How Maintainability Issues of Android Apps Evolve

Ivano Malavolta*, Roberto Verdecchia*†, Bojan Filipovic*, Magiel Bruntink‡, Patricia Lago*
*Vrije Universiteit Amsterdam, The Netherlands - {i.malavolta | p.lago}@vu.nl, b.filipovic@student.vu.nl
†Gran Sasso Science Institute, L'Aquila, Italy - roberto.verdecchia@gssi.it
‡Software Improvement Group - Amsterdam, The Netherlands - m.bruntink@sig.eu

***Abstract*—*Context*. Android is the largest mobile platform today, with thousands of apps published and updated in the Google Play store everyday. Maintenance is an important factor in Android apps lifecycle, as it allows developers to constantly improve their apps and better tailor them to their user base.**
***Goal*. In this paper we investigate the evolution of various maintainability issues along the lifetime of Android apps.**
***Method*. We designed and conducted an empirical study on 434 GitHub repositories containing open, real (i.e., published in the Google Play store), and actively maintained Android apps. We statically analyzed 9,945 weekly snapshots of all apps for identifying their maintainability issues over time. We also identified maintainability hotspots along the lifetime of Android apps according to how their density of maintainability issues evolves over time. More than 2,000 GitHub commits belonging to identified hotspots have been manually categorized to understand the context in which maintainability hotspots occur.**
***Results*. Our results shed light on (i) how often various types of maintainability issues occur over the lifetime of Android apps, (ii) the evolution trends of the density of maintainability issues in Android apps, and (iii) an in-depth characterization of development activities related to maintainability hotspots. Together, these results can help Android developers in (i) better planning code refactoring sessions, (ii) better planning their code review sessions (e.g., steering the assignment of code reviews), and (iii) taking special care of their code quality when performing tasks belonging to activities highly correlated with maintainability issues. We also support researchers by objectively characterizing the state of the practice about maintainability of Android apps.**
***Conclusions*. Independently from the type of development activity, maintainability issues grow until they stabilize, but are never fully resolved.**

***Index Terms*—Maintainability, Android, Empirical study**

I. Introduction

As of March 2018, there are more than 3.3 million Android applications available [1], with more than one thousand apps being published *everyday* [2]. Also, according to the official Android developer portal [3] there are more than 1.5 billion downloads from Google Play Store *every month*. A platform of such a large scale leads to an extremely crowded market and fierce competition. If developers are to succeed in such a competitive environment, it is of paramount importance that the mobile apps they produce are of extremely *high quality*.

Software maintenance is the activity of modifying a software product after its delivery in order to improve performance, add functionalities or perform corrective tasks on the existing product [4]. Software maintainability can be defined as the property of software that provides insights about how easily a software system (in this case an Android app) can be maintained [4]. In principle, apps with higher maintainability can be released and updated with less effort and provide the users with high quality features. Maintenance can be seen as one of the most important activities within the mobile app lifecycle. For example, updates of widely popular mobile apps like Facebook are consistently published *on a daily basis* [5].

Apart from the Android guidelines [6], there is little guidance about Android apps maintainability. If developers would be able to characterize their own apps w.r.t. the maintainability issues they are prone to encounter over timethey would be able to make better informed development choices.

In this paper we present a *large-scale empirical study on the evolution of statically-detectable maintainability issues across the Android ecosystem*. In particular, in this study we refer to "maintainability issue" as code that is classified as *high risk* by the Software Analysis Toolkit (SAT) [7], an industrial toolset developed by the Software Improvement Group (SIG) [8]. From a methodological perspective, we firstly built a dataset of 434 Android apps that are (i) open (i.e., available as open-source projects in GitHub), (ii) real (i.e., distributed through the Google Play Store), and (iii) actively maintained (i.e., no single-commit or toy projects). Then, we (i) mined their GitHub repositories and extracted 9,945 weekly snapshots[1] from 106,689 commits, (ii) analysed each snapshot via an industrial tool for static analysis, and (iii) identified the occurrences of 5 types of maintainability issues in each snapshot. Afterwards, we manually analyzed 1,230 apps for building a reusable taxonomy of the trends in which the density of maintainability issues of Android apps can evolve over time. In order to do so, we considered the notion of commit-level *issue density (cd)* [9] reported in Formula 1. Moreover, we identified 3,374 maintainability hotspots, defined as the points along the lifetime of an Android app where developers are injecting an anomalous number of maintainability issues with respect to the current app size. Finally, in order to understand the context in which maintainability hotspots occur, we investigated on the (self-reported) activities performed by developers in the context of all identified maintainability hotspots. We carried out the latter step by manually inspecting and categorizing 2,112 GitHub commits belonging to the identified maintainability hotspots by conducting independent content analysis sessions.

The main **contributions** of this paper are:
• a characterization of the frequency of maintainability issues in Android apps;

[1]A "snapshot" is defined as the state of a repository after a week of active development, i.e. a week in which at least one commit occurred.

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00042

IEEE computer society

• a taxonomy and characterization of the evolution trends of the maintainability issues' density in Android apps;
• a characterization of maintainability hotspots for Android apps, together with an investigation about the Android development activities performed when those hotspots occur;
• the replication package of our empirical study containing its results, raw data, and mining- and data analysis scripts.

The **target audience** of this paper includes both Android developers and researchers. We support developers by providing a set of actionable and evidence-based insights for improving the maintainability of Android apps; we support researchers by objectively characterizing the state of the practice about maintainability of Android apps.
Paper structure. Section II provides background information. Section III presents the design of the study. Section IV presents and discusses obtained results. Sections V and VI describe threats to validity and related work, respectively. Section VII closes the paper and discusses future work.

II. Background

Software maintainability. This study relies on the software quality model defined in the ISO/IEC 25010 standard [10]. The standard defines a generic software quality model composed of the following characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. There, **software maintainability** is defined as the degree of effectiveness and efficiency with which a product can be modified to improve it, correct it or adapt it to changes in environment, and in requirements [10]. The ISO/IEC 25010 software quality model further divides software maintainability into 5 sub-characteristics: modularity (degree of change impact of one component w.r.t. others), reusability (degree to which an asset can be used to build other systems), analyzability (extent to which a software product can be analyzed, with the goal of identifying parts to be modified), modifiability (extent to which a software product can be modified without lowering its quality), and testability (extent to which a software product can be tested).
Source code quality tools. In order to minimize maintenance costs, developers can track and improve their source code quality with the help of open-source tools. For example, **Lint** [11] is an Android Studio source code scanning tool. The Lint tool provides support in finding potential bugs and optimization improvements with respect to security, performance, usability, correctness and internationalization. Another open-source solution is **FindBugs** [12]. It reports nearly 300 different bug patterns in Java and reported bugs are categorized and assigned a priority level. **SonarQube** [13] is also a prominent open-source static code analyzer which focuses on the detection of bugs, code smells and security vulnerabilities. SonarQube currently supports more than 20 programming languages including Java and offers a vast range of customization parameters for ad-hoc analyses. **PMD** [14] is a popular source code analyzer, able to find common programming issues such as empty catch blocks, unused variables etc. PMD supports a variety of different languages, among which are Java and XML, making it suitable for analyzing source code of Android applications. Furthermore, PMD provides support for identification of duplicated code in Java source files. Another freely-available tool is **CheckStyle** [15]. Although it does not identify bugs, it allows Java developers to write code that adheres to coding standards, thus increasing code readability. **JArchitect** [16] is instead a commercial static analysis tool that, in addition to code metrics, provides capabilities to inspect the quality of software architecture through dependency analyses and validation of architectural rules. **SciTools Understand** [17] is another commercial tool aimed at providing an overview of a software product through a mix of code metrics analyses and dependency visualization techniques.

