MARBLE: Mining for Boilerplate Code to Identify API Usability Problems

Daye Nam[*], Amber Horvath[*], Andrew Macvean[†], Brad Myers[*], and Bogdan Vasilescu[*]
[*]Carnegie Mellon University, USA
{dayen, ahorvath, bam}@cs.cmu.edu; vasilescu@cmu.edu
[†]Google, Inc., USA
amacvean@google.com

***Abstract*—Designing usable APIs is critical to developers' productivity and software quality, but is quite difficult. One of the challenges is that anticipating API usability barriers and real-world usage is difficult, due to a lack of automated approaches to mine usability data at scale. In this paper, we focus on one particular grievance that developers repeatedly express in online discussions about APIs: "boilerplate code." We investigate what properties make code count as boilerplate, the reasons for boilerplate, and how programmers can reduce the need for it. We then present MARBLE, a novel approach to automatically mine boilerplate code candidates from API client code repositories. MARBLE adapts existing techniques, including an API usage mining algorithm, an AST comparison algorithm, and a graph partitioning algorithm. We evaluate MARBLE with 13 Java APIs, and show that our approach successfully identifies both already-known and new API-related boilerplate code instances.**

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

// DOM code to write an XML document to a specified output stream.
private static final void writeDoc(Document doc, OutputStream out) throws
    ↪ IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().
        ↪ getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out));
  }
  catch(TransformerException e) {
    throw new AssertionError(e); // Can't happen!
  }
}
```

Listing 1. Writing an XML document to a specified output stream in Java may involve significant boilerplate code for initialization and error handling [9].

I. Introduction

Almost all modern software programs adopt and use a large number of APIs. Therefore, dimensions of API usability, including learnability, effectiveness of use, and error-proneness, are increasingly becoming significant concerns for API designers [1]–[3]. To investigate API usability issues and to improve APIs, researchers have used several methods such as lab studies [4] and API design reviews [5], [6]. The understanding gained from such studies, along with the insights from experienced API designers, have led to the development of guidelines for API designs and heuristics for evaluating APIs [1]–[3], [7]. However, despite these efforts, many APIs are still difficult to use [8]. In particular, API designers have reported that anticipating how developers will use their API in the wild is difficult and leads to usability challenges when developers use the API in unexpected ways [7]. API designers have also reported significant trouble discovering what are the usability barriers at scale [7]. Although online sources such as Stack Overflow and GitHub may contain ample amounts of real client code or insights into how programmers perceive APIs, designers report that there are not so many automated approaches to mine usability data from these repositories at scale, nor to gauge the severity of the usability issues [7].

In contrast, mining software repositories techniques have long been used to identify API usage patterns [10]. For example, existing API usage pattern mining tools such as ExampleCheck [11], [12] and PAM [13] automatically identify API methods that are frequently called together in client code. Primarily, these tools have been designed to help users learn a new API, by identifying idiomatic usage examples, as well as to help API designers gain insights into how their APIs are being used. In this paper we argue that API usage pattern mining tools may also help reveal certain API usability issues.

Specifically, we focus on one particular grievance that developers express repeatedly [14]–[17] in online discussions about APIs (and programming languages more generally): *boilerplate code*. Wikipedia [18] refers to boilerplate as "sections of code that have to be included in many places with little or no alteration", and code "the programmer must write a lot of to do minimal jobs." One Stack Overflow user [17] calls boilerplate "any seemingly repetitive code that shows up again and again in order to get some result that seems like it ought to be much simpler"; most users agree that boilerplate is tiresome to write and error-prone [14]–[16]. Listing 1 shows a typical example: Whenever one wants to write an XML document to a specified output stream in Java, which is a common usage scenario, this requires significant boilerplate code. One could imagine that this use case could be accomplished natively by calling a single API method such as `writeXML`.

From an API designer's perspective, *the existence of boilerplate code may serve as an indicator of poor API usability*. This is because the need for boilerplate code often indicates that the API does not directly provide the methods that programmers need, so the extra code is needed to do even common tasks. Another cause may be that the API designers assume users will need the flexibility to put things together in multiple ways, but most users do not, so everyone uses the same collection of methods in the same way [19]. Users may also use boilerplate code even though there are already implemented API methods that can succinctly perform the task, which indicates discoverability problems [19].

However, despite general consensus on the undesirability of having to write boilerplate code, as well as API design guidelines explicitly mentioning boilerplate as an anti-pattern [1]–[3], the concept remains largely undefined and understudied. We start by reviewing boilerplate code examples and definitions from multiple sources (Section III). Through qualitative analysis, we confirm that boilerplate involves sections of code that have to be written repetitively to accomplish common and otherwise simple tasks that users largely do not want to think about. Moreover, we find that the main reasons for boilerplate code are underlying language and API limitations. We also find that developers and API contributors make efforts to reduce the amount of boilerplate code by introducing new helper functions and abstractions.

Next, we present MARBLE (Mining API Repositories for Boilerplate Lessening Effort), an automated technique for identifying instances of boilerplate API client code. Since a key property of boilerplate is that it is repetitive, we designed MARBLE on top of an existing API usage pattern mining approach, specifically PAM [13], which is automated and can be run at scale. However, not all idiomatic API usage patterns that an approach like PAM extracts, of which there are typically many, should be considered boilerplate. Therefore, we developed novel filters using AST comparison and graph partitioning (Section IV) to identify, among the frequent API usage patterns, those which are most likely to involve boilerplate. By reducing the number of false positives, API designers could then focus manual review on the most likely candidates. The source code of MARBLE is available online [20].

We evaluated MARBLE on 13 Java APIs, for which we mined around 10,000 client code files from GITHUB open-source projects, with 768 client code files per library on average after random sampling. Our results (Section V) show that not only does MARBLE return a sufficiently short list of boilerplate candidates for manual review to be feasible, but also that more than half of these candidates are considered boilerplate by two experienced Java programmers (the first and third authors of this paper), where the third author is an API designer at a large software company. To further the discussion about what boilerplate is and how it impacts APIs, we discuss some of the boilerplate candidates and suggest potential API design improvements. The full list of boilerplate candidates mined is available online [20].

Note that we are *not* claiming that all boilerplate code is bad, or that boilerplate code should always be eliminated. In fact, some of the patterns we identified as boilerplate are important to leave as-is to achieve other code quality requirements, such as increased readability or separation of concerns. However, as has been proposed elsewhere [8], we argue that these kinds of API design decisions are best made with full knowledge of the tradeoffs. We argue that MARBLE provides data which may be used in practice by API designers as a basis for such discussions. We also recognize that our method, like any other data-mining approach, is only applicable after an API has sufficient client code using it, and is therefore complementary to lab studies and API design reviews.

In summary, we contribute: i) the boilerplate API code mining problem; ii) properties which can be used to identify boilerplate code; iii) an automatic boilerplate code mining algorithm; iv) an empirical evaluation on 13 Java libraries.

II. Related Work

To our knowledge, our work is the first to automatically mine potential boilerplate code instances from software repositories. We contribute to the literature on API usability, which serves as the motivation for our work, and API usage pattern mining. Our work is also related to code clone detection, which can be helpful for identifying some API usability problems, but has limitations for boilerplate code mining.

A. API Usability Studies

Previous research and industrial work have sought to aid API designers in creating more intuitive, learnable, and usable APIs (see, *e.g.*, [2], [8] for surveys). Researchers have investigated ways of evaluating API usability in lab-based studies, including diagramming the concepts used in the API, and combining interviewing and observational studies during a programming task [5], [21], [22]. While these methods have been successful in the lab, it remains unclear how they might scale for evaluating larger APIs.

Peer reviews are also often used to evaluate APIs [6], [7]. For example, Macvean *et al.* [6] developed a peer review process at Google called "Apiness" that assigns two reviewers to asynchronously provide feedback on the API's design. While this approach has proven to be effective at Google, it requires a level of expertise for both the API reviewer and API designer, which may not be available. Moreover, such approaches are mainly designed to be used before the API is released, therefore, they are not ideal to identify potential usability issues of the API in client code at scale, after an API has been released.

