Code Implementation Recommendation for Android GUI Components

Yanjie Zhao
Monash University
Melbourne, Australia

Li Li*
Monash University
Melbourne, Australia

Xiaoyu Sun
Monash University
Melbourne, Australia

Pei Liu
Monash University
Melbourne, Australia

John Grundy
Monash University
Melbourne, Australia

ABSTRACT

We present a prototype tool *Icon2Code*, targeted to helping app developers more quickly implement the callback functions of complex Android GUI components by recommending code implementations learnt from similar GUI components from other apps. Given an icon or UI widget provided by designers, *Icon2Code* first queries a large pre-established database to locate similar icons that other apps have utilized. It then leverages a collaborative filtering model to suggest the most relevant APIs and their usage examples associated with the intended behaviours of these icons. Experimental results on 5,000 randomly selected real-world apps show that *Icon2Code* is useful and effective in recommending code examples for implementing the behaviours of complex GUI components. It has over 50% of success rate when only one recommended API is taken into account, and over 94% of success rate if 20 APIs are considered. The video demo can be found at https://youtu.be/pM3ZBGrQTdQ.

KEYWORDS

Android, App Development, Collaborative Filtering, Icon Implementation, API Recommendation

ACM Reference Format:
Yanjie Zhao, Li Li, Xiaoyu Sun, Pei Liu, and John Grundy. 2022. Code Implementation Recommendation for Android GUI Components. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516849

1 INTRODUCTION

More than 3.48 million Android apps are currently available on the Google Play store [26]. This number will continue to grow significantly. Users have many choices from the enormous number of downloadable Android apps, and developers need to develop and refresh their apps promptly [14, 16, 30]. Managing a short release cycle is not easy for developers because they are often under pressure to fix vulnerabilities [5], solve compatibility issues [2, 12, 13, 29], and learn new development methodologies, libraries, and the most recent technologies [15, 28, 31]. To assist them, researchers have proposed many approaches to ease the development work of developers [7, 9–11, 19, 27].

Unfortunately, to the best of our knowledge, no existing approaches have been proposed to support the code implementation for the GUI component event handlers of Android apps. A GUI is a common feature of all mobile apps, which are event-centric programs that provide a rich graphical user interface to interact with the users [17]. Since managing the complex and intertwined callback events from user interaction usually requires a lot of coding and debugging work [27], this increases the complexity of implementing the mobile app GUI to some extent. Fortunately, developers can implement the functionalities in callback methods with the help of functional APIs. Nevertheless, it is still a time-consuming and labour-intensive task to accurately utilize proper functional APIs to implement callback methods for feature requirements [27].

Based on the idea that similar GUI components are normally designed to trigger similar application behaviour, we propose a prototype tool named *Icon2Code*[1] to learn similar GUI callback implementations of other Android applications to help developers effectively implement the callback methods of their own GUI components. *Icon2Code* is intended to extract common implementations to help app developers accomplish the development of interactions driven by GUI components, e.g., the heart-shaped icon for *like*. *Icon2Code* first uses static analysis to analyze existing apps and build a mapping relationship from GUI components to the callback methods associated with them. For each callback method, *Icon2Code* extracts its call graph and collects all accessed APIs, including APIs of third-party libraries. Finally, *Icon2Code* achieves the purpose of recommendation based on a collaborative filtering algorithm. Taking GUI components (i.e. icons) as input, *Icon2Code* outputs code implementations learned from the apps containing similar GUI components.

2 MOTIVATION

As stated by Chen et al. [3], GUI design and GUI implementation are two autonomous activities in the process of developing the GUI of an app. Since creating an intuitive interface with a good user experience is critical to the success of an app in a competitive market, a professional designer usually does the former. The latter involves programming the GUI interface itself, such as the layout,

*Corresponding author.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516849

[1]The project is available at https://github.com/carol233/Icon2Code.