In this study we use **SAT**, a toolset developed by SIG, a software consultancy company providing insights into software systems' source code quality. SAT is based on the SIG maintainability model, which defines metrics to measure ISO 25010 maintainability based on source code. SIG has used this model in a consulting practice for the past 10 years, measuring hundreds of systems and billions of lines of code. Empirical studies have shown that the metrics used by the SAT tool are correlated with the maintainability issue resolution performance of software developers [18]. As such, this model is a very good candidate for this study, also due to its compliance with the well-acknowledged ISO/IEC 25010 standard and to the full automation enabled by the SAT analysis tool. SAT allows us to automatically perform static code analysis on snapshots of multiple apps, and provides maintainability metrics for further statistical analysis.

III. Study Design

To provide objective and replicable findings, a complete replication package is available[2] with the source code of all the developed mining and analysis software, and raw data.

A. Goal and Research Questions

The **goal** of this study is to *analyze* the source code of Android mobile apps *for the purpose of* characterizing their evolution *with respect to* maintainability issues *from the viewpoint of* software developers, *in the context of* open-source apps published in the Google Play Store. The related research questions are described in the following.
RQ1 – *Which are the most recurrent types of maintainability issues in Android apps?*

Answering this research question enables developers and researchers to get a better understanding of Android-specific maintainability issues through empirical evidence, lying the groundwork for the efficient management of maintainability issues in Android apps (e.g., by taking special care of *code duplication* issues, which are the most recurrent ones).
RQ2 – *How does the density of Android maintainability issues evolve over time?*

RQ2 investigates whether the evolution of the density of maintainability issues exhibits identifiable characteristics (i.e., specific trends). It provides insights about *how* each

[2]https://github.com/S2-group/ICSME2018ReplicationPackage

type of maintainability issues tend to remain/grow/decrease in Android apps over time, potentially with a negative or positive impact on the overall maintainability of the app in future releases. The trends emerging from this study can guide developers in classifying their own apps, comparing them with others, and taking action depending on the level of maintainability they want to achieve. For example, if a team of developers recognizes that their activities are falling in the *stable increase* pattern, then they can counteract it by planning refactoring-specific sessions.

RQ3 – *What are the development activities in which maintainability hotspots occur?*

RQ3 aims at identifying the relation between the activities performed by developers and the occurrence of maintainability hotspots. The identification of those relations helps in understanding what are the Android development activities that are more sensible to the injection of each type of maintainability issues. Android developers can use this information for (i) better planning code refactoring sessions, (ii) better planning their code review sessions (e.g., steering the assignment of code reviews), (iii) taking special care of their code quality when performing tasks belonging to activities highly correlated with maintainability issues.

B. Context and Dataset

1) Context selection: Since this study is focused on *real-world Android apps for which we can track their maintainability and development activities over time*, the **context** of this study consists of a set of Android apps that (i) are freely distributed in the Google Play store and (ii) have their versioning history hosted on GitHub.

2) Dataset building: The dataset building process of this study is similar to the one proposed in [19]. As shown in Figure 1, we consider the following initial sources: GitHub, FDroid, and Wikipedia. From Github, a custom search was performed that targeted all the repositories containing a link to a Google Play Store app page in their readme files. The second source for our dataset is FDroid [20], a largely known online catalogue of free and open-source Android projects. From this catalogue, a search was applied that locates apps that contain: a) a link to the respective GitHub repository, and b) a link to the respective Google Play store page. The third source is a Wikipedia [21] containing a maintained list of free and open-source Android applications. We performed a manual selection from this list. The merging step (1) of the three considered data sources resulted in a total of 9,400 apps.

Some of the apps were not published on the Google Play store, and were therefore excluded (2). This occurs if developers decide to remove the app from the store or if Google takes down apps for violating some publishing policies. Next, duplicate entries have been removed (3).

In the next filtering step we identified repositories containing actual Android app source code (4). This filtering step has been done by considering only the repositories containing the mandatory AndroidManifest.xml file.

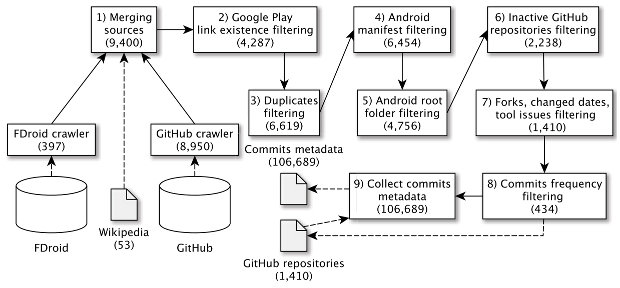
1) Merging sources (9,400)
2) Google Play link existence filtering (4,287)
3) Duplicates filtering (6,619)
4) Android manifest filtering (6,454)
5) Android root folder filtering (4,756)
6) Inactive GitHub repositories filtering (2,238)
7) Forks, changed dates, tool issues filtering (1,410)
8) Commits frequency filtering (434)
9) Collect commits metadata (106,689)
FDroid crawler (397)
GitHub crawler (8,950)
Commits metadata (106,689)
FDroid
Wikipedia (53)
GitHub
GitHub repositories (1,410)

Fig. 1: The dataset building process

Then, we filtered out repositories which did not contain an Android manifest file in the source code folder (5). The rationale for this step is that the folder containing the Android manifest file should also contain the complete source code for each application.

In order to avoid inactive or unmaintained repositories [22], we considered only repositories with at least 6 commits and having a lifetime span of at least 8 weeks (6). The 6-commits threshold corresponds to the median of the number of commits for all considered repositories before this filtering step, while the 8-weeks threshold comes from the fact that 8 weeks is the average development time for an Android app [23].

