2009 IEEE/ACM International Conference on Automated Software Engineering

Improving API Usage through Automatic Detection of Redundant Code

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

***Abstract*—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 the most effective. We developed a technique and tool support to automatically detect such patterns of API usage in software projects. The main hypothesis underlying our technique is that client code imitating the behavior of an API method without calling it may not be using the API effectively because it could instead call the method it imitates. Our technique involves analyzing software systems to detect cases of API method imitations. In addition to warning developers of potentially re-implemented API methods, we also indicate how to improve the use of the API. Applying our approach on 10 Java systems revealed over 400 actual cases of potentially suboptimal API usage, leading to many improvements to the quality of the code we studied.**

***Index Terms*—API usage; code quality; code analysis; recommendation system**

I. 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/API).[1]

We have observed many cases where client code uses an API in ways that are clearly not the most effective. In particular, it is not unusual to observe client code that independently re-implements a service provided by an API.

This phenomenon is not surprising when we consider the variety of possible causes. First, APIs can grow very large and complex, and developers using them may not discover all the functionality they offer. Second, in many cases, APIs evolve independently from the projects that rely on them, and developers may remain unaware of improvements to the API that could translate into improvements to their code. This situation is compounded by a third factor: APIs sometimes evolve in a backward-compatible fashion, without any element being annotated as deprecated. This way, updating an API will not break the code, and therefore will not always attract the attention of developers to potentially different ways of using the API. As we illustrate in Section II, detecting unused functionality can be far from obvious. Consequently, as a project and its associated libraries co-evolve, the client code is at risk of degrading in quality through the ever-growing presence of needlessly re-implemented library methods.

The re-implementation of services offered through APIs can have many negative effects on the quality of a system, including: decreased modularity, convoluted client code, obscured intent of a statement's purpose, suboptimal performance, and eventual obsolescence. Although the immediate impact of suboptimal API usage is difficult to assess, the long-term effect on the maintainability of a system is unlikely to be good. In fact, "Don't make the client do anything the library could do" is an explicit guideline for API designers, and according to Bloch, violating this rule leads to code that is "annoying and error-prone" [1]. Ideally, for a system to be well-maintained, client code should use APIs as effectively as possible.

We propose a static analysis-based technique for finding a specific class of client code whose usage of an API can be improved. The main hypothesis underlying our technique is that client code that imitates functionality provided by a method of an API can potentially be improved by calling the method itself, instead of imitating it.

Our technique works by analyzing a target system (and its libraries) to discover any method in the client 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 imitations are grouped together into potential *API usage patterns*, which include the name of the API method that is imitated, provided as a *recommendation* to improve the code. A software engineer can then inspect the imitations corresponding to each pattern to determine whether these truly correspond to re-implementations of API services. We have completely implemented our automatic detection technique, along with additional tool support for visualizing the results, as a plug-in for the Eclipse Platform.[2]

We applied our approach to ten open-source Java projects to assess its practicality and usefulness. We analyzed our target systems to collect potential imitations of library methods, and manually inspected every reported imitation to assess

[1] In many contexts, the terms "API" and "library" can be used interchangeably. We chose to use the term "API" except when specifically referring to an API's implementation, in which case the term library is more appropriate.

[2] http://www.eclipse.org

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.62

its validity. For valid imitations, we measured how much code improvement could be derived from knowledge about its corresponding usage patterns. Our experiments led to the detection of over 400 valid imitations within the target systems. Adapting the target systems to instead use the recommended API yielded savings of over 650 method calls. Our experiments also demonstrated the scalability of our approach, by supporting the analysis of over 1.5 million lines of source code in just over 7 minutes.

A brief overview of this approach, together with preliminary results, has been showcased in a 5-minute presentation and short report [2]. This paper contributes the first complete report on our new, fully-implemented technique for improving API usage through automatic detection of redundant code, with a considerably expanded description of the technique, new tool support, and a detailed analysis of our completed experiments.

In the rest of this paper, we illustrate the motivation for our work with an example in Section II; We then present the details of our approach (Section III) and describe our experimental setup and results (Section IV). We conclude with a discussion of related work (Section V) and a summary of the paper (Section VI).

II. Motivating Example

We illustrate the need for an approach to improve API usage with a concrete example taken from the source code of JasperServer v. 3.1.0, an open-source business intelligence platform.[3] In our example, references to APIs are in **bold**, and client code is in normal monospaced typeface. The example shows that API method imitations cannot always be detected trivially using code comparison.

In JasperServer, the client method `HttpUnitTest.gettingURLResponse` does not use the API provided by the library class **`WebConversation`** of package **`com.meterware.httpunit`** in the most effective way possible because it re-implements one of its **`getResponse`** methods. The body of `gettingURLResponse` is given below

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

A more effective implementation is given below:

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

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

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

The source code of the actual **`getResponse`** method used by the client also fails to justify the similarity:

```
getResponse(WebRequest request){
  return _mainWindow.getResponse(request);
}
```

It is only by examining these inner `getResponse` methods that the equivalence becomes apparent. However, the fact that the response behavior is delegated through two levels of method calls, among other complications, makes it unlikely that the equivalence could be discovered through traditional program similarity detection.