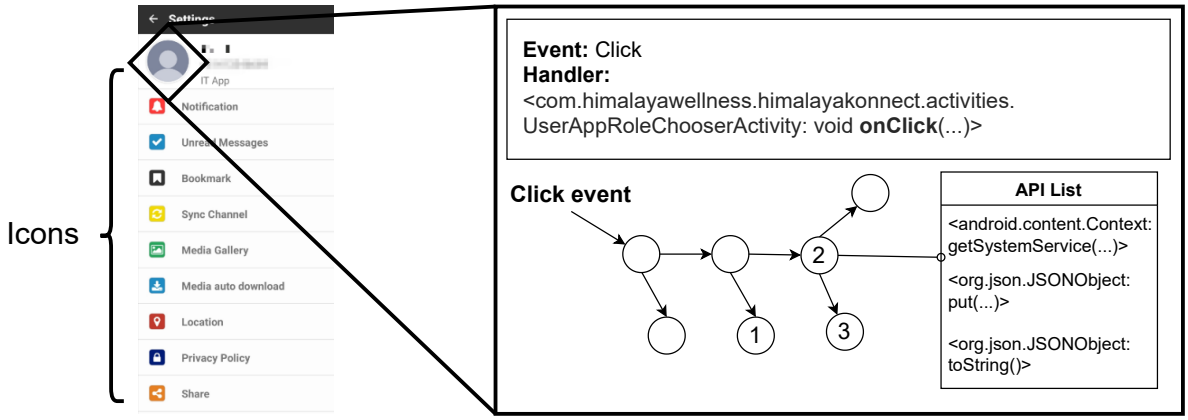

Figure 1: An example of icon-bound GUI components and icon-related event handler callback methods.

constraints, and user interaction processing of GUI widgets, e.g., what should happen when a button is clicked. Chen et al. [3] propose a neural machine translator to convert GUI design images into GUI skeletons, but they haven't endeavoured to help developers implement the user interaction handling code of GUI components.

Listing 1: Examples of API usages in the method (e.g., Node 2 in Figure 1) reached by the click event. The method has accessed both Android APIs and third-party library APIs.

```
1  //Node 2
2  TelephonyManager telephonyManager = (TelephonyManager) this.
       la.getSystemService("phone");
3  String str = telephonyManager.getNetworkCountryIso().
       toUpperCase();
4  JSONObject jSONObject = new JSONObject();
5  jSONObject.put("isoccode", str);
6  new b(..., jSONObject.toString(), ...);
```

Figure 1 shows a typical GUI page extracted from the Android app *com.himalayawellness.himalayakonnect*, which contains various GUI components, also referred to as **icons** in this paper. After these GUI components accept user inputs – such as clicking, sliding, or other interactions – the app needs to respond accordingly. For example, when a user clicks the *Profile* icon on the upper left, it will switch to a new page that allows the user to set their profile data. **Callback methods** are utilized to achieve these behaviour changes driven by user input events. Each GUI icon should have at least one event handling callback method. However, implementing these callback methods is usually complicated, involving method calls that access multiple Android APIs and possible third-party libraries.

Listing 1 shows a simplified code example extracted from one of the methods in the call chain (i.e., node 2) triggered by the callback method *onClick*. This single method accesses at least 3 APIs, including Android official APIs and third-party library APIs. A callback associated with an icon may contain multiple such methods. Furthermore, an app's UI page may contain dozens of icons, which makes it harder to implement and debug their related callback methods effectively [20, 25]. As far as we know, no existing approaches are proposed to assist developers in implementing the complex event-driven callback methods related to Android GUI icons.

3 *ICON2CODE*

Figure 2 shows an outline of our *Icon2Code* tool. It has three key modules: (1) Database Construction Module (DCM), (2) Similarity Calculation Module (SCM), and (3) API Recommendation Module (ARM).

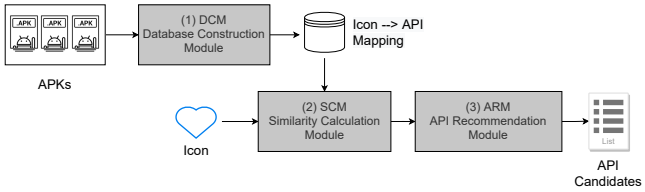

Figure 2: The architecture of *Icon2Code*.