We further cleaned up the dataset by filtering out all those GitHub repositories containing (i) commit dates which were manually modified by developers (our study has a strong emphasis on the time dimension), (ii) forks of other repositories (to avoid duplicates), and (iii) source code not analyzable by the SAT tool, e.g., Kotlin-based apps (7).

The last filtering step involved (i) the identification and removal of all the snapshots for which there were no commits in the GitHub repositories and (ii) the subsequent filtering of all the GitHub repositories having less than 8 snapshots after the snapshots removal (8). After this step, our final dataset contains **434** GitHub repositories containing open, published, and actively maintained Android apps, for which an analyzable commit history is available.

Finally, we extracted all commits of the main branch of each GitHub repository, leading to a total of **106,689** commits (9).

3) Demographics: In the following we provide an overview of the apps included in our dataset.

As shown in Figure 2, the median app of the dataset results to be developed for 16 **active weeks of development** (hereafter, snapshots).

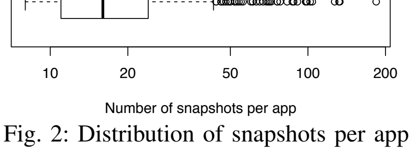

Fig. 2: Distribution of snapshots per app

As we may expect, the **number of commits per app** is characterized by a high variance, ranging from a minimum of

12 to a maximum of 2,407 commits per app. The median app of the dataset has 123 commits (see Figure 3).

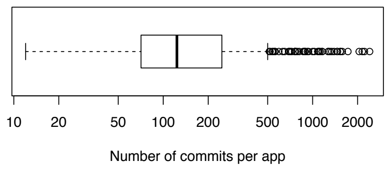

Fig. 3: Distribution of commits per app

The **number of commits per snapshot** varies from a minimum of 1 commit (which was required in order to characterize a snapshot as active) to a maximum of 251 commits. The median snapshot is composed of 6 commits (see Figure 4).

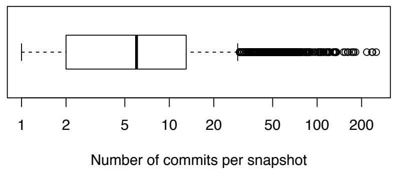

Fig. 4: Distribution of commits per snapshot

Regarding the **period of development**, we ensured that the apps within our dataset are heterogeneous, in order to do not bias our study due to some specific versions of the Android platform. In particular, the earlier app development start date is close to the initial release of the Google Play market (end of 2008) till early 2017 (close to when our crawling process was executed, see also Section III-B). In Figure 5 the development start date of all the apps is reported.

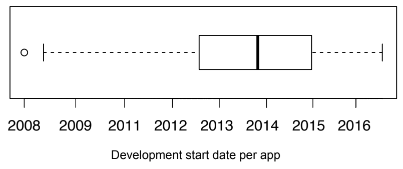

Fig. 5: Distribution of development start date per app

Finally, by considering the **number of unique contributors** per app, the median shows that apps developed by 3 unique contributors result to be the most common in our dataset. In Figure 6 the distribution of number of unique contributors per app is depicted.

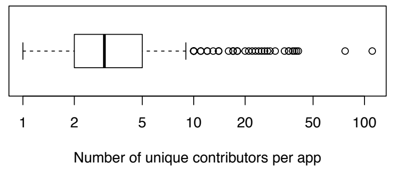

Fig. 6: Distribution of unique contributors per app

C. Data Extraction

Snapshots Extraction. This study considers the evolution of maintainability issues of an Android app as a sequence of snapshots $(S_1, S_2, ..., S_n)$, where a snapshot is defined as a set of source code files of an app at a given point in time. A time-windowing approach has been adopted and closely follows the approach presented in the work by Di Penta et al.[9]. A snapshot series can be extracted from an app's GitHub repository by considering all the commits performed between the begin and end of a snapshot interval. For this research, the time interval between two snapshots in a snapshot series is defined as one week, mainly because it has been empirically shown that Android apps are updated once a week or less frequently [24]. Once the snapshot series has been obtained, each app repository has been cloned and subsequently checked out for each snapshot in the series.

In total, the whole analysis process of extracting snapshots and applying static code analysis to each snapshot spanned a period of 2 months. Furthermore, a total of 800 million LOCs have been processed, and more than 7,000 GB of file system resources have been considered across 9,945 snapshots.

Maintainability Issue Density Identification. In this step we used the SAT to process every snapshot of each app, producing the source code metrics related to maintainability issues of Android apps. The total static code analysis processing of all snapshots took 12.45 days, with an average execution time of 8.73 seconds per snapshot. As mentioned before, the metrics in SAT are based on the ISO/IEC 25010 quality model. The model underlying SAT measures the properties of a software product at four levels of abstraction, namely: unit, module, component and overall system level. In the context of this study, **unit** is a Java method, a **module** is a Java class, a **component** is a Java package, and a **system** is the whole app.

In this study, we consider the maintainability issues defined by SAT ([7]), namely:

- **Unit Size (US):** Methods exceeding 30 lines of code[3].
- **Unit Complexity (UC):** Methods whose McCabe Cyclomatic Complexity exceeds 10 [28].
- **Unit Interfacing (UI):** Methods having more than 4 (formal) parameters.
- **Module Coupling (MC):** Classes exceeding 20 incoming dependencies (eg. method invocations, class extensions, interface implementations).
- **Duplication (DP):** Code clones of at least 6 lines of code. SAT detects type-1 clones [29] after a cleaning process that removes empty lines and comments.
- **Maintainability (MT):** The total count of occurrences of the previous issues types.

Once the maintainability issues have been identified, a unified and comparable measure of their amount is needed, in order to allow for objective comparisons between apps with different size and between different issue categories. Therefore, in this study we consider a notion of commit-level *issue density (cd)* [9], defined as follows:

$$cd^i_{c_a} = |\bigcup_{x \in I^i_{c_a}} x| / NKLOC_{c_a} \tag{1}$$

where i uniquely identifies one of the types of maintainability issues according to the ISO/IEC 25010 quality model (e.g.,

[3]The thresholds for maintainability issues have been defined based on the results of empirical studies involving the SAT tool [25], [26], [27].

unit complexity), a is the app being considered, $c_a \in C_a$ is a commit in the GitHub repository of a, $I^i_{c_a}$ is the set of identified issues of type i in the repository of a after checking it out at commit c, and $NKLOC_{a,c}$ is the number of thousands of lines of Java source code in the repository of a after checking it out at commit c. Intuitively, cd indicates the number of issues of a specific type that are introduced by a commit, normalized by considering the current size of the repository. Being S_a the set of all weekly snapshots of a, the issue density $d^i_{s_a}$ of each $s_a \in S_a$ is defined as $cd^i_{c_a}$, where c is the last commit performed within the time window of s_a.