III. Approach

The idea behind our approach is to detect cases where the quality of source code can be improved by replacing code that imitates an API method by a call to the method itself. Our current implementation supports the analysis of Java programs. The complete approach works as follows:

1) The source code of a library client and the byte code of its libraries are abstracted to a common representation. We abstract library *byte* code because the source code of a library is not always available or included within client projects.
2) The abstracted client code is compared against the abstracted library code to determine if any client method imitates the behavior of a library method without calling the method itself. We compare each method within a given client file against all library methods which are visible to the type(s) declared in the file.
3) All detected imitations are grouped together into *API usage patterns* that describe both the library method being imitated and *how* that method is imitated.
4) All imitations corresponding to a given pattern are filtered for invalid or trivial cases according to a set of heuristics.
5) A software engineer validates each pattern by inspecting the client code of the remaining imitations to determine whether these truly correspond to actual imitations of the suggested library method. Validated patterns are stored in a file.
6) Any project using an API for which valid patterns are documented can be scanned for instances of these patterns. For the client projects analyzed in steps 1–4, these instances correspond to imitations classified in step 4.

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

7) A developer can inspect the reported instances to perform the suggested perfective maintenance based on the recommended library methods and their imitations in the client.

In the rest of this section, we provide the technical details and rationale for each step of the approach, including our mechanism for matching the behavior of client code with that of library methods (Section III-A), the heuristics we use to filter detected imitations (Section III-B), and our process for reusing patterns (Section III-C). The automated portions of our approach have been fully implemented as a standalone Eclipse plug-in, described in Section III-D.

A. Detecting Imitations

Detecting cases where client code imitates the behavior of a library method requires us to compare program behavior. Because we seek to determine when different pieces of code are approximately equivalent, we rely on abstracted representations of method bodies to do our comparison.

Abstracting Method Bodies: Similar to other approaches that rely on source code equivalence comparisons (e.g., Strathcona [3]), we abstract a method body as the set of program elements referred to in the corresponding code. Specifically, we abstract a method body as the set of fully-qualified signatures of the fields, methods, and types referenced by that method body.

The body of the imitated `getResponse(String url)` method from our motivating example in Section II is abstracted as the following set of element references:

```
_mainWindow, WebWindow.getResponse(String),
WebWindow, WebResponse, String[4]
```

In this example, the `WebWindow` type appears because it is the declared type of the `_mainWindow` field.

Comparing Individual Elements: Establishing behavioral equivalence requires us to compare individual elements in method bodies.

Definition 1 (Element match): 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 general type theoretical definition of substitutability to fit our goal of estimating potential replacement of client code with library methods.

Definition 2 (Element specialization): If y is a type, then x **specializes** y if x is a subtype of y. If y is a field, then x cannot specialize y. If y is a method then x specializes y if x implements or overrides y or if x overloads y and its parameter types form an ordered superset of the parameter types of y.

[4]We present a simplified form of each element to conserve space.

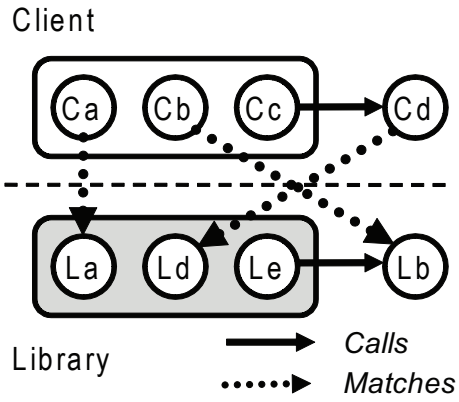

Fig. 1. Matching Types

For example, the constructor `GetMethodWebRequest(URL,String)` specializes `GetMethodWebRequest(String)`, but the reverse is not true. The intuition behind our definition is that the shorter version of the method often calls the longer version with some default value for the extra parameter. Similarly, if client code calls the longer version, it may also be using the default value as the additional parameter, and could actually use the shorter version instead. This case is found in the motivating example.

Matching Elements in Method Bodies: Our goal is to find a match between each element in a library method and a client method. Because early experimentation 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.

We introduce a simple operation to simplify the following descriptions. The $\mathbf{method}(m)$ function is a predicate that returns true if m is a method (as opposed to a field or type). The $\mathbf{body}(m)$ function returns the set of program elements M corresponding to the abstraction of m's body.

Definition 3 (Direct Match): Given a method body M and an element e, there *exists* a **direct match** of e within M if $\exists\ m\ \in\ M$ such that m matches e.

This is the simplest kind of match. Figure 1 represents different matching scenarios (rectangles represent abstracted method bodies, and circles represent referenced elements). In the figure, there is a direct match of La in client method $\{Ca, Cb, Cc\}$ because Ca matches La (i.e., Ca is either identical to La or specializes it).

Definition 4 (Indirect Match): Given a method body M and an element e, there *exists* an **indirect match** of e within M if $\exists\ m\ \in\ M$ such that $\mathrm{method}(m)$ and there exists a direct match of e within $\mathrm{body}(m)$.

In this case we say that e is either "indirectly matched" or "specialized" via m, depending on what element in m is matching e.

In Figure 1, there is an indirect match of Ld within the client method because Cc calls Cd, which matches Ld. This strategy was designed to help detect cases where imitations were distributed across methods. Note that the definition applies regardless of whether m is a client or library method.

Definition 5 (Nested Match): Given a method body M and an element e, there *exists* a **nested match** of e within M if $\mathrm{method}(e)$ and there exists a direct or indirect match within M of every element in $\mathrm{body}(e)$.

In Figure 1, there is a nested match of Le within the client method because Le only calls Lb, which is matched by Cb. This strategy was designed to help detect nested cases of imitation, i.e., when client code *imitated* a method call made by a library method, instead of *matching* it.

Definition 6 (Imitation): Given two method bodies M_1 and M_2, M_1 **imitates** M_2 if $\forall e \in \mathrm{body}(M_2)$, there exists a match of e within M_1.

According to this definition, Figure 1 describes a situation where $\{Ca, Cb, Cc\}$ imitates $\{La, Ld, Le\}$.