3.1 DCM: Database Construction Module

We recommend code implementations for GUI icons by learning from the code implementations of existing apps that use similar icons in their GUIs. To this end, the goal of the first module of *Icon2Code* is to preprocess the Android apps to build a database that maps icons to its specific code implementations. The code implementations here ignore user-defined APIs or methods and only include JDK, Android official APIs and third-party library APIs. Figure 3 shows the working pattern of this module.

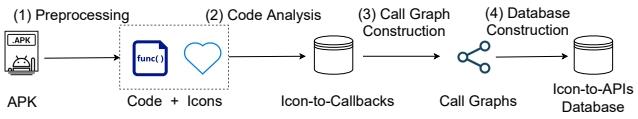

Figure 3: The working process of the DCM module.

3.1.1 Preprocessing. The key objective of the first step is to preprocess the Android APKs to extract useful information (e.g., the icons) in preparation for further analysis. The Android application package (APK) is a file format used to distribute and install applications on the Android OS. There are usually two major icon forms in APKs: the *vector* defined by an XML file and the *image*

presented as an image file (e.g., PNG format). Since the former does not come as specific images, we ignore it in this work and only consider the latter. To improve the utility of the recommended code, we should make the recommended APIs consistent with the target SDK version of the target app. Therefore, when disassembling the Android APKs, we extract the target SDK versions, e.g., the value of *targetSdkVersion*, by parsing the manifest files.

Listing 2: Examples of using XML attributes to bind icons with GUI components.

```
1  //Example 1: ImageView, android:src
2  <ImageView android:src="@drawable/next_btn"
3   android:id="@+id/nextBtn"
4   android:contentDescription="@string/
        next_button_content_desc"
5   android:onClick="onClick"/>
6
7  //Example 2: Button, android:background
8  <Button android:background="@drawable/button_cancel"
9   android:id="@+id/cancel_btn"
10  android:text="@string/cancel"/>
```

3.1.2 Code Analysis. With static analysis techniques, this step establishes a mapping between the icon-bound GUI components and their corresponding callback method code that responds to user interaction events. The first subtask of this step is to break down how the icons are bound to the GUI components of the application. The XML attributes in the layout configuration files of an app are responsible for binding the icons to the GUI components, such as the source of ImageView (i.e., android:src) and the background of the Button (i.e., android:background) displayed in Listing 2 at lines 2 and 8, respectively. Table 1 shows the list of such attributes we have taken into account, which has already been leveraged by other techniques [1]. After locating the GUI components bound

Table 1: The set of attributes liable for binding icons to GUI components.

Attribute	Explanation
android:src	Set a drawable as the content of the view (e.g., ImageView).
android:background	Set a drawable as the background of the view.
android:drawableRight	Set a drawable to the right of the text.
android:drawableTop	Set a drawable on top of the text.
android:drawableLeft	Set a drawable to the left of the text.
android:drawableBottom	Set a drawable below the text.
android:drawableEnd	Set a drawable at the end of the text.
android:drawableStart	Set a drawable at the start of the text.

to the target icons using XML attributes, the second subtask of this step is to infer the callback methods associated with the GUI components. Similar to the binding between GUI components and icon files, developers can also define callback methods through XML attributes. For example, in Listing 2, the callback method that is triggered when the ImageView is clicked (i.e., *onClick*) is specified by the attribute *android:onClick* (line 5). This type of static binding can be resolved through the layout configuration files similar to the first subtask.

Callback methods can also be bound dynamically in the program code. For the same *onClick* callback method, rather than the XML attributes (line 5 in Listing 2), developers can use the code presented in Listing 3 to achieve the same goal. Although it is more challenging to identify this type of binding in an Android APK, it can be concluded from observation that in most cases, the callback methods are added after the GUI components are created (e.g., *findViewById()* in line 4). Finally we can map the icon to its dynamically defined callback method(s) by statically linking the information obtained through these two subtasks.