D. Data Analysis

1) RQ1: In order to test whether the issue types exhibit a significant difference w.r.t. issue density, we adopt the omnibus Kruskal-Wallis test, i.e., a non-parametric test for testing the difference among multiple medians. In order to avoid potential threats to validity due to fishing rate we adjust the significance level by means of the Bonferroni correction [30]. In addition to the Kruskal-Wallis omnibus test, in order to assess if there is a significant difference between the pair-wise comparison of issue types, we conduct a series of two-tailed Mann-Whitney tests. This is not required in order to test the *null* hypothesis, i.e. that all means of the issue density among issue types is equal. Nevertheless we use this statistical tests to get further insights in the data. As the previous test, the Bonferroni correction is used to adjust the significance level.

Apps characterized by long-lasting active development might affect the results due to their high number of snapshots present in the dataset. In order to mitigate this potential threat to internal validity, the above presented tests are carried out both on the complete dataset of snapshots and on the median values of the snapshots aggregated per app. Due to space limitations, and as the results do not significantly differ, in the result section we report the results relative to the analysis of the complete dataset.

2) RQ2: For each app a and maintainability issue i, we firstly create a time series representation TS^i_a by temporally ordering all the density values d^i_s across all snapshots $s \in S_a$ of a. The time of the first and last observations of each TS^i_a are set to the timestamp of the first commit among the set of all commits C_a of a and the timestamp of the end of the time window identified by the last snapshot in S_a, respectively.

As initial exploration, we check if the obtained time series exhibit a stationary behaviour. From a statistical perspective, the mean and variance of a stationary time series are constant over time [31] (i.e., it has no trend over time). We apply to each TS^i_a the Augmented Dickey-Fuller test (ADF) [32] (with $\alpha = 0.05$), which is a unit root test for stationarity with H_0 = *the time series has a unit root (it is non-stationary)* and H_1 = *the time series does not have a unit root (i.e., it is stationary).* We adjust the obtained p-values via the Bonferroni correction since we are applying the ADF test 6 times, one for each type of maintainability issue. From this preliminary test, apps result to be mostly non-stationarity for all types of maintainability issues, motivating us to further inspect their exhibited trends.

We decompose each TS^i_a into its seasonal, trend and irregular components [33] using the STL algorithm [34]. Intuitively, given a time series, STL iteratively extracts its seasonal component by loess smoothing the seasonal sub-series (e.g., the series of the first values of all seasons, of the second values of all seasons, etc.). Then, the seasonal values are removed, and the trend component is extracted by smoothing the remainder. The irregular component is computed as the residuals from the seasonal plus trend fit [34]. We use the STL algorithm as it does not assume any distribution of the time series, it has been successfully used in previous software engineering studies [30], [35], and an efficient implementation is available as an open-source R package[4].

For answering RQ2, we perform a qualitative study on the plots of the trend components of all TS^i_a. Since the manual analysis of all the collected trend components is infeasible, we randomly selected a sample composed of 205 apps and analyzed their trend component for each type of maintainability issues[5] (summing up to 1,230 distinct trends), and manually scrutinize and categorize them into relevant groups by applying the open card sorting technique [36]. To minimize bias, two researchers have been involved in this activity and the results have been checked by a third researcher. This activity resulted in the taxonomy of maintainability issues presented in Section IV-B.

3) RQ3: We answered RQ3 by following two main phases. The first phase targets the *identification of maintainability hotspots* along the lifetime of an Android app. Given an app a and its density time series TS^i_a (one for each type of maintainability issue), the set of maintainability hotspots H^i_a is defined as follows:

$$H^i_a = \{s_j \mid (d^i_{s_j} - d^i_{s_{(j-1)}}) > \sigma(TS^i_a), \\ j = 2, \dots, |TS^i_a|, s_j \in TS^i_a\}$$

where $d^i_{s_j}$ is the density of the maintainability issue of type i in snapshot s_j and σ is the standard deviation of the density values along the time series TS^i_a. In other words, we consider as hotspot every snapshot s_j in which the density of a maintainability issue i w.r.t. its preceding snapshot $s_{(j-1)}$ is higher than the standard deviation of the whole time series of i. We build upon this standard-deviation-based strategy since (i) it is not feasible to build upon more advanced models for outliers detection (e.g., ARIMA [37]) for all 434 apps since they require delicate manual tuning for each app and (ii) it is computationally efficient. Despite its apparent simplicity, an independent manual exploration of a subset of the apps by two researchers confirms that this strategy proves effective in correctly identifying maintainability hotspots. This phase led to the identification of 3,374 maintainability hotspots over 46,873 GitHub commits.

In the second phase, we consider GitHub commit messages as indicators of the actual activities performed by developers

[4]http://stat.ethz.ch/R-manual/R-devel/library/stats/html/stl.html

[5]By considering 205 apps we achieve a 95% statistically significant sample of the 434 apps of our dataset with a 0.05 confidence interval.

and we *manually* analyze the commits performed during the occurrence of maintainability hotspots. As manually analyzing all 46,873 GitHub commits is unfeasible, we build a representative sample of commits for each type of maintainability issue (95% confidence level, 0.05 confidence interval), leading to a set of 2,112 unique commits to be manually analyzed. To this aim, we conduct content analysis sessions [38] on all 2,112 commit messages and categorize them according to the taxonomy of self-reported activities of Android developers proposed and empirically validated by Pascarella et al. [39].

For answering RQ3, we present (i) the frequency and distribution of maintainability hotspots within all Android apps and (ii) how frequently each category of Android developers' activities appears in maintainability hotspots.Finally, we reuse the publicly-available dataset of 5,000 manually categorized commits produced in [39] as ground truth for all commits i.e., commits either belonging to maintainability hotspots or not. We use a Chi-Square test to assess the relationship between the activities performed in all commits and those performed in commits belonging to maintainability hotspots and Cramer's V to establish the effect size [30].

IV. Results

A. RQ1. Which are the most recurrent types of maintainability issues in Android apps?

Figure 7 shows that on average almost 18 issues, spread out throughout different issue types, are present every NKLOC (M=15.65, m=17.87)[6]. The aggregated maintainability issue distribution is reported in the leftmost violin plot of Figure 7.