There are also many publications that discuss general API design processes and guidelines [1], [3], [7], [19], [23], [24], but these still require expertise in API design, and the methods introduced in those articles may be hard to operationalize [1], [19], [23]. Heuristics are also used to evaluate APIs after their creation [25], [26], sometimes automatically [27], [28].

B. Mining API Usage Patterns

Significant previous research has addressed mining of API usage patterns, which has proven to be useful for detecting both correct and incorrect API uses [10]. API misuse detectors [29]–[32] automatically mine incorrect uses of an API which deviate from normal patterns. However, these algorithms focus on detecting potential anomalies or bugs in the programs, mainly mistakes made by developers, rather than focusing on the design of the API itself.

There are also algorithms for automatically mining API usage patterns which identify API methods that are frequently called together [11], [13], [33], [34]. For example, PAM [13] uses a probabilistic model and an expectation maximization algorithm to mine sequences of API calls that are not only frequent, but also occur more often together in a sequence than would be expected by chance. ExampleCheck [33] uses the closed frequent sequence mining algorithm BIDE [35] combined with an SMT solver, to mine API usage patterns as well as guard conditions that protect an API call. However, while these algorithms can help users learn APIs more effectively, they are not very useful for API designers, in that it is hard to filter out which instances are directly related to API usability issues given the long list of API usage patterns they return.

Particularly noteworthy is Google's StopMotion tool [36], which focuses on identifying API usability issues in a scalable fashion by analyzing the developers' work history. StopMotion analyzes editor logs from developers, focusing specifically on instances where code edits relate to API method usage. Although this approach is able to identify instances of usability issues from client code, it currently only considers changes made for a single method call. It is therefore not ideal for capturing usability issues involving multiple API calls in a block of code, or identifying code which surrounds API method calls, namely boilerplate code.

C. *Code Clone Detectors*

Code clones have long been regarded as harmful to a system's design, so researchers have studied them extensively, including their causes [37] and detection algorithms [38], [39]. Kapser and Godfrey [37] found that some code clones are caused by limitations of the programming language [37]. Kim *et al.* [40] further found that programming language limitations partially explain why many long lived clones are not easily refactored. This suggests that API design issues could be identifiable in client code, among the code clones.

There are also approaches to detect code clones from large code bases that could potentially be used to identify clones indicative of API design issues, given a large repository of client code. Deckard [41] is a tree-based code clone detection algorithm that measures the similarity of subtrees in syntactic representations. Deckard clusters "characteristic vectors" which are transformed from parse trees of the source code, to detect code clones based on this clustering. More recently, White *et al.* [42] devised a learning-based code clone detection technique that uses deep learning, by training an autoencoder to encode the lexical level information and syntactic level information into vector representations, and match them to detect the code clones. SourcererCC [39], on the other hand, uses a bag-of-tokens technique to detect Type-3 clones. It exploits an optimized inverted-index and filtering heuristics to quickly query the clone candidates with less computation for more scalable clone detection within large codebases. Both of these detectors successfully find code clones and scale well. However, these techniques are designed to detect all kinds of clones, not just API-related or boilerplate-related ones. Therefore, false positive rates for boilerplate candidates would likely be high and additional filtering would be needed to identify the clones indicative of API design issues. We choose to base our boilerplate approach on automatic API usage pattern mining techniques rather than code clone detection tools, since the former are specifically designed for APIs, which we expect would require less filtering of false positive boilerplate candidates.

III. STUDYING BOILERPLATE CODE

As far as we have been able to find, studies of boilerplate code, or studies that even mention boilerplate code, are scarce (exceptions include [3], [9], [37], [43]). Mostly we have found it to be "I-know-it-when-I-see-it," with the existing explanations being vague and abstract, rather than deterministic.

At the same time, although boilerplate code is regarded as something that programmers want to avoid [14]–[16], and API design guidelines suggest that API designers should reduce the need for boilerplate code [1]–[3], we still have not seen any studies of whether some boilerplate code is induced by APIs, and if so, whether it is possible to reduce it at the API level.

Thus, to help understand the characteristics of boilerplate code, and the impact of API design on the need for boilerplate in client code, we first investigated three research questions:

- RQ_1: What is a good definition and what are common properties of boilerplate code?
- RQ_2: What are reasons for needing boilerplate code?
- RQ_3: How do API users and API authors deal with boilerplate code?

A. *Resources*

We reviewed the literature, surveyed our social media contacts, and reviewed Stack Overflow questions and GITHUB commits. Mainly, we looked for boilerplate code examples, but when available, we also collected the rationale behind the boilerplate designation, reasons for the boilerplate, and how programmers dealt with boilerplate. We looked for Java boilerplate code examples involving at least one API call. We chose Java because the API usage pattern mining technique we build on (Section IV) was tested for Java. In some communities (*e.g.*, web developers), boilerplate code is used as a synonym for template code [44], but we exclude this context as we are looking for boilerplate related to API usability.

1) Literature: We searched for definitions or explanations of boilerplate code in Google Scholar [3], [9], [37], blog posts, online discussion boards (*e.g.*, reddit) and Wikipedia. When available, we also collected boilerplate code examples.

2) Survey: We asked our Twitter contacts to share boilerplate code examples and the reasons behind the boilerplate designation. Overall, 8 participants submitted 1 to 3 boilerplate examples each and all provided the reason why they thought each example qualifies as boilerplate.

3) Stack Overflow: The first author identified five popular Java API tags (`android`, `swing`, `jdbc`, `spring-mvc`, `jsp`) in Stack Overflow and manually collected questions asking about how to reduce boilerplate code, using Stack Overflow search queries (*e.g.*, "[Swing] boilerplate"). We checked the first page (15 questions) of the results for each Java API tag, and collected boilerplate code examples and the reasons why the questioner thought it was boilerplate.

4) GitHub Commits: We identified and cloned the top 10,000 most starred Java repositories from GITHUB, using the March 2018 version of GHTorrent (details in Section V). Then, we identified all commits including the keyword "boilerplate" in the commit message. Finally, the first and second authors manually coded all the matching commits.

B. RQ_1: What is a good definition and what are common properties of boilerplate code?

We investigated the available definitions of boilerplate code from the literature, and iteratively discussed the boilerplate examples among the research team (which includes an experienced API designer in a large software company, who is often involved in large software projects using APIs), distilling common properties. We did not use GITHUB commit data in this analysis because (1) it does not explicitly express the characteristics of boilerplate, and (2) it does not indicate the exact location of boilerplate code in many of the code changes.

1) Undesirable: Commonly, boilerplate code is identified using subjective properties, sometimes explicitly: "It's a subjective definition" [17]. Mostly, such properties have *negative connotations*. One source calls it "uninteresting, unchanging, repetitive, and/or tedious" [45]. Another common but subjective property is that boilerplate code is needed even for simple functionality. The highest voted answer from Stack Overflow defines it as "it ought to be much simpler" [17]. We summarize all of these properties as being "undesirable."

2) High frequency: Most of the definitions and explanations require that boilerplate code occurs frequently in client code, such as "shows up again and again" [17], or "code that has to be included in many places" [18]. Frequency is a particularly intuitive property given the negative connotation of boilerplate code: indeed, if it were rare, its impact would likely be reduced. The high frequency property also implies that boilerplate API code examples should be found among idiomatic API code examples, as the latter are by definition frequent, hence our choice to base our approach on an existing API usage pattern mining tool [13].

3) Localized: The statements constituting boilerplate code are usually *closely located near each other*, rather than spread over multiple methods or files. All examples from Stack Overflow and the survey, and three examples from Google Scholar [3], [9], [37] were parts of a single method. The Wikipedia example of getters and setters within a class [18] is the only one not limited to a single method.

4) Little structural variation: The boilerplate code instances appear in *similar form without significant variation*. Many sources describe that it "gets copied and pasted" [3], and is used "with little or no alteration" [18]. We also found that many explanations of boilerplate code describe the examples as "I find myself writing the same ugly boilerplate code" [46], or "a lot of code that must be duplicated" [47].