We note an important distinction between imitations and nested matches. A nested match only allows direct and indirect matches, whereas an imitation includes direct, indirect, *and* nested matches. We included the restriction on nested matches to prevent deep nested matches. Experimentation with nesting levels showed that nesting imitations beyond a single method call resulted in the detection of very few additional imitations, all of which were of dubious validity.

In our motivating example (Section II), the elements of `WebConversation.getResponse(String)` are matched through all three types of matches. The client code contains a direct match of the `String` reference, an indirect match of the `_mainWindow` field access via the method `WebConversation.getResponse(WebRequest)`, and a nested match of the method `WebWindow.getResponse(String)`.

Representing Matches and Patterns: A *match descriptor* is a data structure that comprises both a matched element and the *way* in which that element is matched. Different types of matches have slightly different descriptors.

Definition 7 (Direct Match Descriptor): A **direct match descriptor** of an element e is a tuple (e, r), where $r \in \{\text{identical}, \text{specialized}\}$.

For example, if a type reference to `ArrayList` specialized a type reference to `Collection`, the tuple (`Collection`, specialized) would embody this direct match between the two elements.

Definition 8 (Indirect Match Descriptor): An **indirect match descriptor** of an element e is denoted as a tuple (e, r) where $r \in \{\text{indirect identical}, \text{indirect specialized}\}$.

Definition 9 (Nested Match Descriptor): A **nested match descriptor** within M of a method call m is the set of all direct and indirect matches of the element references in $\mathrm{body}(m)$ occurring within M.

We introduce an ordering between matches so that we are later able to distinguish between patterns that are likely to be valid and those in which we have less confidence.

Definition 10 (Match Order): We define a **superior to** ordering as follows: indirect matches are superior to nested matches, and direct matches are superior to both indirect and nested matches. Furthermore, we say that identical matches (direct or indirect) are superior to specialized matches.

Definition 11 (API Usage Pattern): Given an imitation of library method M_l by client method M_c, the pattern corresponding to that imitation is the set of direct, indirect, and nested match descriptors for that imitation.

The pattern associated with the imitation in our motivating example is presented below:

```
WebClient.getResponse(String)
 -_mainWindow: indirect iden.
 -String: identical
 -WebWindow.getResponse(String): nested
  -WebWindow.getResponse(WebRequest): indirect iden.
  -new GetMethodWebRequest(String): specialized
```

We note that the `getResponse` method above is declared by `WebClient`. The familiar `WebConversation` type from the motivating example is a direct subtype of the abstract `WebClient` type and does not override the latter's `getResponse` methods. We present the `WebClient` type in this example to stay consistent with the actual recommendations made by our tool, described in Section III-D.

We note also that a given element $e \in M_l$ might be matched in multiple ways within M_c. For example, a `List` type reference in M_l might be directly specialized in M_c with a reference to `ArrayList` and indirectly matched by an indirect reference to `List`. In such cases, the pattern associated with the imitation of M_l by M_c includes only the most superior match descriptor. In our case, we retain the (`List`,specialized) descriptor.

Two different imitations of the same library method are abstracted by the same pattern whenever both imitations can be described using identical sets of the most superior match descriptors. We emphasize that a given descriptor does *not* include the client code element that was used to match the descriptor's library element. For example, if another client method M_c' specializes the `List` reference in M_l with a reference to `LinkedList`, then we retain an identical

(List,specialized) descriptor. Similarly, if M_c references both ArrayList and LinkedList, the same single descriptor would be retained.

B. Filtering Heuristics

We prototyped our approach on the source code of LIMEWIRE, an open-source file-sharing platform.[5] During this phase, we observed that many imitations returned by the technique represented trivial or unusable results. After partitioning these unusable results into different categories, we were able to design a number of filtering heuristics to remove them from the final set of imitations presented to developers.

Most of the heuristics test for simple conditions. For example, we ignore API methods that reference only fields and types because such methods are too easily imitated. We also ignore cases where a client method specializes the API method it imitates, because in such cases the client method is clearly meant to provide extended or refined behavior and not the behavior it was meant to specialize. Similarly, we ignore cases where a client method indirectly references the method it imitates because in such cases the client method is probably extending the imitated method, rather than imitating it.

The following five heuristics improve the accuracy of the approach in more complex cases. We report on the effectiveness of these heuristics in Section IV. In the following descriptions, we assume that a client method M_c has been found to imitate an API method M_l.

Restrict Nested Matches: If M_l calls a method m and m calls no methods of its own (but contains only field accesses and type references), then we do not allow m to be matched through a nested match. We found that allowing such method calls to be nested matched leads to many meaningless results. For example, the String method getLength(), which only accesses the field count, can be nested matched by calling many other String methods, such as trim() or charAt(int), all of which also access count, but for different purposes. Library methods which call getLength() are probably not imitated by clients that only call trim().

Emphasize Types in the Signature of M_l: If M_c does not directly match all types in the signature of M_l (including the declaring and return types), then a call to M_l by M_c is likely to be specious because there are, for example, no actual objects in M_c that could be passed as input arguments to M_l. This filter is motivated by the LIMEWIRE library method Argument.setValue(int). A client method was found to imitate setValue using code similar to

```
Action action = ...;
action.setArgumentValue("string1",int);
action.setArgumentValue("string2","string3");
```

There is no reference to an Argument instance in this imitation. In fact, the client method does not directly reference an Argument instance at all. This makes it unclear how a call setValue could replace the above code snippet, or how it could be made in the first place.