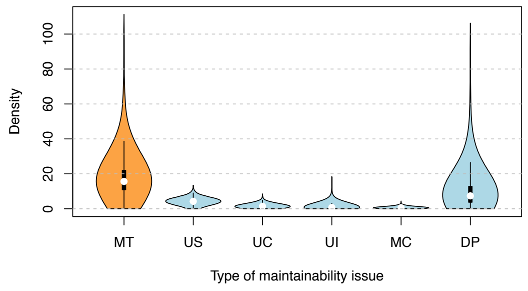

Fig. 7: Distribution of unique contributors per app

From the remaining violin plots we observe that the issue density varies among the different issue types. From the Kruskal-Wallis omnibus test we evince that the distributions significantly differ (*p-value* $< 2.2 \cdot 10^{-16}$). We hence reject the *null* hypothesis, i.e. that all means of the issue density among issue types is equal. From an additional pair-wise comparison between issue types we see that all issue types occur with different rates. The most recurring maintainability issue results to be *duplication* (M=7.393, m=10.230), followed by *unit size* (M=4.290, m=4.249), *unit complexity* (M=1.4510, m=1.6150), *unit interfacing* (M=0.7736, m=1.1030), and *module coupling* (M=0.6258, m=0.6746). Apart from *duplication* and to a certain extent *unit size* violations, the remaining issue types present only small differences in distribution.

[6]Where M is the median value and m the mean value.

Not all code duplicates are bad [40]. We conjecture that the higher frequency of *code duplication* issues is primarily due to the Android programming idiom and code duplication in Android can be related to the templating phenomenon due to the activity-intent-based nature of the Android programming model. Nevertheless, idiom-based templating can lead to introducing bugs (if not carefully used) and overlooking inconsistencies [40], which might be remarkably detrimental for the maintainability of mobile apps.

Different types of maintainability issues occur with different rates. *Code duplication* is the most recurrent maintainability issue in Android apps, followed by *unit size violations*, *unit complexity violations*, *unit interfacing violations* and *module coupling violations*. In our dataset, the overall maintainability of the analyzed apps is highly impacted by *code duplication* issues.

B. RQ2. How does the density of Android maintainability issues evolve over time?

Firstly, we assess if the density of maintainability issues exhibits a stationary trend in time. We applied the ADF test for obtaining the number of stationary and non-stationary apps for each type of maintainability issue. As shown in Table I, the ADF test reveals that 93.01% of all apps exhibit a non-stationary behaviour (*p-value* < 0.008 because of the Bonferroni correction), whereas only 6.99% are stationary in at least one type of maintainability issue.

TABLE I: Number of stationary and non-stationary trends per maintainability issue type

Maintainability issue	Stationary	Non-stationary
MT	**1**	**433**
US	4	430
UI	61	373
MC	67	367
UC	37	397
DP	12	422
TOTAL	182 (6.99%)	2,422 (93.01%)

This finding provides evidence that, according to our dataset, *the density of maintainability issues in Android apps is not stable over time*. This means that Android developers actually have an impact on the overall maintainability of their apps over time. Developers can use the instantaneous value of $d^i_{s_a}$ for keeping the maintainability of apps under control and taking informed decisions when planning for maintainability-related activities (e.g., refactoring sessions).

The vast majority of apps do not exhibit a stationary behaviour across all types of maintainability issues.

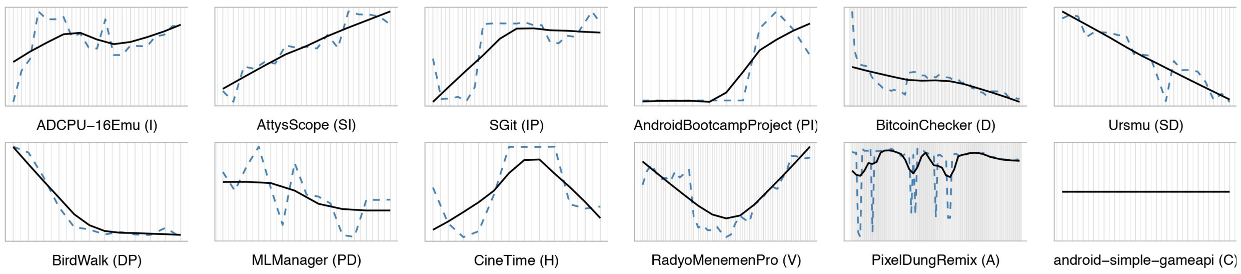

Fig. 8: Examples of evolution trends of the density of maintainability issues of Android apps

As the majority of apps does not exhibit stationarity, we further inspect their trends as detailed in Section III-D2. The manual analysis of the 1,230 trend components led to the elicitation of the taxonomy presented in Table II. The taxonomy objectively represents the categories in which the density of maintainability issues of Android apps can evolve. The taxonomy is composed of two levels, where the first one represents trends with similar overall characteristics (e.g., density growth or reduction), whereas the second level represents trends at a finer grain (e.g., stable increase, valley). Figure 8 shows an example of each trend category of the taxonomy.

TABLE II: Taxonomy of evolution trends

Trend category	Description
Growth (G)	In the first weeks the app has a low density of maintainability issues, followed by an overall non-decreasing trend
Increase (I)	The density follows a generally increasing trend (some minimal decreasing parts could be present)
Stable Increase (SI)	The density is relatively low in the initial weeks, then it gradually increases throughout the whole lifetime of the app
Increasing Plateau (IP)	The density reaches plateau(s) after an overall non-decreasing trend
Plateau Increasing (PI)	The density is stable and relatively low in the first weeks, then it increases for the whole lifetime of the app
Reduction (R)	In the first weeks the app has a high density of maintainability issues, followed by an overall non-increasing trend
Decrease (D)	The dual of I
Stable Decrease SD)	The dual of SI
Decreasing Plateau (DP)	The dual of IP
Plateau Decreasing (PD)	The dual of PI
Mixed (M)	Mixed trend where the density of maintainability issues grows and declines over time
Hill (H)	The density is relatively low in the initial weeks, it gradually increases up to a certain peak, and then it gradually decreases
Valley (V)	The dual of H
Anomalous (A)	It does not fall into any of the previously defined categories
Constant (C)	The density of maintainability issues is the same over time
Constant (C)	Same as C

In order to interpret the manually gathered results, we represent the frequency of each trend across issue types in the heatmap in Figure 9. We evince that the trend category G (composed of I, SI, IP, and PI) is the most recurrent. For the majority of maintainability categories new issues are introduced over time in the apps by following different increasing trends. Only seldom are maintainability issues resolved.

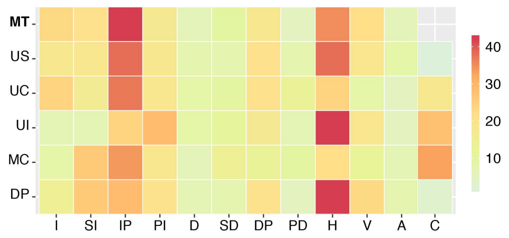

Fig. 9: Evolution trends by maintainability issue types

Of the subcategories composing G, IP is the most recurring one. We conjecture that this is the result of the introduction of maintainability issues in the early stages of development, followed by a “saturation period”. This later period of the IP trend is attributed to a higher awareness that is given by developers to maintainability issues after reaching a certain level of technical debt. From this recurrent trend we observe that developers, while not actively resolving issues, avoid to introduce more in the more mature stages of the app, potentially before maintainability issues become unmanageable in number. Issues characterized by an initial period of stability before increasing (PI) are less frequent: if maintainability issues are introduced, usually it starts from the early development stages.