In fact, this corresponds to the definition of code clones, especially "templating clones" [37]. However, while code clones need not occur with high frequency to be considered clones, boilerplate should occur frequently (Property 2).

C. RQ_2: What are reasons for needing boilerplate code?

1) Method: The first and second authors performed closed coding for all of the boilerplate related commits we collected from GITHUB. As one property of boilerplate code (RQ_1 Property 4) corresponds to a subcategory of code clones, we borrowed Roy and Cordy's "reasons for cloning" [48] as our starting set of codes: development strategy (reuse approach, programming approach), maintenance benefits (avoiding, ensuring, reflecting), overcoming underlying limitations (language limitations, programmers' limitations), and cloning by accident (protocols to interact with APIs and libraries, programmers' working style). As there were commits referring to different types of boilerplate (*e.g.*, boilerplate license), we also coded the commits with types of boilerplate: boilerplate, client (*i.e.*, reduce the boilerplate code using the API), comment (*i.e.*, boilerplate in the comments such as license, javadoc), and Non-Java. Each commit was assigned one type and one reason based on its commit message and code diffs. The first and second authors started coding collaboratively and, after 10 agreements, each separately coded half of the data. In total, we randomly sampled and coded 120 commits, and the two coders reached 87.5% agreement (Jaccard Index) both for boilerplate types and for reasons, on 20% of the data.

2) Results: Among 120 commits, 40 of them were commits to reduce the use of boilerplate code. We found that the predominant reason for needing boilerplate was overcoming underlying language limitations (mentioned in 19 commits). Examples of this include needing to initialize many getters/setters and verbose error handling in Java. 11 were induced in order to interact with APIs, for example, tagged as "Protocols to interact with APIs". Some of the boilerplate code was due to questionable API designs (*e.g.*, requiring the client to cast the output by providing an abstract object), but some seemed inevitable due to the design patterns or apparent trade-offs in the design of the APIs. For example, an API adopting a builder pattern usually involves a lot of boilerplate to set properties of an object. Another 10 were due to programmers using the API inefficiently, such as using an API call which is not ideal that requires more code.

Since most of boilerplate code instances are by-products of language and API limitations, analyzing boilerplate code can help review their designs and find usability issues. Despite some of the limitations being unavoidable, such as error handling in Java, there are many other situations where API designers can reduce the need for boilerplate, such as by adopting annotation libraries or introducing helper functions. Boilerplate code due to programmers using the API inefficiently may be a signal that there are discoverability issues, so the documentation and tutorials might need to be improved to overcome the conceptional gap between the API designers and API users.

Note that while we were able to code every boilerplate instance with codes from "reasons for code cloning" [48], which indicates that boilerplate can be considered a type of code clone, the two are not identical, as clearly not all code clones can be considered boilerplate under our definition (high frequency, localized, API related). Therefore, while our approach to automatically mine boilerplate candidates (Section IV) starts from an existing API usage pattern miner, future work could also consider boilerplate mining approaches that start from code clone detectors, but exploring this goes beyond the scope of the current work.

D. RQ_3: How do programmers and API contributors deal with boilerplate code?

1) Method: To answer RQ_3, the first and second authors performed descriptive coding on the changes that were made to reduce the amount of boilerplate code – either boilerplate within the source code itself, or boilerplate that is needed by the client to use the library. We used the same 40 boilerplate instances found in commits from RQ_2 (*i.e.*, attempts to reduce the boilerplate code by editing the project code), and 6 commits that changed the API itself (*i.e.*, attempts to reduce the boilerplate code using the API). We coded based on the code diffs and commit messages, and extracted the means used to reduce boilerplate code.

2) Results: The majority of boilerplate code reductions within a project were made by introducing new helper functions or classes, either by writing new ones, or by including a function or class from an external API. For the simple Java-specific boilerplate such as getters/setters, some used annotations (*e.g.*, Project Lombok [49]) or injection to reduce the amount of boilerplate. When changing the API to reduce the client-side boilerplate, programmers added more processing into the library, thereby reducing the need for pre/post processing for the input/output of API calls. Some made the interfaces more specific to reduce the need for parsing or casting in the client code. Also, like within-project boilerplate reduction, some commits added a set of helper functions or new classes to allow users to have a more specific but simpler interface, which can usually be done without making breaking changes to the API.

IV. Mining Boilerplate Code

Using the results from the previous section (Section III-B), we seek to find code instances that contain calls to a target API and satisfy the properties of boilerplate code we identified: (1) are undesirable, (2) occur frequently in client code, (3) occur within a relatively condensed area, and (4) are used in similar forms without significant variations.

To this end, we designed MARBLE, which combines an API usage pattern mining technique with a graph partitioning algorithm to identify candidate boilerplate code from software repositories. MARBLE consists of several steps, depicted in Figure 1 and described below. In summary, we first identify a large set of API usage patterns which represents our initial set of boilerplate candidates. We then filter out any patterns that are spread over multiple methods, or which have many variants, to finally provide a short list of boilerplate candidates that satisfy all of the properties above, except for Property 1 (undesirable). These candidates could then sorted and contextualized with the real-world client code, and delivered to the API designers so they can review the candidates for Property 1. We intentionally designed the process in this order because testing Property 4 (little structural variation) is computationally expensive. By filtering out the candidates that do not satisfy the other properties, we are able to reduce the number of AST comparisons (Section IV-C1).

A. API Usage Pattern Mining

We start from an existing API usage pattern mining technique, to collect boilerplate candidates containing one or multiple target API calls and satisfying the high frequency property. Specifically, we chose PAM (Probabilistic API Miner) [13], a state-of-the-art parameter-free probabilistic approach, which is fully automated and available open-source. In their evaluation, Fowkes *et al.* [13] found that PAM returns less redundant and less numerous results compared to other API usage pattern mining algorithms the authors compared.

1) The PAM Core: PAM uses a *probability model over API call sequences* to identify "interesting" sequences of API calls / API usage patterns. Given a target API, the model can be trained unsupervised on a corpus containing code from open-source GitHub repositories. Concretely, PAM first parses each source file and extracts the sequence of target API calls within each method (only Java code is currently supported), using a depth-first traversal of the abstract syntax tree (AST). At the same time, frequency information for each API call over methods is recorded. For example, given `javax.xml.transform` as a target API and Listing 1 as one client method using it, PAM's API call extractor returns

- `javax.xml.transform.TransformerFactory.newInstance`
- `javax.xml.transform.TransformerFactory .newTransformer`
- `javax.xml.transform.Transformer .setOutputProperty`
- `javax.xml.transform.dom.DOMSource.<init>`
- `javax.xml.transform.stream.StreamResult.<init>`
- `javax.xml.transform.Transformer.transform`

Then, PAM uses expectation-maximization (EM) [50] to iteratively infer "interesting" API usage patterns (*i.e.*, sequences of API calls) and learn the probability model. That is, an API call sequence `[A, B]` is "interesting" if the two API calls `A` and `B` occur together more often than expected by chance, given the individual frequencies of `A` and `B`. The EM algorithm iteratively interleaves API call sequences, and searches for the set of patterns that maximizes the probability that the model assigns to all client methods in the input dataset. For more details we refer to the original paper by Fowkes *et al.* [13].

PAM returns a ranked list of API usage patterns $P = [P_1, P_2, \ldots, P_n]$, where $P_i = [c_1, c_2, \ldots, c_m]$, is an

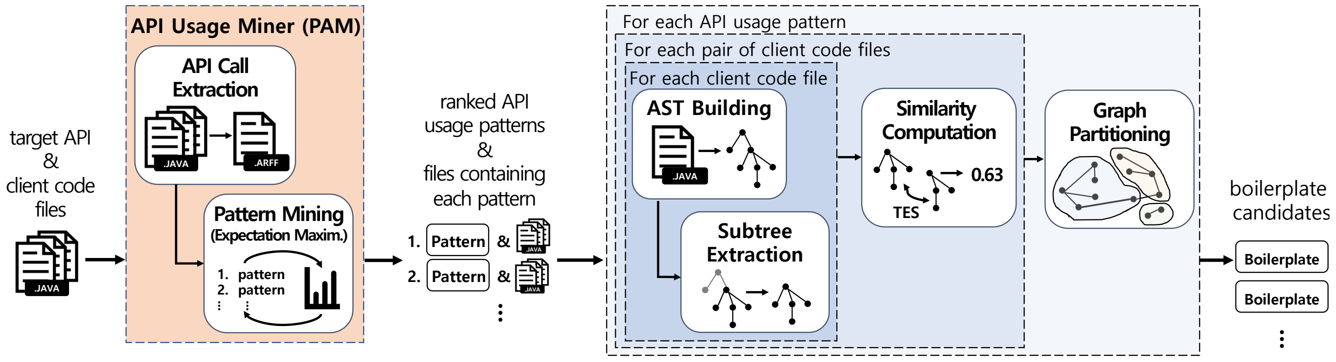

Fig. 1. Overview of our mining process and the steps involved.

API call sequence. For example, when we run PAM on `javax.xml.transform`, it returns:

- P_1 = `[javax.xml.transform.dom.DOMSource.<init>, javax.xml.transform.stream.StreamResult.<init>]`
- P_2 = `[javax.xml.transform.TransformerFactory. newTransformer, javax.xml.transform.Transformer.transform]`
- P_3 = `[javax.xml.transform.dom.DOMSource.<init>]`
- $P_4 = \ldots$

2) Modifications to the PAM Core: As our main goal is to help API designers identify the patterns that are likely to reflect API usability issues, the list of candidates to be considered must be relatively short, since such reasoning requires designers' manual effort. We modified the base PAM algorithm to reduce the number of false positive boilerplate patterns returned. This step involved setting two thresholds empirically, which we did after reviewing a sample of PAM results: First, if there is a pair of patterns such that one fully contains the other (*e.g.*, P_1 and P_3 above), we remove the sub-pattern (P_3) unless the number of occurrences is more than 50% different from its super-pattern's, to avoid reporting multiple small variations of one boilerplate candidate. For example, if there is a sequence `[a, b, c]` which occurred 100 times among the client code files and another sequence `[a, b]` occurred 120 times, we keep `[a, b, c]` and ignore `[a, b]` because `a`, `b`, and `c` are mostly used all together. However if `[a, b]` occurred 500 times, we do not ignore it because it is likely that there are other uses not involving `c`. Second, to avoid reporting rare and project-specific boilerplate code candidates, we also ignore patterns that occurred in less than 5% of client code methods for a given API.