Listing 3: An example of dynamically defining an icon's event handler (i.e., callback method) through program code.

```
1  public class MusicWallpaper extends Activity implements View
       .OnClickListener {
2   public void onCreate(Bundle bundle) {
3    setContentView(R.layout.layoutlagu);
4    ImageView v = (ImageView) findViewById(R.id.nextBtn);
5    //Binding callback method to the icon
6    v.setOnClickListener(this);
7   }
8   @Override
9   public void onClick(View view) {
10   //This is the callback method
11   ...
12 }}
```

3.1.3 Call Graph Construction. Based on the previous step, we collect the APIs accessed by the callback methods to map from icons to their corresponding APIs invoked when the icon (GUI component) receives user interaction input. Unfortunately, as our motivating example illustrates, a given callback method can use many other methods, and each can access a set of APIs, thus making it not easy to gather the whole API set. We therefore construct a call graph for each callback method viewed as an entry point, where nodes represent methods and edges represent method invocations, to ease the extraction of APIs.

3.1.4 Database Construction. For the mappings between icons previously obtained and the API sets accessed by traversing the call graphs and extracting methods in the graphs, *Icon2Code* utilizes these to generate a database, which is also the ground truth used to support the following recommendation. Ideally, one way to improve the reliability of the API recommendation approach is to use more apps for training. In addition, *Icon2Code* further records complete API usage examples (such as the code snippets shown in Listing 1) into the database to support *Icon2Code* recommending API usage examples to developers.

3.2 SCM: Similarity Calculation Module

Taking an icon as input, SCM aims to locate similar icons from the pre-established *icon* → *APIs* database. To control the number of most similar icons, a configurable parameter m is introduced, and *Icon2Code* only selects the top-m most similar icons. Icons can be associated with icon purpose-describing texts given by developers in Android apps. Other than an immediate comparison between icon images, we thus also leverage alternate text similarity to track down the most similar icons.

Image Similarity Calculation. We employ three algorithms to calculate the similarity between iconic images – Oriented FAST and Rotated BRIEF (ORB) algorithm [22], Locality Sensitive Hashing (LSH) algorithm [4], and the Histogram algorithm [6]. This hybrid image similarity calculation method can reduce the random error

caused by a single method. Given two images p and q, their image similarity is measured via Formula 1, where the fusion similarity threshold is assigned as 0.85. If the maximum similarity from the three algorithms is greater than or equal to 0.85, the maximum is taken as the final; otherwise, the minimum is considered the final.

$$\begin{gathered} Let\ Max_s = max(ORB(p,q), LSH(p,q), Histogram(p,q)), \\ Let\ Min_s = min(ORB(p,q), LSH(p,q), Histogram(p,q)), \\ Sim_{image}(p,q) = \begin{cases} Max_s, & Max_s \geq 0.85 \\ Min_s, & Max_s < 0.85 \end{cases} \end{gathered} \tag{1}$$

Text Similarity Calculation. We have determined three major sources that furnish alternative text for iconic GUI components: (1) **S1**: The icon's reference name (e.g., given by *android:src*), often describing the function of the icon; (2) **S2**: The id name of the view to which the icon is bound (e.g., given by *android:id*), often portraying the function of the view hosting the icon; and (3) **S3**: The alternative text defined by *android:contentDescription* or *android:text*, set to describe the function of the view. Take Listing 2 as an instance – alternative texts {S1, S2, S3} of Example 1 are {next_btn, nextBtn, next_button_content_desc}. All three alternative of these text values specify the icon's purpose and are similar to some degree. *Edit distance* is widely used to figure the similarity of two text strings, by computing the minimum number of edit operations demanded to alter one text into the other [21]. *Levenshtein distance* is such a type of generally used edit distance [8], upon which two texts' syntactic similarity can be represented with the *Levenshtein ratio* [23]. Given two texts a and b, their *Levenshtein ratio* can be calculated by Formula 2. For the two icons p and q, their alternate text similarity is determined by Formula 3, where p' and q' are the according alternative text, and w_1, w_2, w_3 are the weights of each type of alternative text, namely $S1, S2, S3$.

$$LevenshteinRatio(a,b) = 1 - \frac{LevenshteinDistance(a,b)}{|a|+|b|} \tag{2}$$

$$\begin{aligned} Sim_{text}(p',q') = \quad & w_1 \times LevenshteinRatio_{S1}(p',q') \quad + \\ & w_2 \times LevenshteinRatio_{S2}(p',q') \quad + \\ & w_3 \times LevenshteinRatio_{S3}(p',q') \end{aligned} \tag{3}$$