The only exceptions opposing the higher occurrence of the G trend can be observed for the issues of type UI and DP, which exhibit more frequently an H trend. This can be caused by the nature of such type of issues. In fact, UI issues manifest themselves as Java methods become more complex, directly impact development time required to use them, and can be easily spotted by developers. Hence, after a period of initial growth, effort might be spent to resolve such issues and avoid cumbersome development activities in the future. A similar reasoning can be applied to the H trend of DP issues: while due to time constrains DP issues can be acceptable at initial stages of development, in time resolving such apparent issues can be a valuable activity which eases future development.

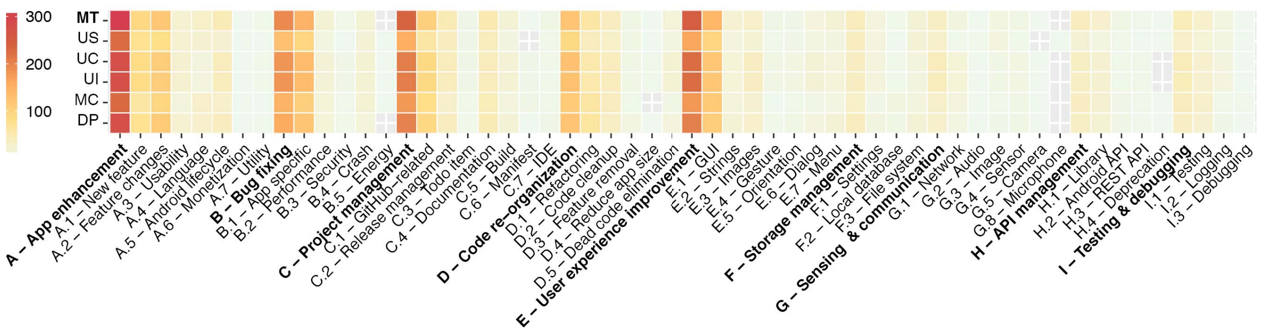

Fig. 10: Frequencies of Android development activities performed during a maintainability hotspot

From the lower frequency of R trends (D, SD, DP, PD) we evince that only seldom development activities lead to a consistent decrease of maintainability issues. This may be attributed to scarce effort put into structured refactoring processes aimed to improve and maintain apps' software quality.

Constant trends also result be less occurrent. We can conjecture that this is due to the selection criteria adopted to systematically filter the weeks of development considered in our dataset. In fact we selected exclusively active weeks of development, i.e. when changes were carried out on the apps, which reflected in a changing number of issues.

The scarce occurrences of anomalous trends (A) shows that our taxonomy proved to be effective, and encompassed the vast majority of the trends of maintainability issues analyzed.

> Maintainability issues in Android apps exhibit mainly a *growth* in time. Reaching a plateau after an initial increase is a particularly recurrent trend. *Unit interfacing* and *code duplication* issues result to be outliers, displaying a *hill* like trend, which can be motivated by the nature of these issues. Only seldom maintainability of apps exhibits an overall *decrease* or *constant* trends in time.

C. RQ3. What are the development activities in which maintainability hotspots occur?

As anticipated in Section III-D3, we identified a total of 3,374 maintainability hotspots involving 46,873 GitHub commits. Their descriptive statistics are reported in Table III.

Overall, the number of maintainability hotspots per app ranges between 0 and 8, with a median of 1 (similar values for the mean and standard deviation). The low values for standard deviations also tell us that the distribution of the number of hotspots across apps is very compact, with a strong tendency towards one hotspot per app.

> Maintainability hotspots do not occur often in the lifetime of each app. Nevertheless, on average each app has at least one maintainability hotspot.

Now we zoom into the activities performed by developers during the occurrence of maintainability hotspots. To do so, we manually categorized 2,112 commit messages according to a taxonomy of self-reported Android development activities [39]. The taxonomy entails a wide variety of different activities at different levels of abstraction (e.g., bug fixes, functionality implementation, release management, access to sensors, etc.). The taxonomy is composed of two levels, where the first layer groups together activities with similar overall purpose (e.g., app enhancement, bug fixing, API management), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization [39]. Figure 10 shows the frequency of each category of Android developers' activities across the various types of maintainability hotspots[7]. The colour of each tile reports the sum of all occurrences of each development activity in correspondence of a hotspot of a specific type of maintainability issue (note that multiple activities can be assigned to each hotspot). The order of development activities follows the rank of most frequent activities in Android apps, as emerged in [39] (e.g., group A is more frequent than group B, activity A.1 is more frequent than activity A.2).

TABLE III: Descriptive statistics for the hotspots of each maintainability issue type per app (SD = standard deviation, CV = coefficient of variation)

Issue type	**Min.**	**Max.**	**Median**	**Mean**	**SD**	**CV**
MT	0	6	1.0	1.38	1.03	0.75
US	0	6	1.0	1.35	1.03	0.76
UC	0	8	1.0	1.16	1.09	0.94
UI	0	8	1.0	1.16	1.09	0.94
MC	0	7	1.0	1.31	1.16	0.88
DP	0	6	1.0	1.41	1.12	0.79

Overall, our categorization follows the same trends identified in [39], with the most prominent categories of activities being *app enhancement* (specially for new and updated features), followed by (app specific) *bug fixing*, and *project management*. Interestingly, the category *user experience improvement* is quite frequent in the presence of hotspots

[7]In Figure 10 we depict in bold the top level categories and in plain text all the subcategories of the taxonomy.

(+7.28% more than it is for "standard" commits, i.e., commits unrelated to hotspots), specially for those activities related to the graphical user interface (E.1 - GUI). This result can be an indication that Android developers should pay special attention when working on the business logic related to buttons, UI layouts, event listeners, etc., as those activities are potentially more related to the presence of maintainability hotspots. Moreover, we noticed an increase also for the category *Android lifecycle* (+2.86% w.r.t. all commits), which refers to the activities about the management of Android components lifecycle events and transitions (e.g., the onCreate of Activity). Finally, also the *Documentation* category exhibits an increase in frequency when maintainability hotspots occur (+ 3.23%); this kind of activities refers to adding/refining comments in the source code and working on the documentation of the app (e.g., description of the app, its requirements, UI mockups). This result happened for 85 commits and was quite unexpected as in principle this category should not be related to maintainability at all. A deeper investigation reveals that all snapshots containing the 85 documentation-related commits include also other commits, which may be related to other development activities playing a role in the occurrence of hotspots. Nevertheless, the co-occurrence of documentation-related activities and maintainability hotspots may be an indication that commits chronologically close to documentation-oriented activities are correlated with hotspots. We leave this analysis for future work.

