Detecting Inefficient API Usage

David Kawrykow and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{dkawry, martin}@cs.mcgill.ca

Abstract

Large software projects often rely on third-party libraries, made accessible through Application Programming Interfaces (APIs). We have observed many cases where APIs are used in ways that are not efficient. We developed a technique to automatically detect inefficient API usage in software projects. The main hypothesis underlying the technique is that client code that imitates the behavior of a library method without calling it is likely not to use the library as efficiently as possible. In addition to warning developers of potentially inefficient API usage, our technique also indicates how to improve the use of the API. Application of the technique on Java open-source systems revealed many cases of inefficient API usage, and corresponding recommendations that led to code improvements.

1. Introduction

Large software systems generally rely on reusable collections of implemented functionality called software libraries. Functionality provided by libraries is made available through Application Programming Interfaces, or APIs. In Java, for example, an API is the set of classes, methods, and fields that can be accessed by *client code* (the code making use of the library).

We have observed many cases where use of a library's API can be considered inefficient. In particular, if a sequence of calls to the methods (or functions) of a library could be replaced by a shorter or more appropriate one, then the original sequence is not the most efficient use of that library's API.[1]

Inefficient API usage can have many negative effects on the quality of a system, including: convoluted client code, decreased modularity, obscured intent of a statement's purpose, suboptimal performance, and eventual obsolescence. Although the immediate impact of inefficient API usage is difficult to assess, its long-term effect on the maintainability of a system is unlikely to be good.

We propose a technique for finding a particular class of inefficient API usage in source code. The main hypothesis underlying our technique is that client code that imitates the behavior of a library method without calling it is probably not using the library as efficiently as possible: the client code could call the method it imitates.

Our technique works by analyzing a target system (and its libraries) to discover any method in the project code that imitates the behavior of a library method. Our definition of the *imitates* relation relies on an abstraction of the behavior of both client and library code. Detected cases of potentially inefficient API usage are grouped together into *potentially inefficient library usage patterns*. These include the name of the imitated library method, which can serve as a *recommendation* to improve the code. Detected instances of inefficient usage patterns correspond to actual cases of potentially inefficient usage, which can be inspected, and in many cases corrected. Our approach has been implemented to analyze projects in the Java programming language.

We applied our approach to popular open-source Java projects to assess its practicality and usefulness. Our experiments to date led to the detection of 61 valid patterns of potentially inefficient usage in nine systems. In turn, these patterns led to the discovery of close to 200 instances of inefficient API usage.

The contributions of this work include the description of a new, fully-implemented technique for detecting inefficient API usage, and initial data documenting its usefulness.

2. Motivating Example

We illustrate the need for an approach to detect inefficient API usage with a concrete example taken from the source code of JasperServer v. 3.0.0, an open-source business intelligence platform.[2] In our example, references

[1] Our use of the term *inefficient* is thus not related to the run-time performance of the client code.

[2] http://sourceforge.net/projects/jasperserver

to library APIs are in **bold**, and client code is in normal monospaced typeface. The example shows that API imitations cannot always be detected trivially using code comparison.

In JasperServer, client method `HttpUnitTest.gettingURLResponse` uses the library class **`WebConversation`** of package **`com.meterware.httpunit`** inefficiently by re-implementing one of its **`getResponse`** methods. The body of `gettingURLResponse` is given below

```
gettingURLResponse(String url) {
  WebResponse response = null;
  URL serverUrl = new URL(url);
  conversation = new WebConversation();
  request = new GetMethodWebRequest(serverUrl,"");
  return conversation.getResponse(request);
}
```

An alternate implementation is given below:

```
gettingURLResponse(String url){
  return new WebConversation().getResponse(url);
}
```

However, discovering that the original implementation is equivalent to the above method is not trivial. A look at the source code of **`WebConversation.getResponse(...)`** does not immediately indicate why the two implementations are interchangeable:

```
getResponse(String url){
  return _mainWindow.getResponse(url);
}
```

In fact, among other reasons, the actual implementation of the response behavior is delegated through two levels of method calls, making it unlikely that this equivalence could be discovered through traditional program similarity detection.

3. Approach

The idea behind our approach is to detect cases where the quality of source code can be improved by replacing the imitating code by the library method it imitates. The complete approach works as follows:

1. The source code of a library client and the *byte code* of its libraries is analyzed to determine if any client method imitates the behaviour of a library method. We compare each method within a given source file in the client code against all accessible library methods imported by the file. We consider a client method's usage of a library class to be *potentially inefficient* if it *imitates* the code of a library method without calling the library method itself.
2. All detected cases of potentially inefficient usage are grouped together into *potentially inefficient library usage patterns* that describe both the library method being imitated and *how* that method is imitated.
3. All the cases corresponding to a given pattern are filtered for invalid or trivial imitations according to a set of heuristics.
4. A software engineer inspects each remaining pattern to determine whether it truly corresponds to an actual imitation of the suggested library method. Validated patterns are stored in a file.
5. Any project using libraries for which patterns are documented, including client projects scanned in steps 1-4, can then be scanned for instances of these patterns.
6. A developer can then inspect the reported instances to perform the suggested perfective maintenance based on the recommended library methods and their imitations in the client.