3) Limitations: The returned API usage patterns are sequences of API calls, without any structural information. This ensures that the returned patterns are robust to variations in local context, *e.g.*, conditionals, loops, exception handling, *etc.*, which is desirable when the goal is mining generic API usage examples. However, this is at odds with our third boilerplate requirement that the call sequence should appear in similar form without significant variation.

Another limitation of PAM for boilerplate mining is that the order of API calls matters. When a boilerplate instance involves multiple API calls that can be used in any order, such as `getHeight()` and `getWidth()`, PAM would consider `[getHeight(), getWidth()]` and `[getWidth(), getHeight()]` to be different sequences, and the "interestingness" of this API usage would be lower than it should be.

Finally, PAM was originally designed to capture the usage patterns of a single library, whereas API usage patterns or boilerplate can have multiple libraries involved.

We address these limitations in the following steps, by also considering the context around the "interesting" API call sequences.

B. AST Extraction

To decide whether an "interesting" API usage pattern involves boilerplate, we should also consider the (structural) context around the API calls. For example, if other methods (*e.g.*, built-in language APIs) are always used around or between the target API calls, or if the sequence of target API calls is always used inside a certain loop construct, we should also consider this context as part of the candidate boilerplate instance. Therefore, to determine this context, given a list of API usage patterns and a list of client code files containing instances of those patterns $[(P_1, F_1), (P_2, F_2), \ldots, (P_n, F_n)]$, respectively, for each P_i we extract and post-process the ASTs of the files in F_i. Moreover, since we are only interested in code that occurs in local areas (Property 3 above), *i.e.*, the areas around the target API calls, we restrict this analysis to individual methods and split the file-level ASTs (which correspond to entire classes) into method-level subtrees.

Still, the method-level AST subtrees may contain nodes unrelated to target API calls and the candidate boilerplate pattern. To narrow down the relevant parts of the method-level AST subtrees S, we use a simple slicing heuristic: we extract the smallest sub-subtrees of each subtree S, which completely encompass the target API call pattern. For example, given an AST of the client code in Listing 1 (Figure 2) and an API usage pattern P = `[javax.xml.transform.dom.DOMSource.<init>, javax.xml.transform.stream.StreamResult.<init>]`, we extract the first common ancestor of the `DOMSource` and `StreamResult` nodes, *i.e.*, the subtree rooted at `Method Invocation`.

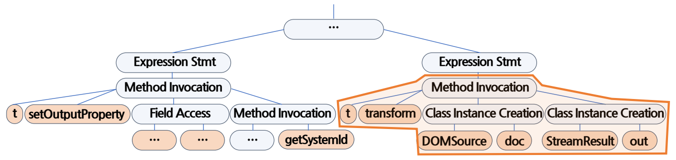

Fig. 2. A part of the AST for the code in Listing 1 and the extracted subtree (colored) for the API usage pattern `[DOMSource.<init>, StreamResult.<init>]` using our slicing heuristic.

For patterns with a single API call, the smallest subtree is the same as the API call, which means we do not acquire further contextual information. Therefore, we modify the smallest subtree heuristic for these patterns, and find the smallest `if`/`loop`/`try` subtree containing the API call. However, this heuristic may not always extract a smaller subtree than the entire method (*e.g.*, if no `if`/`loop`/`try` is used in the method). Based on the third property that statements constituting boilerplate code are closely located near each other, we applied another heuristic: when the subtree has over 20 method invocations, trim the sub-subtrees that are far from the sub-subtrees containing the API calls. We chose the threshold 20 informed by the examples collected from the qualitative study in Section III.

If an API call of a pattern occurred multiple times in a client file, there might be multiple potential subtrees. In this case, we use the smallest one, following Property 3: the further the calls are apart, the less probable it is that they form a single pattern. In the case that the full smallest pattern occurs multiple times in a client code file, we keep multiple subtrees.

C. Graph Partitioning

As the third step in our approach, we check Property 4: whether the API call sequence is used consistently in similar contexts (*i.e.*, structures) throughout the client code.

To capture this property, we devise an approach to 1) compute the similarity between all pairs of subtrees containing the API call sequence contexts; and 2) cluster together similar subtrees. Intuitively, if there are many clusters with low similarity, this indicates that there are many different ways a sequence of API calls is being used at the code level, suggesting that the pattern is less likely to be a part of boilerplate, as per Property 4. In contrast, if there is a cluster having a number of subtrees, and the similarity between them is high, the cluster (*i.e.*, specific use case) can be a boilerplate candidate.

1) Pairwise Similarity: Given a list of subtrees for each client file in F_i containing a same API pattern P, we compare every pair of subtree lists from $\langle f_i, f_j \rangle$ in F_i, and calculate the similarity between them. We use AP-TED (All Path Tree Edit Distance) [51] as our distance/inverse similarity measure, since it is memory efficient and fast. Other tree differencing algorithms such as GumTree [52] could be applied as well.

To calculate the AP-TED, we visit each subtree in pre-order, collecting the types of each node (*e.g.*, MethodDeclaration or IfStatement). To avoid noise from lexical details, such as variable names, we only collect nodes for: loops (*e.g.*, ForStatement), error handling (*e.g.*, TryStatement), conditions (*e.g.*, IfStatement), casts, and method invocation types. For the MethodInvocation nodes, we also collect the names (*e.g.*, `newInstance`) to compare different API calls used in the boilerplate candidate. To overcome PAM's limitation that it only considers the usage patterns of one target API, we also collect the names of method invocations that are not from the target API. By doing this, even though PAM is not able to capture external API calls as part of a sequence, our approach can still use them to calculate the similarity, and the external calls will be seen in the boilerplate candidate if they frequently occur together with the target library's API calls. It also helps mitigate PAM's other limitation—not capturing a set of API calls into a usage pattern unless they occur in the same order.

To calculate the similarity using the tree edit distance, we invented TES, the Tree Edit Similarity. When s is a list of subtrees in f, each of which encompass the target API, given two lists of subtrees $\langle s_1, s_2 \rangle$ for $\langle f_1, f_2 \rangle$, we define TES as:

$$TES(s_1, s_2) = max\left(\frac{1}{e^{AP\_TED(s_{1i}, s_{2j}) \cdot 0.1}}\right) \quad (1)$$

In the case that the client code file uses the pattern multiple times so there are multiple subtrees for s_i, we calculate the distance between every pair of subtrees, and use the maximum value for the next step.

2) Clustering: With these pairwise similarity values, we build a weighted graph for each pattern, in which nodes n_i are client code files, and edges $n_i \rightarrow n_j$ are weighted by $TES(n_i, n_j)$. We then cluster the nodes in this graph, to capture the different contexts (structures) in which an API call sequence is being used. As the similarity values between client code files are computed based on the ASTs, boilerplate candidates that are structurally similar would likely be clustered together. On the other hand, even if the same API calls are used in two client files, if their structures are significantly different, or external API calls around the pattern are significantly different (which the edge weight captures), they would likely be clustered separately. Therefore, after graph partitioning, the clusters would indicate structurally-different usage patterns given a sequence of target API calls.

For clustering we use the Louvain community detection algorithm [53], a heuristic method based on modularity optimization: Given a weighted graph of n nodes, Louvain first assigns a different cluster to each node. Then, for each node n_i, Louvain calculates the gain in modularity by removing n_i from its cluster and placing it into its neighboring n_j's cluster. If the gain is positive, and maximum among the gains from other neighbor nodes, the algorithm removes n_i's cluster, and merges n_i into n_j. The process repeats and is applied until there is no further improvement in modularity. Secondly, Louvain adjusts the weights of the edges. The weights of the edges between the new clusters are the sum of the weights of the edges between nodes in the corresponding two clusters. The algorithm keeps iterating the first phase, merging clusters, and second phase, adjusting weights, until a fixed point.

We applied this technique for three main reasons: (i) unlike most other graph clustering algorithms for which the number of clusters should be given as input, Louvain determines it as part of the algorithm; (ii) its computation time is short; (iii) it was originally designed for large networks (*e.g.*, 118 million nodes [53]), hence we expect it to scale up well.