In order to better characterize how Android development activities may be correlated with maintainability hotspots, we test if there is a significant difference between the frequencies in our categorization of 2,112 commits and the ones observed in the 5,000 manually categorized commits in [39]. It is important to note that the commits categorized in [39] have been randomly selected, they can belong either to maintainability hotspots or not. For all maintainability issue types we obtained a statistically significant measure of correlation between these two categorizations (*p-value* $< \sigma$, where $\sigma = 0.008$ because of the Bonferroni correction), allowing us to reject the null hypothesis that the two categorizations are independent. The Cramer's V test reveals a small effect size for all maintainability issue types ($0.19 \leq V \geq 0.22$). Together, those results (i) confirm our preliminary exploration that the frequencies of Android development activities in commits belonging to maintainability hotspots are in line with those involving all commits and (ii) such a relationship is only weak.

Maintainability hotspots in Android apps tend to occur independently of the type of development activities performed by developers. Activities related to the *GUI* and the management of the *Android lifecycle* are slightly more prone to co-occur with maintainability hotspots.

V. Threats to Validity

Construct validity. The results of this study are based on the current implementation of SAT, the used static code analysis tool. It is hence paramount that the tool was implemented and configured correctly. This major threat was mitigated through different strategies. Firstly, the tool documentation is made available in order to detail the maintainability issue detection processes and the configuration settings adopted. Furthermore, interviews have been conducted with the tool developers to investigate the details related to the identification of the identified maintainability issues. Additionally, an inspection of several detected issues across issue categories was performed manually. Finally, the SAT tool is utilized on a daily basis in industrial settings, and was also utilized in previous researches carried out by independent researchers [41], [42], [43].

An additional threat to construct validity is constituted by the representativeness of the selected apps. This threat was mitigated by carrying out an in-depth data quality assurance process (reported in Section III-B2). In addition, we ensured that the data was encompassing and heterogeneous in terms of development lifespan, number of commits, and number of contributors etc.

Conclusion validity. The most prominent threat to conclusion validity is constituted by the data extraction and analysis processes adopted to gather the results. In order to mitigate it, we strictly adhered to a set of *a priori* defined data extraction and analysis processes. Such processes were explicitly conceived to gather and analyze significant data to answer our research questions. In addition, a replication package with the raw data and analysis scripts is made available for the complete reproducibility of the results.

The majority of the statistical tests produced p-values far below the chosen significance level of 0.05. To minimize the error rate of the results, the Bonferroni correction was adopted to adjust the significance level, when required. In order to further mitigate potential threats to conclusion validity, the data analysis process was jointly discussed by the researchers and the results were inspected independently. The level of agreement, especially for the manual labelling processes, was assessed statistically. Disagreements were jointly discussed to scrupulously align the data extraction and analysis processes and ensure a high quality level of the gathered results.

Internal validity. As repositories containing the app source code differ in structure, it is possible to obtain false results with the inclusion of non-app related source code (e.g., third party libraries). This threat has been mitigated both before and during the static code analysis. Firstly, the root app folder containing the app source code has been identified for each repository, and the code metrics have been collected only for the source code contained within this folder. Also, the SAT tool allows the selection of different files during static code analysis, and this was exploited in the sense that .jar files and library directories have been excluded from the analysis.

Another threat relates to the exclusion of component-related maintainability metrics from the measurements. In order to ensure the envisioned quality of the analyzed data, an in-depth data quality assurance process was carried out (see Section III-B2). All instances presenting inconsistencies led to the total exclusion of the entire app data from which the instance belonged to. In this way, we were able to strictly control the quality of the analyzed data by including exclusively the apps of which every piece of data adhered to our quality criteria.

External validity. In this study we consider a set of Android apps sampled from a real-world setting. This was possible by considering exclusively those apps which are published in the Google Play Store. In order to further mitigate potential threats to external validity, we ensured that the apps considered were representative of the apps present in the Android ecosystem. From an inspection of the gathered dataset the apps resulted to be highly heterogeneous in terms of size, development lifetime, number of contributors etc. (see section III-B3).

This research considers Android apps published in GitHub since we require access to their full versioning history. We do not target app binaries in the Google Play store as it only provides the latest release of each app. Further, we are interested in the maintainability issues introduced by developers in the Java code of their apps; in Google Play only the binary code of the app is available, which may be structurally different from the source code produced by developers (e.g., because of code obfuscation). Nevertheless, due to the high heterogeneity of the dataset (see Section III-B3) and the presence of all the considered apps in the Google Play Store, we do not deem this as a major threat to external validity.

VI. Related Work

The state of the art on the maintainability evolution of Android apps is quite scarce, yet it exhibits some related work on Android source code quality and software evolution.

Hecht et al. [24] presented an approach for performing static code analysis on Android app's bytecode and detecting software antipatterns. They analyzed the evolution of the quality across 3,568 versions of 106 different Android apps obtained from the Google Play Store. They identified relationships between antipatterns and five different quality evolution trends. This work ties into our research with a similar methodology and focus on quality aspects and their evolution in the context of mobile (Android) apps. Differently, we focus on maintainability-related issues (rather than software antipatterns) and on how development activities are related to them. Our research scope involves the analysis of over 400 Android apps, compared to the 106 analyzed in [24].

Di Penta et al. [9] analyzed the evolution trends of statically detectable vulnerabilities of software projects. For the detection of vulnerable source code lines, they have used 3 different static code analysis tools, namely Splint, Rats and Pixy. Three different networking systems were analyzed by means of executing the static code analysis tools on different snapshots of the system. This study is methodologically similar to ours. However, the subjects and therefore the outcomes of their study differ from ours, as we are specifically focussing on Android apps and their maintainability, as opposed to vulnerabilities in source code of generic software systems.

Tufano et al. [44] investigated on *when* and *why* code smells are introduced in a software project. Their study involves investigating the circumstances and rationales behind bad code smells introduction, and is conducted on a change history of 200 open-source projects. Our research is specific to Android apps, and thus our results are more fine-grained with respect to the ones obtained in [44]. The focus of our study is on maintainability-related issues at a higher level of abstraction (i.e., units, models, and components) w.r.t. Tufano et al. who focus on fine-grained code smells at the level of source code.