Detecting Library Imitations. Detecting cases where client code imitates the behaviour of a library method requires us to compare program behaviour. Because we seek to determine when different pieces of code are approximately equivalent, and because we compare client source to library byte code, we rely on abstracted representations of method bodies to do our comparison. Specifically, we abstract a method body as the set of fields, methods, and types referenced by the method.

We establish behavioural equivalence by comparing individual elements in method bodies. We say that an element x **matches** an element y if x **is identical to** or **specializes** y. Our concept of specialization is slightly adapted from the type theoretical definition of substitutability to fit our goal of estimating potential replacement of client code with library methods. For space reasons we illustrate one aspect of our concept of specialization with a case related to our previous example (Section 2): we say constructor **`WebRequest(URL,String)`** specializes constructor **`WebRequest(String)`** (defined in the same class) because the shorter version of the constructor simply calls the longer version with a `null` value for the first parameter. Hence any call to the shorter version can always be replicated (or *matched*) by a call to the longer one.

Our goal is to find a match between each element in a library method and a client method. Because early experimentation clearly demonstrated that it was too simplistic for our purposes to only match elements directly, we developed three different strategies for matching the elements abstracting a library method: direct match, indirect match, and nested match.

An element is **directly matched** by a method m if it is a member of the set of elements abstracting m. An element is **indirectly matched** by a method m if it is a member of

the set of elements abstracting a method *called* by m. Finally, an element e is **nested matched** by a method m if e is a method and all the elements in the set abstracting e are directly or indirectly matched by m. Given these definitions, we say that a library method m_l **is imitated** by a client method m_c if all the elements in the set abstracting m_l are matched by m_c.

For each type of match, our technique produces a *match descriptor* that describes both the matched element and the *way* in which that element was matched (e.g., directly or indirectly or nested, through specialization or not).

Distinguishing the way we match elements allows us to define a **superior to** ordering between matches. For example, indirect matches are superior to nested matches.

Given an imitation of a library method m_l by a client method m_c, the pattern corresponding to that imitation is the set of match descriptors for that imitation. We note that a given element $e \in m_l$ might be matched in multiple ways within m_c. In such cases, the pattern associated with the imitation of m_l by m_c includes only the most superior match.

Filtering Heuristics. When first testing our approach, we observed that many patterns returned by the algorithm represented trivial or unusable results. After partitioning these unusable results into different categories, we were able to design several filtering heuristics to remove them from the final set of patterns presented to developers. For example, we filter out cases where a client method m_c indirectly calls the method being imitated, because in such cases the behaviour of the client code is unlikely to be inefficient (the imitated method is being used after all).

Detecting Pattern Instances. Patterns found in the detection phase of the approach are stored in a file in XML format. When scanning a project for instances of inefficient API usage, patterns are loaded from their XML representation. The client source code is again scanned, but this time it is compared only against loaded patterns. If a client method matches a loaded pattern in an equivalent or superior way, then it is said to contain an instance of that pattern.

4. Preliminary Results

Given its heuristic nature, the development of our approach has involved considerable experimentation. We summarize here the findings of our most recent study.

For our approach to be effective and practical, the patterns it discovers in a code base need to be identified with reasonable accuracy and efficiency, and to lead to tangible improvements in the source code's quality. We assessed whether our approach meets these two criteria by applying it on a per-library basis to ten large-scale open source projects totalling over 1.7 million lines of code (see Table 1 for system names). The target systems were different from the ones we used to develop our approach and refine our heuristics. We scanned each target project with respect to its most commonly-used libraries to automatically identify potentially inefficient library usage patterns. We then manually classified each discovered pattern as either *valid* or *invalid* by systematically going through a validation checklist. Patterns were deemed valid if the abstracted imitation embodied by those patterns could feasibly constitute actual imitations in the client code. We also assessed the instances of all patterns classified as valid. We proceeded by examining the client methods associated with the pattern instances. In each case we determined whether the client method could make use of the imitated library method and, if so, how this might improve the quality of the source code. Any instance associated with a valid improvement was classified as valid. Instances were rejected as invalid when no obvious improvements were associated with them, or if we had any doubts about the correctness of the code resulting from the recommendation.