3) Additional Filtering: The clustering algorithm does not guarantee that within a cluster the client code instances of the API usage pattern are *all* highly similar among each other, *i.e.*, that they all represent the same boilerplate candidate instance. To further prune spurious clusters which may increase noise in the results, we require that the average pairwise *within cluster similarity* is greater than a threshold. Empirically, we observed that patterns involving many API calls would have more variance in the subtrees and thus the similarity would be lower than short usage patterns, even though qualitatively they would appear similar; therefore, when the pattern is longer, the similarity threshold should be lower. We set this threshold for average TES (Equation 1) to be $1/e^{2(x+1)\cdot 0.1}$, where x is the number of API calls in the sequence. For example, when a usage pattern contains only one API call, the threshold is $1/e^{(2\cdot 1+2)\cdot 0.1} = 0.67$, *i.e.*, we discard clusters with average within-cluster similarity below 0.67.

D. Viewer

Since boilerplate code has the subjective properties discussed in Section III that cannot be automatically tested, manual review of the candidates is necessary. To help API designers efficiently review them, we implemented a viewer for the boilerplate candidates. Based on the intuition that more verbose boilerplate candidates should be reviewed first by API designers, MARBLE's viewer ranks all the boilerplate candidates by their length, and for each one, displays usage examples from three representative client code files from different GitHub projects.

V. Evaluation

In this section, we evaluate the accuracy and potential usefulness of our mining algorithm, by answering:

- RQ$_4$ (Validation): How well does MARBLE identify known boilerplate examples?
- RQ$_5$ (Precision): How many of the boilerplate candidates found by MARBLE would human experts agree with?
- RQ$_6$ (Practicality): Does MARBLE return a reasonably short list of boilerplate candidates for manual review?
- RQ$_7$ (Usefulness): Does MARBLE identify informative boilerplate candidates that could help review an API?

RQ$_4$ is to test whether MARBLE finds the 13 boilerplate examples corresponding to 13 Java APIs we collected from the literature, survey, and Stack Overflow in Section III.

RQ$_5$ evaluates MARBLE's false positive rate. Quality assurance tools, such as defect prediction [54] or static analysis [55], should generate few false warnings to be usable in practice.

RQ$_6$ evaluates MARBLE's practicality. Reviewing boilerplate candidates and investigating potential usability issues require manual effort from the API designers, as labelling something as boilerplate is ultimately a judgement call. On the same set of 13 APIs for which we collected 13 known boilerplate examples, we evaluate whether MARBLE returns a sufficiently short list of candidates in the right order, so that it may be usable in practice. Specifically, we test to what extent our filtering steps involving AST comparison and graph partitioning (discussed above) will help to substantially reduce and rank the list of candidates for manual review compared to the baseline PAM [13].

RQ$_7$ is to qualitatively evaluate MARBLE's usefulness. The first and third authors manually reviewed all the mined boilerplate candidates for the same 13 APIs, and analyzed to what extent the candidates signal places where the APIs might be improved.

A. Experimental Setup

1) Dataset: To collect API client code, we identified and cloned the top 10,000 most starred Java repositories from GitHub, using the March 2018 version of GHTorrent [56], excluding forks and repositories marked as deleted. We then mined the Java source files importing the APIs, by matching import statements (*e.g.*, `import javax.xml.transform`). For APIs with more than 800 client code files, to reduce the runtime of our experiments, we sampled files randomly, ensuring 95% confidence level and 3% margin of error. Table I gives an overview of our dataset.

Each row in the table is a separate API on which we ran MARBLE, one API per each of the 13 known boilerplate examples. The "API Patterns" column shows the number of API usage patterns returned by just running PAM, and the "Boilerplate Candidates" column shows the number of boilerplate candidates that MARBLE identifies. The "Found Known Boilerplate" column shows whether MARBLE found the known examples. The "Precision" column shows the percentage of boilerplate candidates that the first and third authors labeled as actual boilerplate among the number of the candidates that our approach retrieved for each API. The "Client Files" column shows the number of client code files that were used for boilerplate mining. The "Files w/ BP" column shows the number of client code files that involve at least one boilerplate instance. The "Avg. Len." column shows the average length of the boilerplate candidates. The precision on the "Total" row is the micro-average precision; that is, the average precision after aggregating the data of all libraries.

2) Implementation Details: For the *API usage pattern mining* part (Section IV-A), we adjusted the output generation part of PAM's public implementation [57], without modifying the core algorithm. We ran PAM using default parameter settings: 10,000 iterations with a priority queue size limit of 100,000 candidates. For the *AST comparison* (Section IV-B), we wrote a Java program to parse and traverse ASTs, and extract subtrees with our heuristics. To compare the subtrees, we used the `apted` library [58], but customized the cost model to weigh insertion, deletion, and rename operation equally.

TABLE I
SUMMARY STATISTICS ON THE NUMBER OF CANDIDATE BOILERPLATE INSTANCES FOR THE APIS IN OUR DATASET.

API	API Patterns	Boilerplate Candidates	Found Known Boilerplate	Precision	Client Files	Client w/ BP	Avg. Len.
`android.app.ProgressDialog`	134	12	True	0.92	641	296	5.96
`android.database.sqlite`	508	7	True	0.57	796	96	7.49
`android.support.v4.app.ActivityCompat`	26	5	True	0.60	486	93	8.01
`android.view.View`	940	11	True	0.36	1,051	100	9.16
`com.squareup.picasso`	79	0	False	-	565	-	-
`java.beans.PropertyChangeSupport`	32	8	True	0.38	604	48	6.04
`java.beans.PropertyChangeEvent`	32	5	True[1]	0.00	749	-	-
`java.io.BufferedReader`	39	3	True	0.67	998	343	3.52
`java.sql.DriverManager`	30	0	False	-	744	-	-
`javax.swing.JFrame`	185	0	False	-	791	-	-
`javax.swing.SwingUtilities`	71	2	False	0.50	800	14	6.64
`javax.xml.parsers`	196	3	True	1.00	893	39	11.23
`javax.xml.transfrom`	325	3	True	0.67	871	45	9.2
Total	2,597	59	0.69	0.56	9,989	1,074	6.06

[1] During the evaluation, authors came to the conclusion that the known boilerplate instance for `java.beans.PropertyChangeEvent` from Stack Overflow is not a strong boilerplate code. Therefore, even though MARBLE mined the similar pattern from the client code (Found Known Boilerplate: True), none of the candidates from this library is labeled as boilerplate code (Precision: 0.00).

For the *graph partitioning* (Section IV-C), we wrote a Python program to build a graph, preform graph partitioning using the NetworkX package [59], and filter spurious clusters using our heuristics. The final boilerplate candidate viewer generator for the API designers is implement in Python, and generates a html page for each API.

B. Results and Discussion

1) RQ_4 (Validation): How well does MARBLE identify known boilerplate examples?: We ran MARBLE on all 13 APIs represented in the discovered boilerplate examples (Table I), and manually compared the returned candidates to the known examples. Among the 13 known boilerplate examples, MARBLE could identify 9 (69%).

Three out of four false negatives were not caught by MARBLE because they incorporate a variety of real-code (*i.e.*, non-boilerplate code) inside them, like `invokeLater` in `javax.SwingUtilities`. Although the boilerplate wrapping the real-code was repetitive and the same for every usage, the non-boilerplate part varied widely among the client code files, which lowered the similarity between the client code AST subtress containing this pattern. This could be improved in the future by applying program analysis to more accurately slice the API-call-related and unrelated parts of the code (we discuss more in Section VI). eliminating irrelevant non-boilerplate code by applying program analysis, and we discuss more in Section VI.

Another false negative was a builder pattern, which MARBLE did not identify because client code files used different combinations of setter calls. A Stack Overflow user [60] complained about this because the same builder was needed multiple times within the project, which does not necessarily mean that other programmers use it in the same way in other projects. Since MARBLE's goal is more general, to inform API designers about potential boilerplate in a wide range of client code, this example was not exactly in scope. However, MARBLE could be extended to also identify these within-project boilerplate examples if API designers feel the need, by adding the within-project pattern frequency to the algorithm.

We conclude that MARBLE is valid.

2) RQ_5 (Precision): How many of the boilerplate candidates found by MARBLE would human experts agree with?: MARBLE returned 59 boilerplate candidates overall for the 13 APIs in our sample (Table I). To compute MARBLE's precision, the first and third authors labelled each candidate as potentially boilerplate or not. The authors first separately labelled all of the candidates (77% inter-rater agreement), then discussed disagreements until reaching consensus, finally updating the labels. The main criteria for the boilerplate designation were whether it potentially reduces the API usability, and whether it could be further abstracted.

As a limitation, note that we labelled some candidates as false positives even though they resemble boilerplate (verbose and seemingly abstractable), because we could not confirm that these could significantly lower the API usability without looking at the number of occurrences within a single file or a single project, which goes beyond the scope of this work. As discussed above, we designed MARBLE to identify boilerplate candidates across a large number of client code files / projects to help API designers focus on boilerplate that might affect more users. However, while we were reviewing the candidates, we found that within-project boilerplate might also impact API usability. For example, as a programmer, it could be annoying if three lines of an API sequence need to be duplicated across many methods within a project, even though those three lines might be rarely used in other projects.

Overall, MARBLE's precision is 56%: the two annotators agreed that 33 out of 59 candidates could be considered boilerplate. We conclude that MARBLE has acceptable precision.

3) RQ_6 (Practicality): Does MARBLE return a reasonably short list of boilerplate candidates for manual review?:

```
if (ActivityCompat.shouldShowRequestPermissionRationale(this,
        Manifest.permission.READ_EXTERNAL_STORAGE)) {
} else {
    ActivityCompat.requestPermissions(this,
            new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
            MY_PERMISSIONS_REQUEST_READ_EXTERNAL_STORAGE);
}
```

Listing 2. Boilerplate code instance of Android ActivityCompat client codes.

Comparing the MARBLE results to PAM's, the API sequence miner our approach is built on (Section IV-A), we observe that MARBLE significantly reduces the number of resulting instances by applying further filtering on the PAM output using AST comparison and graph partitioning. The reduction is from a mean of 200 usage patterns per API with PAM down to a mean of 4.5 boilerplate candidates with MARBLE (median down from 79 to 3); the largest reduction is for `android.view.View`), from 940 down to 11 (Table I).

We also measured the time it takes to label the boilerplate candidates, to roughly estimate the time needed for a designer's manual review. The third author, who is an expert API designer, took less than 3 minutes per boilerplate candidate, and the first author, a software engineering Ph.D. student, took around 5 minutes per candidate. Since we did not have enough experience with some of the APIs, it took more time to read the documentations and use cases. However, for API designers, we believe that it would take less time to review the boilerplate candidates, and find potential usability issues.

We conclude that MARBLE returns a sufficiently short list of candidates for manual review to be feasible.

4) RQ$_7$ (Usefulness): Does MARBLE identify informative boilerplate candidates that could help a designer review an API?: We manually reviewed all 59 boilerplate candidates identified (full list available online [20]), looking for causes and potential improvements.

Given the space constraints, we discuss only three boilerplate candidates returned by MARBLE.

Android ActivityCompat. Listing 2 shows potential boilerplate involving Android's ActivityCompat, with two API calls: `ActivityCompat.shouldShowRequestPermissionRationale` and `ActivityCompat.requestPermissions`.

This boilerplate is to ask a certain permission to a user, but also provide an explanation if a user has already denied the permission request from this app. `shouldShowRequestPermissionRationale` returns true if the user has previously denied the request, but did not select the "Don't ask again" option in the permission request dialog; or false if the app has never asked a permission, a device policy prohibits it, or the user has selected the option.

MARBLE identified that out of 486 files importing ActivityCompat, this pattern is used in the same format in 36.

One potential redesign for better usability in this case is to abstract this into the API by adding a simpler method which handles permission checking and request rationale internally: if the permission is not granted, it checks if the permission request has been already denied or not, requests the permission with or without explanation, and sends the results to the client.

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int currentVersion) {
    Log.w(TAG, "Upgrading␣test␣database␣from␣version␣" +
            oldVersion + "␣to␣" + currentVersion +
            ",␣which␣will␣destroy␣all␣old␣data");
    db.execSQL("DROP␣TABLE␣IF␣EXISTS␣data");
    onCreate(db);
}
```