These two similarity calculation algorithms are aggregated to calculate the overall similarity of two icons via Formula 4, α and β representing the weights of Sim_{image} and Sim_{text}, individually.

$$Sim(p,q) = \alpha \times Sim_{image}(p,q) + \beta \times Sim_{text}(p',q') \tag{4}$$

3.3 ARM: API Recommendation Module

ARM aims to recommend suitable callback code APIs for the input icon under development by learning from a set of similar app GUI elements and their callback code implementations. We leverage collaborative filtering [24] for our tool's recommendation of API usages. Collaborative filtering has usually been utilized to recommend items for users to purchase dependent on their past shopping records or other users' records with similar buying practices. In our *Icon2Code* tool, an icon plays the role of a *user*, and each API plays the role of an *item*. A *rating* (e.g., a numerical value) is further associated with a user and an item. The objective of ARM is to recommend users (icons) a list of items (APIs) to purchase (to access).

Relying on the m most similar icons collected by SCM module, *Icon2Code* first calculates the number of APIs (k) invoked by the related callback methods of the m icons and models them into a $(m+1) * k$ matrix. Table 2 demonstrates such an example, where the selected m icons $i_1 \rightarrow i_m$ plus the one being edited i_{edit} are represented as rows and APIs are represented as columns. For the m icons, each of their cells is assigned to either true (1) or false (0) in the matrix, indicating whether the icon-related callback methods have invoked the corresponding API or not. For example, cell (i_1, api_k) is set to be 0, representing that callbacks of icon i_1 has not invoked api_k. For the icon being edited (i.e., i_{edit} in the last row), all of its cells are initialized to unknown (-1). The objective of this module is then refined to predict possible values for those unknown cells. The cells assigned higher values, i.e., corresponding APIs, will then be recommended for developers to implement the icon-related callback method.

Table 2: An example encoding matrix.

	api_1	api_2	...	api_k
i_1	1	0	1	0
i_2	1	1	0	1
...	1	1	1	0
i_m	0	1	1	0
i_{edit}	-1	-1	-1	-1

The possibility of recommending a given API api to i_{edit} is computed via Formula 5 [24], where $neighbours(i_{edit})$ is the set of the m most similar icons, $sim(i_{edit}, i)$ is defined by Formula 4, and $\bar{r_{i_{edit}}}$ and $\bar{r_i}$ are the mean ratings of i_{edit} and i, respectively. In the implementation, $\bar{r_i}$ and $r_{i,api}$ are determined by the encoding matrix. For example, for the encoding matrix in Table 2, $\bar{r_i}$ can be computed by measuring the average rating of the cells in the row relating to i. For $\bar{r_{i_{edit}}}$, we assign its value to 0.8 following the practice of Nguyen et al. [18]

$$p_{i_{edit},api} = \bar{r_{i_{edit}}} + \frac{\sum_{i \in neighbours(i_{edit})} (r_{i,api} - \bar{r_i}) \cdot sim(i_{edit}, i)}{\sum_{i \in neighbours(i_{edit})} sim(i_{edit}, i)} \tag{5}$$

The output of *Icon2Code* is a list of Android API candidates that are ranked by the scores returned from Formula 5.

4 EVALUATION

We have evaluated *Icon2Code* from four aspects and all of our experiments leverage ten-fold cross-validation. First, we validated the performance and effectiveness of *Icon2Code* on a dataset constructed with 47,827 icons extracted from about 5,000 randomly selected apps. With default parameters ($m = 20$, $\alpha = 1$, $\beta = 0$), *Icon2Code* can achieve over 50% of the success rate when only one recommended API is considered and it can reach over 94% if the number is increased to 20. Second, we explored the impact of changing the number of similar icons (i.e., m) by designing multiple sets of experiments with different numbers of neighbours, i.e., $m \in \{5, 10, 15, 20, 25, 30\}$ ($m = 5$ means that *Icon2Code* builds the encoding matrix with five similar icons). The results show that on our dataset, $m = 20$ is a suitable setting for *Icon2Code*. Third, we explored the impact of different similarity calculation methods, i.e., the value of α and β defined in Section 3.2. We compared the default setting with another four settings with different weights, i.e., (0.8, 0.2), (0.5, 0.5), (0.2, 0.8), and (0, 1). Surprisingly, the text-only setting shows the best performance, which shows that developers can resort to using alternative texts to find good matching GUI

