| addButton(1) lookupIcon(1) | 41 |
| localize(1) addField(2) | 28 |
| findFigsForMember(1) findType(1) | 23 |
| **addModelEventListener(2) removeModelEventListener(2)** | **19** |
| **addModelEventListener(3) removeModelEventListener(3)** | **13** |
| **addFocusListener(1) addKeyListener(1)** | **12** |
| **hasMoreElements(0) nextElement(0)** | **12** |
| error(2) debug(1) | 11 |
| addSeperator(0) addField(2) | 10 |
| **info(1) isInfoEnabled(0)** | **10** |
| max(2) isDisplayed(0) | 9 |
| **containsNode(1) containsEdge(1)** | **8** |

**Table 1: Usage patterns for ArgoUML.**

## 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 `iterator`, `hasNext`, and `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.

- `addModelEventListener`, `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.

```
if (Model.getFacade().isAElement(target)) {
   Model.getPump().removeModelEventListener
      (this, target);
}
target = t;
if (Model.getFacade().isAElement(target)) {
   Model.getPump().addModelEventListener
      (this, target, "name");
}
```

- `addFocusListener`, `addKeyListener`
  This pattern indicates a relationship between the focus and a key listener: A user may enter text only to graphical elements that are in focus.
- `isInfoEnabled`, `info`
  Sometimes the return value of `isInfoEnabled` is checked before the `info` method is called.

```
if (LOG.isInfoEnabled()) {
   LOG.info("Removing feature " + feature);
}
```

- `containsNode`, `containsEdge`
  These two methods are frequently called with the same arguments to check whether an edge is valid; if not, an error is logged.

```
if (!containsNode(destModelElement)
      && !containsEdge(destModelElement)) {
   LOG.error("some message");
   return false;
}
```

## 4. MINING CROSS-CUTTING CONCERNS

Programs can be modularized in only one way at a time. Aspect-oriented programming (AOP) remedies this by factoring out aspects and weaving them back in a separate processing step. For existing projects to benefit from AOP, these cross-cutting concerns must be identified first. This task is called *aspect mining*.

Our hypothesis is that *not all cross-cutting concerns exist from the beginning, but some emerge over time*. By analyzing where developers add code to a program, we can identify cross-cutting concerns. Our approach searches transactions for sets of locations $L$ where at each location calls to a set of methods $M$ have been added. In other words: The calls to methods $M$ are spread throughout source code locations $L$. We call such a pair $(M, L)$ an *aspect candidate*. In order to identify aspect candidates that actually cross-cut a considerable part of a program, we ignore all candidates $(M, L)$ where $|L| < 7$ or $|M| \cdot |L| < 20$. This means that each aspect candidate has to cross-cut at least 7 locations, and it has to comprise at least 3 method calls that got added.

For ArgoUML we identified 230 aspect candidates in 73 out of 6,286 transactions. Below we discuss a few examples.

**Logging.** We observed that the transaction with the log message *"Replaced deprecated log4j Category with Logger."* inserted several calls to methods `debug`, `error`, and `warn`. The last two methods turned out to be false positives. However, for `debug` we found several cross-cutting calls that logged the method names as shown in the following example:

```
public void doAction(int oldStep) {
   LOG.debug("doAction " + oldStep);
   ...
}
```

This logging could have easily been realized with an aspect.

**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:

```
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:

```
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-cuttings 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

[1] S. Breu and T. Zimmermann. Mining Aspects from History. Submitted for publication.

[2] eROSE. Guiding Programmers in Eclipse. http://www.st.cs.uni-sb.de/softevo/erose/.

[3] V. B. Livshits and T. Zimmermann. Dynamine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proc. Europ. Software Engineering Conf./ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 2005.

[4] MSR. Mining Challenge 2006. http://msr.uwaterloo.ca/challenge/.

[5] Prudsys AG. XELOPES Library. http://www.prudsys.com/Produkte/Algorithmen/Xelopes/.