Listing 3. Boilerplate code instance of Android Database SQLite client codes.

However, we hypothesize that there could be a design rationale behind the current design, *e.g.*, possibly to improve the privacy of the users of Android applications. Although our proposed abstraction could help new Android developers get started with the API, lead to less code, and be less error prone in these common cases, this could also give an impression that providing rationale on permission requests to the Android application users is not critical.

We argue that this trade off is only valid when users understand the API designer's rationale, or at least that the rationale actually plays out as expected. If most users just copy and paste this boilerplate code without much thought, this design decision could reduce API usability without any benefits. Therefore, alerting API designers to situations where their design decisions result in boilerplate may help them review to what extent their design rationale is valid.

Android Database SQLite is an open-source relational database library in Android. Listing 3 shows a boilerplate code instance to upgrade a database by dropping tables and creating a new one, which requires using `SQLiteDatabase.execSQL` and `SQLiteOpenHelper.onCreate`.

Although `OnUpgrade` was intended to provide flexibility for users, MARBLE found that 40 client code files out of 68 overrode it in the same way, similar to Listing 3, by dropping tables using `execSQL` and creating new ones with `onCreate`. To mitigate this boilerplate, as discussed in Section III, API designers could make the common usage, such as logging the update, dropping the table, and recreating the database with a new version, as the default functionality of `onUpgrade`. This would allow users to write less code in general, and also give them some flexibility if needed.

Another way to reduce this type of boilerplate is to use annotation libraries (*e.g.*, the Spring framework [61]), or ORM – Object Relational Mapping – libraries), which offer an object-oriented interface to the relational database. Annotations and ORM tools reduce the need for simple CRUD (Create, Read, Update, and Delete) boilerplate, and many libraries have adopted them (*e.g.*, Neo4j-OGM). In fact, while reviewing the client code using this boilerplate, we observed that 10 of the client code files have adopted GreenDAO [62], which is an ORM tool for Android. The fact that many clients adopt a certain helper function or a tool can be a signal for API designers to update their API similarly, or recommend these tools to their clients for better usability. This boilerplate also shows that seeing the common patterns of use, and whether they could be abstracted by an annotation framework, might be

```
if (videoControlsView != null) {
    this.seekBar = (SeekBar) this.videoControlsView.findViewById(R.id.vcv_seekbar);
    this.imgfullscreen = (ImageButton) this.videoControlsView.findViewById(R.id.
        ↪ vcv_img_fullscreen);
    this.imgplay = (ImageButton) this.videoControlsView.findViewById(R.id.
        ↪ vcv_img_play);
    this.textTotal = (TextView) this.videoControlsView.findViewById(R.id.
        ↪ vcv_txt_total);
    this.textElapsed = (TextView) this.videoControlsView.findViewById(R.id.
        ↪ vcv_txt_elapsed);
}
```

Listing 4. Boilerplate code instance of Android View client codes.

useful for the helper library designers as well, in understanding the needs of users, and developing a new library.

Android View. Notably, as the APIs we used for evaluation are popular and actively maintained, in some cases we could actually find the improvements that had already been made by the API designers. The boilerplate candidate was still detectable in our dataset because the clients had yet to upgrade to the newer version of the API.

Listing 4 shows a classic boilerplate code for Android View. These 5 lines of code are to find the views from the XML layout resource file with the given IDs. The pattern consists of multiple uses of one API call: `android.view.View.findViewById`. We could observe that 217 files out of 518 files use this method at least three times in a row. It is already verbose since developers need to call this method multiple times, but even worse because null checking and typecasting are also needed.

A straightforward way to reduce this verboseness is to make the return type of `findViewById` to a generic `T`, to eliminate the need for manual typecasting. In fact, Android changed the method's definition from `View findViewById(int id)` to `T findViewById (int id)` starting with Android 8.0 [63]. This shows that 1) API designers care about the boilerplate code instances which reduce the API usability, and 2) informing API designers about the boilerplate candidates can actually lead to usability improvements.

Like the previous boilerplate candidate, another way to reduce this type of boilerplate is to use annotation libraries. There are several libraries providing annotation supports for this boilerplate (*e.g.*, `@BindView` of ButterKnife [64]), by helping users easily map the view ID declared in an XML layout file with the Java variable.

C. Threats to Validity

Our approach may be biased by the small number of APIs we tested it on. However, the boilerplate examples cover various domains and design patterns, and we believe that the properties we identified will generalize.

Note also that we only used 13 externally-known boilerplate examples to extract boilerplate properties and as a validation set to empirically choose the different thresholds involved. Still, MARBLE was able to discover many previously unreported boilerplate examples, which reduces the threat of overfitting.

As we only evaluated our algorithm with popular Java APIs which have hundreds to tens of thousands of client files, it is possible that the usefulness or performance of our algorithm varies for other libraries, that are relatively new or less popular. Also, as we analyzed the boilerplate instances with a single API designer and a Ph.D. student, others may disagree that our tool identifies boilerplate that is worth looking at.

VI. Conclusions and Future Work

We presented the novel problem of mining software repositories to identify candidate boilerplate code, as a potential API usability issue. We devised MARBLE, a new boilerplate mining algorithm based on four properties of boilerplate code that we identified from many sources (undesirable, high frequency, locality, and limited structural variation). We evaluated our algorithm on 13 Java APIs, finding many API usage patterns indicative of potential improvements to the API designs.