Keep Apart Getters and Setters: If the name of M_l starts with the prefix "get", then no field access made by M_l may be indirectly matched via a method call with method name prefixed with "set". This filter includes several other prefix pairs, among them "add" vs. "get", and "put" vs. "contains". This filter is motivated by the LIMEWIRE library method Packet.setFrom(String). This method sets the field from to a modified form of the input String. A client method was found to imitate setFrom by calling code similar to

```
Packet packet = ...;
String x = parseBareAddress(packet.getFrom());
```

Although the client method also accesses the from field, it does so using getFrom. This makes it unlikely that the from field is accessed in the same way as if the client were to call the recommended setFrom method.

Reject Included Methods: If M_c calls some other method m and $\text{body}(M_l) \subset \text{body}(m)$, then we reject the imitation of M_l. The intuition behind this heuristic is that the "larger" method probably performs more services than those provided by the "smaller" method, and is likely to be the more appropriate method to use. This filter is motivated by cases such as the methods errors() and addError(Test,Throwable) in the class junit.framework.TestResult. The bodies of both methods contain the elements

```
Vector, Enumeration, fErrors, elements()
```

but the body of addError also contains additional references. If a client calls the "larger" addError method then it is probably adding an element to fErrors and it is unlikely that this call should be replaced by a call to the "smaller" errors method, which only returns the elements found in fErrors. Although this example shows m and M_l to be from the same class, in general we do not require m to be in the same library as M_l.

Avoid Repeated Match Descriptors: If M_c imitates two library methods M_l^1 and M_l^2, and the imitations for both methods can be described using the same set of match descriptors, then both imitations are rejected. This situation usually indicates that the match descriptors are too simplistic to capture possible differences between M_l^1 and M_l^2, and should not be used to justify the reported imitations. In LIMEWIRE, this heuristic allowed us to ignore a number of methods from the Device class, all of which had the general form:

```
getPropertyX(){
  return getDeviceNode().getProperty("X");
}
```

[5] http://www.limewire.org

Whenever a client method called one or more of these methods, all others were automatically considered imitated.

C. Storing and Reusing Patterns

Patterns found in the detection phase of the approach are persisted to disk in a pattern description file. Each pattern encodes the library method being imitated, the element references being matched, and how those references are matched (Section III-A). When scanning a project for imitations, patterns are loaded from their pattern description file at run time. The client source code is again scanned, but this time it is compared only against loaded patterns. A client method is said to contain an instance of a given pattern if the client method imitates the pattern's encoded library method using matches which are equivalent or superior to those encoded by the pattern.

D. Tool Support

We implemented support for our approach in the form of iMaus, a tool for improving API usage. iMaus allows users to detect potential API method imitations within their Eclipse projects and to investigate the structure and quality of reported imitations. Users can leverage iMaus to discover new imitations within their projects or to detect instances of previously validated patterns. iMaus can be used to scan either a single file, or an entire package or project. In this way, if a user detects a valid pattern in one file, the user can re-scan the rest of the project to detect further instances of this pattern in other files. iMaus also provides mechanisms to group together similar imitations based either on their underlying patterns or the set of matches describing each imitation. Groups of (invalid) imitations can be eliminated in a single operation. In addition, iMaus allows users to extract the patterns abstracting detected imitations and to store and reuse these to scan projects more effectively in the future (because only instances of validated patterns are detected). iMaus implements the static analysis required by the detection heuristics by parsing and analyzing Java source and byte code. Byte code is abstracted using the asm bytecode analysis framework[6] and source code by following standard practice that leverages Eclipse's JDT component. We focus our description of iMaus on its user interface, which has undergone in-house usability prototyping to ensure a design that would minimize the cost of inspecting recommendations.

Reported imitations are stored as a list within an *Imitation View*. Each imitation in the list is presented as a pair of method names: the name of the API method being imitated and the name of the client method that imitates it. Users can display the details of an imitation by double clicking on its entry in the view (see Figure 2).

Individual imitations are displayed with the help of the standard Eclipse editor and a separate *Imitation Content View*. The view displays *what* elements are being matched, while the editor displays *how* these are matched by the client

[6]http://asm.ow2.org

Fig. 2. Viewing Imitations with iMaus

method. In particular, the view displays the fields and methods referenced by the body of the imitated library method, as well as the the fields and methods referenced by the bodies of all nested matched method calls. For example, in Figure 2, the *Imitation Content View* displays the fields and methods referenced by the imitated `getResponse` method described in the motivation: a reference to the `_mainWindow` field and a call to the method `WebWindow.getResponse`. The method call is nested matched and its referenced elements also displayed: a call to a `GetMethodWebRequest` constructor and a call to an overloaded `WebWindow.getResponse` method. We note that the `getResponse` method is indeed declared by `WebClient`, an abstract super type of the `WebConversation` type presented in the motivating example.

The editor displays the imitating client method and underlines all method calls made by the client method that contribute to the imitation. A method call contributes to an imitation if it directly or indirectly matches a reference. We note that if the same method call appears multiple times within the client method, each call is underlined. We also note that if multiple method calls contribute matches for the same element reference, all those contributing the most superior matches for that element are underlined. The user must decide which of these are relevant to the imitation, if any. In Figure 2, the `GetMethodWebRequest` constructor referenced by the client method is underlined because it specializes the `GetMethodWebRequest` constructor called by the `WebWindow.getResponse` method.

A user can view how each underlined method call contributes to the imitation by hovering the cursor over that method call. Conversely, the user can click on matched elements in the *Imitation Content View* to display which method calls are used to match them. In our figure, an investigation of the `GetMethodWebRequest` constructor in the *Imitation Content View* leads the user to the (highlighted) `GetMethodWebRequest` constructor.

