API Design Implications of Boilerplate Client Code

Daye Nam
Carnegie Mellon University
dayen@cs.cmu.edu

***Abstract*—Designing usable APIs is critical to developers' productivity and software quality but is quite difficult. In this paper, I focus on "boilerplate" code, sections of code that have to be included in many places with little or no alteration, which many experts in API design have said can be an indicator of API usability problems. I investigate what properties make code count as boilerplate, and present a novel approach to automatically mine boilerplate code from a large set of client code. The technique combines an existing API usage mining algorithm, with novel filters using AST comparison and graph partitioning. With boilerplate candidates identified by the technique, I discuss how this technique could help API designers in reviewing their design decisions and identifying usability issues.**

I. Introduction

APIs are ubiquitous, and their usability is a significant concern for API designers [1]–[6]. In particular, API designers report that anticipating how developers will use their APIs is difficult [7]. Although online sources (*e.g.*, GitHub) contain ample amounts of real client code, designers report that there are not so many automated ways to mine usability data [7].

In this paper, I focus on one particular grievance that developers express repeatedly [8]–[11] in online discussions about APIs: *boilerplate code*, "sections of code that have to be included in many places with little or no alteration" [12]. From an API designer's perspective, *the existence of boilerplate code may serve as an indicator of poor API usability*. The need for boilerplate code often indicates that the API does not directly provide the methods that programmers need, or the API offers unnecessary flexibility [13]. Thus, reviewing boilerplate client code instances may help improve an API's design.

I present an automated technique for identifying instances of boilerplate API client code. The technique combines an existing API usage pattern mining approach with novel filters using AST comparison and graph partitioning. Finally, I discuss the API design implications of the mined boilerplate candidates.

II. Common Properties of Boilerplate Code

To help understand the characteristics of boilerplate code, I collected boilerplate code examples and definitions from the literature and online sources (*e.g.*, GitHub, Stack Overflow), and distilled common properties that are operationalizable:

1) *High frequency*: Boilerplate code occurs frequently among the API client code [11], [12]. 2) *Localized*: The statements constituting boilerplate code are usually *closely located near each other*, rather than spread over multiple methods or files. 3) *Little structural variation*: The boilerplate code instances appear in *similar form without significant variation* [3], [14].

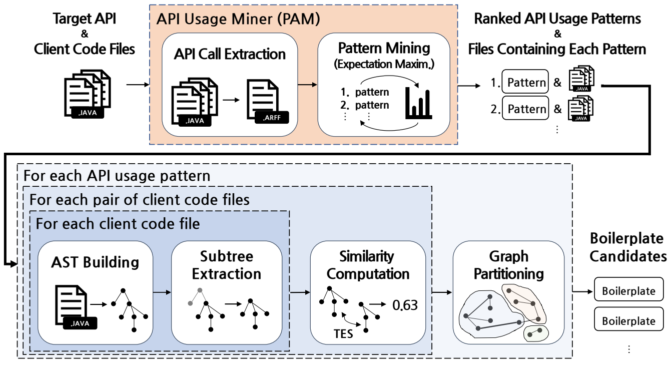

Fig. 1. Overview of the mining process and the steps involved.

III. Mining Boilerplate Code

I developed a technique to identify candidate boilerplate code, *i.e.*, instances containing calls to a target API and satisfying the above three properties, from software repositories. I give a brief overview here (Figure 1). See [15] for details.

1) API Usage Pattern Mining: I start from an existing API usage pattern miner, PAM [16], which satisfies the high-frequency property. PAM probabilistically identifies sequences of API calls that are frequently called together (*i.e.*, API usage patterns). The main novelty of my technique lies in filtering down the many API usage patterns returned by PAM to a short list of boilerplate candidates. I describe these filters next.

2) AST Extraction: The (structural) context around the API calls is important to determine if an API usage pattern is also a boilerplate candidate. Given a list of API usage patterns (*i.e.*, frequent API call sequences) and client code containing instances of those patterns, I extract and post-process the ASTs from the files. Per Property 2 above, I use a slicing heuristic: I extract the smallest subtrees of each client code file's AST, which completely encompass the target API call sequence.