Our study opens up several directions for future research. First, we expect that integrating program analysis techniques, especially program slicing [65], into MARBLE will be helpful. One could run control and data flow analyses to identify and extract the statements that are dependent on the target API calls, or provide parameters and variables used by them, which could improve the precision of our mining algorithm. Second, the definition and properties of boilerplate could also be refined. Future work could survey API designers and developers to get wider input on what properties of boilerplate code they are most interested in having a tool capture, and adjust the algorithm accordingly. Future work could also extend the tool to support other languages, especially Javascript. We expect that Javascript APIs might have more boilerplate code instances, since the language has a huge ecosystem with a variety of APIs, which leads to frequent interactions among them, and also to frequent version changing. Finally, a more extensive evaluation could involve deploying the tool for use by many real designers in industry, in hopes of actually helping them identify and eliminate some boilerplate code from clients of their APIs, and review their design decisions. We plan such a large-scale empirical evaluation as part of our future work.

VII. Acknowledgement

We gratefully acknowledge support from Google Faculty Research Awards. We also thank Joshua Bloch for guidance about API design as well as our survey respondents for their time and input.

References

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

[2] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo, I. Fernández-Varela, and D. Álvarez-Estévez, "A systematic approach to API usability: Taxonomy-derived criteria and a case study," *Information and Software Technology*, vol. 97, pp. 46–63, 2018.

[3] M. Reddy, *API Design for C++*. Elsevier, 2011.

[4] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *International Symposium on Foundations of software engineering*. ACM, 2008, pp. 105–112.

[5] U. Farooq and D. Zirkler, "API peer reviews: A method for evaluating usability of application programming interfaces," in *Conference on Computer Supported Cooperative Work*. ACM, 2010, pp. 207–210.

[6] A. Macvean, M. Maly, and J. Daughtry, "API design reviews at scale," in *Extended Abstracts on Human Factors in Computing Systems*. ACM, 2016, pp. 849–858.

[7] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "API designers in the field: Design practices and challenges for creating usable APIs," in *Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2018, pp. 249–258.

[8] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[9] J. Bloch, "How to design a good API and why it matters," https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf, 2005.

[10] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[11] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[12] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing API usage examples at scale," in *Human Factors in Computing Systems*. ACM, 2018, pp. 580:1–580:12.

[13] J. Fowkes and C. Sutton, "Parameter-free probabilistic API mining across GitHub," in *International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265.

[14] "Collect GeometrySystem $\rightarrow$ drake_visualizer boilerplate by SeanCurtis-TRI pull request #8526 RobotLocomotion/drake," https://github.com/RobotLocomotion/drake/pull/8526.

[15] "Reduce boilerplate for subclasses issue #172 parse-community/Parse-SDK-Android," https://github.com/parse-community/Parse-SDK-Android/issues/172.

[16] "Can java help me avoid boilerplate code in equals()?" https://stackoverflow.com/questions/25183872/can-java-help-me-avoid-boilerplate-code-in-equals.

[17] "Boilerplate code definition of stackoverflow," https://stackoverflow.com/questions/3992199/what-is-boilerplate-code.

[18] "Boilerplate code definition of wikipedia," https://en.wikipedia.org/wiki/Boilerplate_code.

[19] J. Tulach, *Practical API design: Confessions of a Java framework architect*. Apress, 2008.

[20] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu, "Marble source code and the result," https://doi.org/10.5281/zenodo.3408715, 2019.

[21] J. Gerken, H.-C. Jetter, and H. Reiterer, "Using concept maps to evaluate the usability of APIs," in *Extended Abstracts on Human Factors in Computing Systems*. ACM, 2010, pp. 3937–3942.

[22] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of API usability," in *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2013, pp. 5–14.

[23] J. Bloch, *Effective Java*. Addison-Wesley Professional, 2017.

[24] K. Cwalina and B. Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Pearson Education, 2008.

[25] A. Faulring, B. A. Myers, Y. Oren, and K. Rotenberg, "A case study of using HCI methods to improve tools for programmers," in *International Workshop on Co-operative and Human Aspects of Software Engineering*. IEEE, 2012, pp. 37–39.

[26] T. Grill, O. Polacek, and M. Tscheligi, "Methods towards API usability: A structural analysis of usability problem categories," in *International Conference on Human-Centred Software Engineering*, 2012, pp. 164–180.

[27] G. M. Rama and A. Kak, "Some structural measures of API usability," *SoftwarePractice and Experience.*, vol. 45, no. 1, pp. 75–110, 2013.

[28] T. Scheller and E. Kuhn, "Automated measurement of API usability: The API concepts framework," *Information and Software Technology*, vol. 61, pp. 145–162, 2015.

[29] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 1–25, 2013.

[30] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM, 2007, pp. 35–44.

[31] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 383–392.

[32] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static API-misuse detection," in *International Conference on Mining Software Repositories*. IEEE, 2019, pp. 265–275.

[33] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online Q&A forum reliable?" in *International Conference on Software Engineering*. ACM, 2018, pp. 886–896.

[34] N. Katirtzis, T. Diamantopoulos, and C. Sutton, "Summarizing software API usage examples using clustering techniques," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2018, pp. 189–206.

[35] J. Wang, J. Han, and C. Li, "Frequent closed sequence mining without candidate maintenance," *Transactions on Knowledge and Data Engineering*, vol. 19, no. 8, pp. 1042–1056, 2007.

[36] E. Murphy-Hill, C. Sadowski, A. Head, J. Daughtry, A. Macvean, C. Jaspan, and C. Winter, "Discovering API usability problems at scale," in *International Workshop on API Usage and Evolution*. ACM, 2018, pp. 14–17.

[37] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[38] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[39] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *International Conference on Software Engineering*. ACM, 2016, pp. 1157–1168.

[40] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Software Engineering Notes*. ACM, 2005, pp. 187–196.

[41] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[42] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[43] R. Lämmel and S. P. Jones, *Scrap your boilerplate: a practical design pattern for generic programming*. ACM, 2003.

[44] "html5-boilerplate," https://github.com/h5bp/html5-boilerplate.

[45] "Boilerplate code definition of quora," https://www.quora.com/What-is-boilerplate-code.

[46] "How to avoid writing duplicate boilerplate code for requesting permissions?" https://stackoverflow.com/questions/39080095/how-to-avoid-writing-duplicate-boilerplate-code-for-requesting-permissions.

[47] "How to avoid writing boilerplate code in java swing mvc?" https://stackoverflow.com/questions/26154225/how-to-avoid-writing-boilerplate-code-in-java-swing-mvc.

[48] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[49] "Project lombok," https://projectlombok.org,.

[50] T. K. Moon, "The expectation-maximization algorithm," *Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[51] M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157 – 173, 2016.

[52] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *International Conference on Automated Software Engineering*. ACM, 2014, pp. 313–324.

[53] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[54] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," *Software*, vol. 22, no. 6, pp. 23–29, 2005.

[55] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *International Conference on Software Engineering*. IEEE, 2013, pp. 672–681.

[56] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *Internatioanl Conference on Mining Software Repositories*. IEEE, 2012, pp. 12–21.

[57] "Probabilistic API mining implementation," https://github.com/mast-group/API-mining.
[58] "Ap-ted implementation," https://github.com/DatabaseGroup/apted.
[59] "Community detection package," https://python-louvain.readthedocs.io.
[60] "How to avoid boilerplate code when loading images with picasso library," https://stackoverflow.com/questions/32167948/how-to-avoid-boilerplate-code-when-loading-images-with-picasso-library.
[61] "Spring framework," https://spring.io.
[62] "Greendao," http://greenrobot.org/greendao/documentation/introduction.
[63] "Android API 26 release note," https://developer.android.com/about/versions/oreo/android-8.0-changes#fvbi-signature.
[64] "Butterknife," https://jakewharton.github.io/butterknife/.
[65] M. Weiser, "Program slicing," in *International Conference on Software Engineering*. IEEE, 1981, pp. 439–449.