TABLE I
TARGET SYSTEMS

Project	Version	kLOC	Dlds. (K)
JBoss	5.0.1	539	13 879
SpringFramework	2.5.5	247	3322
Hibernate	3.3.1	166	4746
ArgoUML	0.20	165	N/A
iReport	3.0.0	144	1716
JasperReports	3.1.4	131	1672
JasperServer	3.1.0	79	249
FreeMind	0.9.0	53	7512
Jajuk	1.7.1	45	296
Checkstyle	4.4.2	20	3604

IV. EVALUATION

Our evaluation sought to determine *whether our approach efficiently supports improvements of API usage through the detection of redundant code*. For our approach to be practical, the imitations it detects need to:

1) be identified (and filtered) with reasonable accuracy and efficiency.
2) lead to concrete improvements in the source code's quality.
3) generate reusable patterns.

We assessed whether our approach meets these criteria by applying iMaus to ten large-scale open source projects. These projects were different from the ones we used to develop our approach and refine our heuristics. Each project was scanned with respect to its imported libraries to automatically identify potential library method imitations. In this phase, imitations detected for each project were filtered using all of the heuristics described in Section III-B. Following a systematic inspection approach, we then manually classified each reported imitation as either valid or invalid. We collected and classified all imitations which were filtered by the heuristics. We used the results of this process to assess the first criterion. For each valid imitation, we recorded the method calls saved by replacing the imitation with a call to the recommended library method. We used these results to assess the second criterion. Finally, we counted the number of reusable patterns that could be inferred from the valid imitations in order to assess the third criterion.

A. Target Systems

For our set of target systems we retrieved ten popular open-source Java projects from the SourceForge repository.[7] The ten systems we chose are all actively maintained and widely used. The names of the systems and their version, size, and overall number of downloads (Dlds.) are presented in Table I (download statistics, when available, were taken from SourceForge on 26 April 2009). All projects comprise over 20kLOC, and hence preclude the possibility of manual source code inspection as a practical alternative for detecting library imitations.

[7]http://sourceforge.net

TABLE II
IMITATION CLASSIFICATION

Project	# Valid Imitations	# Reported Imitations	Precision
JBoss	341	974	35%
SpringFramework	7	177	4%
Hibernate	12	153	8%
ArgoUML	15	269	6%
iReport	2	17	12%
JasperReports	5	29	17%
JasperServer	17	61	28%
FreeMind	2	8	25%
Jajuk	1	3	33%
Checkstyle	4	37	11%
Total	405	1322	31%

B. Imitation Classification

We classified reported imitations as either valid or invalid using iMaus (described in Section III-D). We considered a reported imitation of a given library method by a given client method valid if

1) the library method was truly being imitated by the client method.
2) the true imitation was discovered by assessing client code explicitly underlined by iMaus.

We included the second criterion to ensure that the rationale given by the approach was used as the primary justification for the discovery of true imitations.

Assessment of the first criterion required us to verify if client methods were re-implementing services provided by recommended library methods. We did this by examining the underlined client elements, the signature of the library method and its referenced elements (all provided by iMaus). We perused the library method's API documentation and/or online source code whenever necessary and available. We always rejected imitations whenever it was not completely clear that the client method was indeed imitating the library method, or if client code reused variables used to store results of intermediate method calls. If we found that the code of the client method imitated the suggested library multiple times, we recorded this as a single valid imitation.

Valid Imitations: Table II lists the total number of imitations reported for each target system, as well as the number of imitations which we classified as valid. We found a total of 405 valid imitations, with at least one valid imitation in every system, and as many as 341 within a single system (JBOSS). The precision of the analysis varies between 4% (SPRINGFRAMEWORK) and 35% (JBOSS). The overall precision of the approach is 31% and the average per-system precision is 21%. Table IV in Section IV-D records the number of patterns found in each system. Systems for which the total number of reported imitations was above 100 all featured patterns that instantiated a high number of invalid imitations. In SPRINGFRAMEWORK

for example, 152 of the 170 invalid imitations are instances of just 5 patterns.

Valid imitations often included specialization, or indirect and nested matches. Of the 405 valid imitations, 15 included at least one nested match and 173 included at least one nested or indirect match. Direct specialization contributed to 7 of the 405 imitations, while indirect specialization did not appear in any imitations.

Effectiveness of the Filtering Heuristics: Across the ten target systems, our filtering heuristics rejected a total of 2123 imitations. Of these, only 20 were actual valid imitations. Without the heuristics, the overall precision of our approach is 11% (vs. 31% when the heuristics are applied), while the average per-system precision is 8% (vs. 21%).

Classification Effort: The amount of time taken to scan projects was above one minute only for JBOSS (110s) and SPRINGFRAMEWORK (69s), which are composed of over 6500 and 3100 client files, respectively. The times reported include the time needed to abstract all the necessary client and class files, as well as the time needed to detect and filter imitations between client and imported library methods. These measures do not include the time required to classify individual imitations. By using iMaus, we found that an individual imitation usually took no more than one minute to classify, with most invalid imitations requiring just 10 to 15 seconds. Many invalid imitations were as obvious as the one presented at the end of this section, and could be rejected almost immediately.

Classification Methodology: To help ensure an overall objective classification process, each reported imitation was independently classified by a member of our research group whose work is not related to the present research. This additional inspector was provided our plug-in, and trained for 45 minutes on how to use it. He was given instructions to reject imitations whenever their validity could not immediately be inferred from the underlined client elements.

Out of the 3851 classified imitations for the two portions of the evaluation (1728 reported imitations and 2123 filtered imitations respectively), only 63 were given conflicting labels. From this result, we infer that the potential error on our manual classification is about $\pm 2\%$.