3) Graph Partitioning: Finally, I check Property 3: whether the API call sequence is used consistently in similar structures throughout the client code. I devise an approach to 1) compute the similarity between all pairs of subtrees containing the API call sequence using Tree Edit Distance [17]; and 2) cluster together similar subtrees. Given a graph where the subtrees are the nodes, and the similarity scores are the weights on the edges, intuitively, if there is a cluster having a number of

```
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 1. Boilerplate code instance of Android View client codes.

subtrees, and the similarity between them is high, the cluster (*i.e.*, specific use case) can be a boilerplate candidate.

IV. API Review

In this section, I show how the mined boilerplate candidates could help API designers review and improve their APIs. I analyzed 13 Java APIs, which contain at least one known boilerplate code example. The proposed technique could identify 69% of known boilerplate instances with 56% precision, and returned reasonable number of candidates for the manual review. I manually reviewed all 59 boilerplate candidates identified automatically, looking for causes and potential improvements to the API design. (See [15] for details.) Given the space constraints, I discuss only two boilerplate candidates here.

Android View. Listing 1 shows a classic boilerplate code for Android View, identified automatically. This 5 lines of code is 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`. I could observe that 217 files out of 518 files use this method for 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)` from Android 8.0 [18]. This shows that 1) API designers care about the boilerplate code instances reducing the API usability, and 2) informing API designers about the boilerplate code can lead to usability improvement.

Another way to reduce this type of boilerplate is to use annotation libraries (*e.g.*, Spring Framework [19]). There are several libraries providing annotation supports for this boilerplate (e.g., `@BindView` of ButterKnife [20]), by helping users easily map the view ID declared in XML layout file with Java's variable. In practice, a number of clients actually adopt them, which can be a signal for API designers to update their API similarly, or recommend these libraries to their clients for better usability. Also, this type of boilerplate candidates and whether they could be abstracted by an annotation framework might be useful for the helper library designers as well, in understanding the needs of users, and developing a new library.

```
private boolean checkPermission() {
  if (ActivityCompat.checkSelfPermission(this, Manifest.permission.
          ↪ ACCESS_FINE_LOCATION) != PackageManager.
          ↪ PERMISSION_GRANTED &&
    ActivityCompat.checkSelfPermission(this, Manifest.permission.
          ↪ ACCESS_COARSE_LOCATION) != PackageManager.
          ↪ PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.
          ↪ ACCESS_FINE_LOCATION},
          ↪ LOCATION_PERMISSION_REQUEST_CODE);
    return false; }
  return true; }
```

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

Android ActivityCompat. Listing 2 shows potential boilerplate code mined from Android's ActivityCompat, checking whether a context has an appropriate permission, containing one or multiple uses of one API call: `ActivityCompat.checkSelfPermission`. MARBLE identified that out of 486 files importing ActivityCompat, this pattern is used in the same format in 36. Also, many clients built their own wrapper to check permission (including Listing 2) due to its verboseness and inconvenience.

One potential redesign for better usability in this case is to provide a more abstract method that handles permission checking: if the permission is not granted, the method internally requests the permission and sends the results to the user, and if it is granted already, the code continues. Another potential mitigation is to provide helper functions from API level, for popularly used permission sets, such as `hasLocationPermission` or `hasStoragePermission`.

However, I hypothesize that there could be a design rationale behind the current design, *e.g.*, possibly to improve the debuggability of the code. Although the proposed abstraction could help new Android developers get started with the API, and lead to less code in these common cases, this could also lead to less debuggable code. This trade-off, however, 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. Thus, analyzing boilerplate candidates could help API designers review to what extent their design rationale is valid, and reconsider trade-offs.

V. Conclusions and Future Work

I presented the novel problem of mining software repositories to identify candidate boilerplate code, as a potential API usability issue. I devised a new boilerplate mining algorithm based on three properties of boilerplate code (high frequency, locality, and limited structural variation), and showed that it could help API designers identify unknown usability issues. Future work could survey API designers to get wider input on what properties of boilerplate code they are most interested, and adjust the algorithm accordingly. Also, an extensive evaluation could involve deploying the tool for use by many real designers in the industry, in hopes of actually helping them identify and eliminate some boilerplate code from clients of their APIs, and review their design decisions.

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] "Collect GeometrySystem → drake_visualizer boilerplate by SeanCurtis-TRI pull request #8526 RobotLocomotion/drake," https://github.com/RobotLocomotion/drake/pull/8526.

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

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

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

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

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

[14] "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.

[15] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu, "Marble: Mining for boilerplate code to identify API usability problems," in *International Conference on Automated Software Engineering*. IEEE, 2019.

[16] 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.

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

[18] "Android API 26 release note," https://developer.android.com/about/versions/oreo/android-8.0-changes#fvbi-signature.

[19] "Spring framework," https://spring.io.

[20] "Butterknife," https://jakewharton.github.io/butterknife/.