Similar to our research, Koch [45] set out to analyze the evolution of open-source software systems on a large scale. Utilizing the data of 8,621 projects coming from SourceForge, the evolutionary behaviour of the systems was characterized by applying both linear and quadratic models to the systems, where the quadratic model outperformed the linear one. Furthermore, the evolutionary behaviour has been modelled as a function of lines of code and time since the first commit. Both Koch's and our study focus on large-scale, open-source systems. However, we focus on Android apps and Android-specific development activities.

VII. Conclusion and Future Work

This study uncovers the frequency and evolution of maintainability issues of Android apps. Its results show that *code duplication* is the most recurrent maintainability issue (RQ1), which is intrinsic in the Android programming model and can be mitigated by a more careful programming style. In general, notwithstanding the issue type, maintainability issue density grows until it stabilizes, but issues are seldom fully resolved, which represents an important hidden lack of quality. Also, maintainability hotspots are independent from the type of development activity (RQ3), which means: *whatever you do, your development style will matter*.

Finally, during dataset building we faced many challenges related to the data obtained from GitHub (e.g., manually-changed commit dates, repositories with long periods of inactivity, etc.): to ensure a high-quality dataset we had to remove many data points, going from 9,400 to 434 repositories. As also confirmed in [22], this is an important reflection point for improving how researchers approach the data in GitHub.

As future work, we will investigate the survivability of maintainability issues in Android apps, specifically on the issues duration, the nature and time of the commits either introducing or solving them, and the activities performed in those commits. This will require an in-depth analysis of the commit logs in order to trace, for each issue, its inducing commits, its changes of location within the repository, and its issue-resolution commit. Moreover, we are planning to investigate on how the maintainability issue evolution trends compare between different static code analysis tools and on a larger dataset of GitHub repositories such as the one in [46]. Finally, we will select a subset of representative apps, build and fine tune prediction models for their maintainability issues (e.g., by using ARIMA), and assess the accuracy of those models in predicting how maintainability issues evolve in the future.

References

[1] "Number of available applications in the google play store from december 2009 to june 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[2] Adam Lella, Andrew Lipsman, "The 2016 U.S. Mobile App Report," 2016, comsCore white paper.

[3] "Android developer portal," 2017. [Online]. Available: http://developer.android.com/about/index.html

[4] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[5] "Facebook app release history," 2017. [Online]. Available: http://www.apk4fun.com/history/2430

[6] "Android core app quality guidelines," 2017. [Online]. Available: http://developer.android.com/develop/quality-guidelines/core-app-quality.html

[7] J. Visser, "SIG/TÜViT evaluation criteria trusted product maintainability: Guidance for producers," Software Improvement Group, 9.0, February 2017. [Online]. Available: https://goo.gl/d18AsP

[8] "Sig official website," 2017. [Online]. Available: http://sig.eu

[9] M. D. Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information and Software Technology*, vol. 51, no. 10, pp. 1469 – 1484, 2009.

[10] I. 25010:2011, "Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," 2011. [Online]. Available: https://www.iso.org/standard/35733.html

[11] "Android lint official website," 2017. [Online]. Available: http://developer.android.com/studio/write/lint.html

[12] "Findbugs official website," 2017. [Online]. Available: http://findbugs.sourceforge.net

[13] "Sonarqube," 2018. [Online]. Available: https://www.sonarqube.org/

[14] "Pmd official website," 2017. [Online]. Available: http://pmd.github.io

[15] "Checkstyle official website," 2017. [Online]. Available: checkstyle.sourceforge.net

[16] "Jarchitect," 2018. [Online]. Available: https://www.jarchitect.com

[17] "Scitools," 2018. [Online]. Available: https://scitools.com

[18] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software Quality Journal*, vol. 20, no. 2, pp. 265–285, Jun. 2012.

[19] T. Das, M. D. Penta, and I. Malavolta, "A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps," in *ICSME '16 Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 443–448.

[20] "Fdroid," 2017. [Online]. Available: http://f-droid.org

[21] "Wikipedia page on open-source android apps," 2017. [Online]. Available: http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications

[22] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[23] "How long does it take to build a mobile app?" 2017. [Online]. Available: http://www.kinvey.com/how-long-to-build-an-app-infographic/

[24] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the Software Quality of Android Applications along their Evolution," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–436.

[25] T. Döhmen, M. Bruntink, D. Ceolin, and J. Visser, "Towards a benchmark for the maintainability evolution of industrial software systems," in *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2016 Joint Conference of the International Workshop on*. IEEE, 2016, pp. 11–21.

[26] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software quality journal*, vol. 20, no. 2, pp. 265–285, 2012.

[28] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[27] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 2007, pp. 30–39.

[29] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 2006, pp. 253–262.

[30] J. Rosenberg, "Statistical methods and measurement," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 155–184.

[31] A. I. McLeod, H. Yu, and E. Mahdi, "Time series analysis with r," in *Handbook of statistics*. Elsevier, 2012, vol. 30, pp. 661–712.

[32] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American statistical association*, vol. 74, no. 366a, pp. 427–431, 1979.

[33] B. L. Bowerman and R. T. O'Connell, "Forecasting and time series: An applied approach. 3rd," 1993.

[34] R. B. Cleveland, W. S. Cleveland, and I. Terpenning, "Stl: A seasonal-trend decomposition procedure based on loess," *Journal of Official Statistics*, vol. 6, no. 1, p. 3, 1990.

[35] A. Atchison, C. Berardi, N. Best, E. Stevens, and E. Linstead, "A time series analysis of travistorrent builds: to everything there is a season," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 463–466.

[36] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[37] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[38] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, 2nd ed. Rockport Publishers, January 2010.

[39] L. Pascarella, F.-X. Geiger, F. Palomba, D. D. Nucci, I. Malavolta, and A. Bacchelli, "Self-Reported Activities of Android Developers," in *5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. New York, NY: ACM, May 2018, p. to appear. [Online]. Available: http://www.ivanomalavolta.com/files/papers/mobilesoft_2018_self.pdf

[40] 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.

[41] B. J. Luijten, "The influence of software maintainability on issue handling," Master's thesis, Faculty EEMCS, Delft University of Technology, 2009.

[42] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6862882/

[43] T. L. Alves, "Determination of number of clusters in K-means clustering and application in colour segmentation," in *Proceedings of Simulation and EGSE facilities for Space Programmes (SESP2010)*, 2010.

[44] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *ICSE '15 Proceedings of the 37th International Conference on Software Engineering*, I. P. Piscataway, Ed. IEEE, 2015, pp. 403–414.

[45] S. Koch, "Software evolution in open source projects - a large-scale investigation," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 6, pp. 361–382, 2007. [Online]. Available: http://dx.doi.org/10.1002/smr.348

[46] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. D. Nucci, I. Malavolta, and A. Bacchelli, "A Graph-based Dataset of Commit History of Real-World Android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR*. New York, NY: ACM, 2018.