Ignoring Irrelevant Method Calls: In the case of valid imitations, iMaus often method calls that were not actually relevant to the imitation itself (see Section III-D). For example, in JASPERSERVER, the library method `View.getActionButton(String)` has the following body

```
AbstractButton getActionButton(String x) {
 TitleBar tbar = getTitleBar();
 return tbar!=null? tbar.getActionButton(x):null;
}
```

and is imitated by a client method using the following code, where the method calls are underlined by iMaus:

```
view.getTitlebar().addAction(...);
view.getTitlebar().getActionButton(...)
```

We see that iMaus underlines an irrelevant call to `getTitleBar` that is not related to an actual imitation of `getActionButton`. However, the reported imitation is still considered valid because the other two calls constitute a valid imitation. We ignored the presence of such irrelevant method calls and classified an imitation as valid as long as a subset of the underlined method calls described a true imitation.

Rejected Imitations: We conclude this section with a typical rejected imitation found in JASPERSERVER. In that project, our approach detects that a library method `Util.equal(String,String,boolean)` is imitated by a client method. iMaus displays the following element references for the library method:

```
String.equals(Object),
String.equalsIgnoreCase(String)
```

The underlined method calls displayed in the editor are found in the following two lines of client code:

```
ConsoleAppender x = new ConsoleAppender(...);
if(args[i].equals(...))
```

iMaus reveals that the constructor is used to indirectly match the `equalsIgnoreCase` method call. In this case, it is likely that the client method is not imitating the library method because the name of the constructor suggests that it does not exclusively compare two `String` instances.

C. Code Quality Improvements

To assess our second criterion we measured improvements to the quality of the target systems' source code induced by our recommendations. To this end we recorded the number of method calls which could be saved by adapting client code to use the suggested library method instead of its re-implementation. As with any software quality metric, our model for measuring quality improvement resulting from recommendations is not perfect. In particular, it does not take into account situations where redundant API usage code would actually be more readable, or display some other desirable characteristic. However, we believe our quality improvement model is fit for our experimental purpose because it objectively captures cases where more of an API's functionality and intent is used, which we hypothesize is a desirable goal in most cases.

We counted all those method calls which would no longer be needed if the client code were modified. Method calls which were erroneously underlined by iMaus (i.e., which did not actually contribute to the imitation) were not counted. If a client method contained several imitations, then we counted

TABLE III
SAVINGS

Project	# Valid Imitations	# Method Calls Saved
JBoss	341	527
SpringFramework	7	7
Hibernate	12	33
ArgoUML	15	15
iReport	2	3
JasperReports	5	16
JasperServer	16	36
FreeMind	2	9
Jajuk	1	1
CheckStyle	4	15
Total	405	662

TABLE IV
DETECTED PATTERNS

Project	Reusable (Strong)	Singleton	Weak
JBoss	26 (16)	16	1
SpringFrmwk	1 (1)	1	1
Hibernate	2 (1)	0	0
ArgoUML	3 (2)	5	2
iReport	1 (1)	1	0
JasperReports	0 (0)	5	0
JasperServer	6 (5)	4	0
FreeMind	2 (0)	0	0
Jajuk	0 (0)	1	0
CheckStyle	1 (0)	3	0
Total	36 (25)	36	4

the method calls saved *per* imitation. Additional savings, such as variable declarations, the use of literals and if-statements, were not recorded. In our motivating example, the client saves two method calls: one to the `URL` constructor and one to the `GetMethodWebRequest` constructor.

Table III summarizes our findings. We see that we could save at least one method call in each of the target systems, with as many as 527 saved within a single system (JBOSS). Most imitations allowed clients to save anywhere from one to four method calls. Some imitations yielded higher savings because they appeared multiple times within the same client method.

D. Pattern Detection

To assess our third criterion we grouped all valid imitations into patterns and recorded the types of patterns found for each project in Table IV. The type of each pattern was determined by considering its instances across all ten projects. We defined four types of patterns: weak, reusable, strong, and singleton patterns. A pattern was labeled "reusable" if at least two of its instances represented valid imitations. If a pattern was reusable and all of its instances were valid, then the pattern was also labeled "strong". A pattern was labeled "weak" if multiple instances of the pattern were found within the projects, and only one instance represented a valid imitation. Those patterns for which a single, valid instance was found were labeled as "singletons".

In total we found 76 patterns across the ten systems. Of these, only four were weak, while the remaining 72 were evenly divided between singleton and reusable patterns. Of the reusable patterns, 25 were also strong. The detected patterns describe a total of 546 instances, 405 of which are valid (see Table II) and 141 invalid.

In Table IV, the total number of Reusable patterns (36) is slightly less than the sum of Reusable patterns across the individual systems (43) because four patterns were found in multiple systems. In this initial experiment, an across-system overlap of four patterns is a modest indication of the reusability of patterns across projects, but this relatively low figure can be explained by the fact that patterns are tied to individual libraries, and our target systems implement widely different functionality using a total of over 100 different libraries. The results in Table IV do suggest that if a user discovers a valid instance (and its associated pattern) by scanning a single project file, then scanning the entire project with respect to this pattern will often yield at least one more valid imitation elsewhere (valid instances of a given pattern were spread across multiple files in almost all cases). When this is not the case, the most likely alternative is that the pattern will detect no further imitations (in which case the user need not investigate any further). We found that only four of the 76 discovered patterns would have yielded only invalid imitations. We also note that our implemented approach indeed allows users to scan individual files (see Section III-D).