callback code suggestions. Finally, we explored the performance of *Icon2Code* over different groups of training sets. We divided the original training dataset into three groups: (1) All icons with no more than five APIs accessed by their related callback methods, (2) All icons with over five but no more than ten APIs accessed by their related callback methods, and (3) All icons with more than 10 APIs accessed by their related callback methods. Following the same experimental setting, we further re-ran *Icon2Code* on the aforementioned three training groups, individually. Our new experimental results suggest that expanding the quantity of APIs accessed by icons selected for training improves the performance of *Icon2Code* to some extent.

5 CONCLUSION

We have proposed a prototype tool *Icon2Code* to advise on APIs to help implement the callback functions of iconic GUI components. *Icon2Code* leverages icon images and their alternative texts to locate the most similar icons to the icon under development. It then employs a collaborative filtering algorithm to get the output of the recommended APIs and usage examples from existing apps.

ACKNOWLEDGEMENTS

This work is supported by ARC Laureate Fellowship FL190100035, Discovery Early Career Researcher Award DE200100016, Discovery Project DP200100020.

REFERENCES

[1] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting behavior anomalies in graphical user interfaces. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 201–203.

[2] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-Scale Study of Application Incompatibilities in Android. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*.

[3] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.

[4] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[5] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2019. Understanding the Evolution of Android App Vulnerabilities. *IEEE Transactions on Reliability (TRel)* (2019).

[6] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. 2015. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann.

[7] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[8] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[9] Bo Li, Qiang He, Feifei Chen, Xin Xia, Li Li, John Grundy, and Yun Yang. 2021. Embedding App-Library Graph for Neural Third Party Library Recommendation. In *The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*.

[10] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Parameter values of Android APIs: A preliminary study on 100,000 apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 584–588.

[11] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).

[12] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*.

[13] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2020. CDA: Characterising Deprecated Android APIs. *Empirical Software Engineering (EMSE)* (2020).

[14] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *arXiv preprint arXiv:1709.05281* (2017).

[15] Li Li, Timothée Riom, Tegawendé F Bissyandé, Haoyu Wang, Jacques Klein, and Yves Le Traon. 2019. Revisiting the Impact of Common Libraries for Android-related Investigations. *Journal of Systems and Software (JSS)* (2019).

[16] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and Characterizing Silently-Evolved Methods in the Android API. In *The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track (ICSE-SEIP 2021)*.

[17] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.

[18] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. Focus: A recommender system for mining api function calls and usage patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1050–1060.

[19] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2015. Recommending API usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 795–800.

[20] Danilo Dominguez Perez and Wei Le. 2017. Generating predicate callback summaries for the Android framework. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 68–78.

[21] Eric Sven Ristad and Peter N Yianilos. 1998. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (1998), 522–532.

[22] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.

[23] Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. 2016. JUNITMZ at SemEval-2016 task 1: Identifying semantic similarity using Levenshtein ratio. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*. 702–705.

[24] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web*. Springer, 291–324.

[25] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–37.

[26] Statista Research Department. 2021. Number of apps available in leading app stores as of 1st quarter 2021. (2021). https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ [accessed 15-June-2021].

[27] Weizhao Yuan, Hoang H Nguyen, Lingxiao Jiang, Yuting Chen, Jianjun Zhao, and Haibo Yu. 2019. API recommendation for event-driven Android application development. *Information and Software Technology* 107 (2019), 30–47.

[28] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. *IEEE Transactions on Software Engineering* (2021).

[29] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *The 44th International Conference on Software Engineering (ICSE 2022)*.

[30] Yanjie Zhao, Li Li, Xiaoyu Sun, Pei Liu, and John Grundy. 2021. Icon2Code: Recommending Code Implementations for Android GUI Components. *Information and Software Technology (IST)* (2021).

[31] Yanjie Zhao, Li Li, Haoyu Wang, Qiang He, and John Grundy. 2022. API-Matchmaker: Matching the Right APIs for Supporting the Development of Android Apps. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3146831