Table 1. Experimental Results

Project	Det. Pat.	Val. Pat.	Det. Ins.	Val. Ins.
JBoss 4.3.2	113	29	200	163
SpringFramework 2.5.5	24	5	6	1
ArgoUML 0.2.0	13	2	2	0
Hibernate 3.2	22	2	12	12
iReport 3.0.0	12	5	5	4
JasperReports 3.0.0	34	3	4	0
JasperServer 3.0.0	20	9	21	13
Freemind 0.8.1	0	0	0	0
Jajuk 1.6.2	10	1	1	1
Checkstyle 4.4.2	38	5	10	2
Total	286	61	261	196

We found valid patterns in nine of the ten target projects scanned. Table 1 provides a summary of our findings. Columns show the detected (Det.) and valid (Val.) patterns (Pat.) and instances (Ins.), respectively. Each pattern took on average approximately one minute to classify. This time includes the effort to find library source code, when available, and to record the result; it took considerably less time to invalidate a pattern than to validate it. For the nine projects in which patterns were detected, the precision of the analysis varies between 9% (Hibernate and JasperReports, 2/22 and 3/34, respectively) and 45% (JasperServer, 9/20).

Table 1 also presents the results of our instance classification. The fourth column lists the number of instances associated with valid patterns, and the fifth column lists the number of valid instances. We see that 75% of all pattern instances yielded tangible improvements to the quality of the source code.

Performing the assessment summarized in Table 1 required us to manually inspect every single result produced by our approach. We were thus able to gain a deep understanding of the nature and practical usability of the results and of the risks and limitations of the approach, through the study of numerous valid and invalid results. Section 2 illustrated a case actually detected with our technique.

5. Related Work

A wide variety of tools and techniques has been proposed to help developers detect and mitigate problems in source code. These techniques range from those modeling program execution [9] to those combining static and dynamic analysis [1]. In practice, many static checkers work by detecting situations that are likely to be associated with errors. For example, Xie and Engler [10] detect nonsensical redundancies in source code and show that they are usually indicative of hard bugs. Our approach works in a similar way, but finds inefficient usage based on imitations instead of bugs based on redundancies.

Among the copious work on defect detection, a number of tools specifically address API-related problems [4, 7]. Tools have also been developed to help prevent API-related bugs in the first place by finding usage examples in existing code [8, 2]. Although these techniques can all be effective in dealing with API-related problems, none specifically attempt the detection of inefficient API usage.

Other work relevant to the problem of improving library usage includes work on the detection of code similarity for the purpose of finding useful methods when initially writing or porting code [6]. In contrast, we seek to recommend alternate, more efficient API usage for client code based on structural *imitations* appearing within that code.

Code similarity analysis also guides example recommendation tools such as Strathcona [3]. These tools abstract a developer's code content and context as a set of structural facts and use these facts to extract similar code fragments from large repositories. Like Strathcona, we detect code similarity at the granularity of element references, although we do so with extended matching relations intended to capture sophisticated imitations.

Finally, there exists a wide array of techniques for detecting duplicate code (code clones) within projects [5]. However, clone detection in general detects *duplicate code* rather than *imitations*, so it would not detect the cases we detect using specialization or indirect and nested matches.

6. Summary

We proposed a novel approach to automatically detect inefficient API usage. Our approach involves automatically detecting code that imitates API methods without calling them. We detect such imitations by extending existing code similarity detection techniques with new matching relations between client and library methods. Our approach also abstracts the detected imitations as *patterns*, allowing future analyses to benefit from previously validated discoveries.

We have completely implemented our approach and preliminary results showed that we were able to detect numerous actual cases of inefficient library usage in the code of popular open-source systems. We found that the recommendations associated with the detected inefficiencies lead to many tangible improvements in the quality of the code.

Motivated by these encouraging results, we plan to increase the scope of our approach to capture more complicated method imitations. To this end, we will experiment with additional matching heuristics. We also plan to investigate ways to reduce the number of false positives in the set of generated patterns, and to study the actual impact of detected imitations on the maintenance of software systems.

Acknowledgments. The authors would like to thank Barthélémy Dagenais and the anonymous reviewers for comments on this paper. This work is supported by NSERC and FQRNT.

References

[1] E. Bodden, P. Lam, and L. Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *Proc. 16th ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering*, pages 26–47, 2008.

[2] B. Dagenais and M. Robillard. Recommending Adaptive Changes for Framework Evolution. In *Proc. 30th Int'l Conf. on Software Engineering*, pages 481–490, 2008.

[3] R. Holmes, R. Walker, and G. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Trans. Software Engineering*, 32(12):952–970, 2006.

[4] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Proc. Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[5] R. Koschke. *Software Evolution*, chapter 2. Identifying and Removing Software Clones, pages 15–36. Springer, 2008.

[6] A. Michail and D. Notkin. Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In *Proc. 21st Int'l Conf. on Software Engineering*, pages 463–472, 1999.

[7] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. 6th European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 35–44, 2007.

[8] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proc. 3rd Int'l Workshop on Mining Software Repositories*, pages 54–57, 2006.

[9] Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *Proc. 32nd Symp. on Principles of Programming Languages*, pages 351–363, 2005.

[10] Y. Xie and D. Engler. Using Redundancies to Find Errors. *IEEE Trans. Software Engineering*, 29(10):915–928, 2003.