E. Qualitative Analysis

We conclude this section with two examples that illustrate how the output of our approach leads to improved API usage and the elimination of redundant code. We also summarize reasons why imitations were rejected.

We were able to replace the following block of code below in four separate cases in JASPERSERVER

```
UploadFileSpec[] uploadFile;
uploadFile = new UploadFileSpec[]
                {new UploadFileSpec(file)};
uploadForm.setParameter("string",uploadFile);
```

with

```
uploadForm.setParameter("string",file);
```

The latter implementation is more effective because it avoids redundant construction of an array and an `UploadFileSpec` instance. In ARGOUML, we were able to replace the following block of code

```
Collection nodes = ...;
Iterator it = nodes.iterator();
while (it.hasNext())
 sm.select((Fig) (it.next()));
```

with

```
Collection nodes = ...;
sm.select(nodes);
```

The overloaded `select` method saves the client two method calls and an unnecessary while-loop.

In JBOSS, we were able to replace the following code block:

```
Thread th=Thread.currentThread();
ClassLoader loader=th.getContextClassLoader();
Class tC=null;
try {
 tC=Classes.getPrimitiveTypeForName(type);
 if( tC == null )
    tC=loader.loadClass(attrType);
} catch(ClassNotFoundException ignore) { }
PropertyEditor editor=null;
if( tC != null )
 editor=PropertyEditorManager.findEditor(tC);
```

with

```
PropertyEditor editor = null;
try {
 editor = PropertyEditors.getEditor(type);
} catch (ClassNotFoundException ignore) { }
```

The latter implementation saves four method calls and avoids two null checks. It is also much easier to understand than the previous implementation. Our approach also detected our motivating example in Section II, which appears four times in JASPERSERVER.

The most common rejected imitations were similar to the one shown in Section IV-B, i.e., the meaning of method names used by the client method made it clear that the client performed a task different from the one offered by the library method. Many imitations were deemed invalid because the client method's control structure required the individual method calls to be used instead of a single call to the library method. Often this structure took the form of temporary variables used to store the results of method calls appearing in the pattern, the contents of which would then be accessed for additional processing not related to the imitation in question. In a few cases, the fact that elements matched between client and library methods were not in the same order in the code invalidated the result.

F. Discussion

Value of the Approach: The current version of our approach is the result of extensive iterative development aimed at maximizing the number of detected imitations (through sophisticated matching mechanisms), while limiting the number of false positives (through filtering heuristics).[8] At this point we believe that the results reported in Tables II and III indicate that the approach has crossed the threshold of practical usability. It is important to note that the precision levels reported in Table II cannot be compared directly with that of more standard bug detectors such as FindBugs [4], since newly detected valid imitations yield potentially reusable *patterns*. As opposed to individual instances of bugs or other issues in the code, it makes sense to invest more effort vetting new imitations (patterns) as the cost of their detection and validation can be spread over the number of instances found in other systems or in subsequent locations or versions of a system. In our preliminary investigation in Section IV-D, we found that instances of four out of 76 patterns are actually present in more than one target system. While we expect most patterns to yield maximal benefits for a single evolving system, our results suggest that some validated patterns can indeed be reused to detect imitations in other systems. Moreover, we found that the total number of invalid instances of validated patterns (141 out of 545) is low enough to suggest that developers will not have to scan through long lists of useless recommendations to find ways to improve their code, if that code is scanned against validated patterns. Finally, although very few cases of actual library imitations were found in five of our ten target systems, this is simply an indication that the types of usage pattern we detect were not found on these systems, possibly because they are too recent, too small, well-maintained, or because the developers are proficient with the APIs they use. In any case, we expect that our approach will be maximally beneficial for very large systems that have had a significant evolution (such as JBOSS).

Limitations of the Approach: A tacit limitation of our approach is that it only detects a specific class of cases where API usage can be improved: those involving client methods that imitate functionality provided by individual library methods. Other types of improvements, and in particular usage of a library that severely decreases run-time performance, will not be detected by our approach. In addition, given its heuristic nature, our approach also detects cases that cannot lead to clear code improvements (i.e., where replacing code with the imitated library method would lead to an incorrect transformation). For this reason, it is paramount that users of the approach understand the nature of the recommendations and not blindly perform changes based on the results.

Threats to Validity: Two main factors potentially impact the results reported in this section: our choice of target projects and the manual assessment of the results of our approach. We applied our approach to ten open-source projects that may not be representative of all software systems. The relative youths of the system we considered, and the number of programmers typically looking at the source code of open-source systems, give us confidence that the results we obtained may in fact represent a low estimate of what we can find in many large, long-lived systems. With respect to the manual classification of imitations, we limited the risk of investigator bias by cross-checking our classifications against those made by a second investigator.

Future Work: Our approach can be further improved by reducing the number of false positives in the set of de-

[8]This experimentation was conducted on systems that were not used as part of the evaluation.

tected imitations. This can be achieved through improvements to the filtering heuristics based on the experimental results themselves. For example, our approach could determine if temporary variables storing intermediate results are reused by client methods, and filter those cases where this is true. We have also considered abstracting method bodies as ordered lists instead of sets, although initial exploration of this idea did not indicate that it would necessarily yield improvements that would justify the increased complexity.

V. Related Work

A wide variety of tools and techniques has been proposed to help developers detect and mitigate problems in source code. These range from techniques modeling program execution [5] to those combining static and dynamic analysis [6]. Lint [7], a tool developed to help enforce type rules in C code, is an early representative of lightweight static checking techniques. In practice, many static checkers work by detecting situations that are likely to be associated with errors. For example, Xie and Engler [8] detect nonsensical redundancies in source code and show that they are usually indicative of hard bugs. Our approach works in a similar way, but seeks to improve API usage based on imitation instead of bugs based on redundancies.

Among the copious work on defect detection, a number of tools specifically address API-related problems. For example, FindBugs [4] and PR-Miner [9] analyze API usage patterns within projects to detect inconsistencies (bugs) in those projects. Tools have also been developed to help prevent API-related bugs in the first place by finding usage examples in existing code [10], [11]. SemDiff [12] provides support for adapting client code to updated libraries by analyzing how the code of the library evolved. Although SemDiff could also be used to improve API usage, it can only make recommendations based on patterns of updates to the library itself. The issue of API deprecation has been studied by Perkins [13], who proposes that API-deprecation be addressed by automatically in-lining deprecated API elements in source code. Although these techniques can all be effective in dealing with API-related problems, none of them specifically addresses the problem of improving API usage by eliminating redundant code.

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. For example, Michail and Notkin [14] use name standardization and free-text indexing to compare methods (and classes) across libraries. Similarly, CodeBroker [15] relies on comments and identifiers to recommend components for reuse. The textual content of identifiers that refer to standard Java API elements in client code has also been proposed as an indexing scheme for that API [16]. In contrast, we seek to recommend alternate, more effective 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] and XSnippet [17]. 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. Our results (Section IV-B) demonstrate that additional imitations were discovered by means of the extended relations. Other tools addressing example recommendations, such as Prospector [18] and ParseWeb [19], detect code that exhibits object-instantiation sequences based on user queries, rather than code similarity. In contrast to these tools, we recommend single API methods based on imitations, as opposed to implicit or explicit queries by a developer.

Finally, there exists a wide array of techniques for detecting duplicate code (code clones) within projects [20]. Most approaches in this area require the source code of the project analyzed, which is not necessarily available when analyzing libraries. One well-known approach proposed by Baxter et al. [21] uses similarity within the abstract syntax trees of code fragments to find duplication at the textual level. Like most approaches in this area, Baxter's approach scans the source code of the project being analyzed, which is not necessarily available when analyzing libraries. Furthermore, as clone detection in general detects *duplicate code* rather than *imitations*, it would not detect the cases we detect using specialization or indirect and nested matches.

VI. Summary

API imitations can arise through a number of factors and typically add unnecessary complexity to client code, a known hindrance to maintainability. We proposed a novel approach to automatically detect such API imitations in software projects. 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 abstractions of the implementation of client and library methods. Our approach also includes a mechanism for abstracting the detected imitations as *patterns*, thus allowing future analyses to benefit from previously validated discoveries. Patterns encode a reference to the imitated library method, so that this method can be recommended to developers when an imitation is detected. We implemented our approach, along with additional tool support for visualizing detected imitations and reusing validated patterns, in an Eclipse plug-in called iMaus.

We evaluated the usefulness of our approach by applying it to 10 open-source systems comprising over 1.5 million lines of Java code. This investigation showed that our approach was able to detect over 400 actual imitations of library methods in the code of popular open-source systems. Moreover, the recommendations associated with the detected imitations led to many concrete improvements in the quality of the client code.

ACKNOWLEDGMENTS

The authors extend special thanks to Tristan Ratchford for his help with the evaluation, and for comments on the paper. The authors also thank Barthélémy Dagenais, Ekwa Duala-Ekoko, and the anonymous reviewers for comments on the paper. This work was supported by NSERC.

REFERENCES

[1] J. Bloch, "How to design a good API and why it matters," in *Companion to the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006, pp. 506–507.

[2] D. Kawrykow and M. P. Robillard, "Detecting Inefficient API Usage," in *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering—Companion Volume*, 2009, pp. 183–186.

[3] R. Holmes, R. Walker, and G. Murphy, "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, 2006.

[4] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Proceedings of the 19th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004, pp. 132–136.

[5] Y. Xie and A. Aiken, "Scalable Error Detection Using Boolean Satisfiability," in *Proceedings of the 32nd Symposium on Principles of Programming Languages*, 2005, pp. 351–363.

[6] E. Bodden, P. Lam, and L. Hendren, "Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time," in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008, pp. 26–47.

[7] S. Johnson, "Lint, a C Program Checker," Bell Telephone Laboratories, Tech. Rep. 65, 1978.

[8] Y. Xie and D. Engler, "Using Redundancies to Find Errors," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 915–928, 2003.

[9] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[10] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, pp. 25–34.

[11] T. Xie and J. Pei, "MAPO: Mining API Usages from Open Source Repositories," in *Proceedings of the 3rd International Workshop on Mining Software Repositories*, 2006, pp. 54–57.

[12] B. Dagenais and M. P. Robillard, "Recommending Adaptive Changes for Framework Evolution," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 481–490.

[13] J. Perkins, "Automatically Generating Refactorings to Support API Evolution," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005, pp. 111–114.

[14] A. Michail and D. Notkin, "Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 463–472.

[15] Y. Ye and G. Fischer, "Supporting Reuse by Delivering Task-relevant and Personalized Information," in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 513–523.

[16] H. Ma, R. Amor, and E. Tempero, "Indexing the Java API Using Source Code," in *Proceedings of the 19th Australian Conference on Software Engineering*, 2008, pp. 451–460.

[17] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for Sample Code," in *Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006, pp. 413–430.

[18] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to Navigate the API Jungle," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 48–61.

[19] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 204–213.

[20] R. Koschke, *Software Evolution*. Springer, 2008, ch. 2. Identifying and Removing Software Clones, pp. 15–36.

[21] I. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the 14th IEEE International Conference on Software Maintenance*, 1998, pp. 368–377.