Do Energy-oriented Changes Hinder Maintainability?

Luis Cruz*, Rui Abreu†, John Grundy‡, Li Li‡ and Xin Xia‡
* INESC-TEC, University of Porto, Porto, Portugal
Email: luiscruz@fe.up.pt
† INESC-ID, University of Lisbon, Lisbon, Portugal
Email: rui@computer.org
‡ Faculty of Information Technology, Monash University, Melbourne, Australia
Email: {john.grundy, li.li, xin.xia}@monash.edu

***Abstract*—Energy efficiency is a crucial quality requirement for mobile applications. However, improving energy efficiency is far from trivial as developers lack the knowledge and tools to aid in this activity. In this paper we study the impact of changes to improve energy efficiency on the maintainability of Android applications. Using a dataset containing 539 energy efficiency-oriented commits, we measure maintainability – as computed by the Software Improvement Group's web-based source code analysis service *Better Code Hub* (BCH) – before and after energy efficiency-related code changes. Results show that in general improving energy efficiency comes with a significant decrease in maintainability. This is particularly evident in code changes to accommodate the *Power Save Mode* and *Wakelock Addition* energy patterns. In addition, we perform manual analysis to assess how real examples of energy-oriented changes affect maintainability. Our results help mobile app developers to 1) avoid common maintainability issues when improving the energy efficiency of their apps; and 2) adopt development processes to build maintainable and energy-efficient code. We also support researchers by identifying challenges in mobile app development that still need to be addressed.**

***Index Terms*—Energy Consumption, Software Maintenance, Mobile Computing**

I. Introduction

Modern mobile applications, popularly known as apps, provide users with a number of features in multi-purpose mobile computing devices – smartphones. The convenience of using smartphones to pervasively accomplish important daily tasks has a big limitation: smartphones have a limited battery life. Apps that drain battery life of smartphones can ruin user experience, and are likely to be uninstalled unless they offer a key feature.

Thus, it is critically important that apps efficiently use the battery of smartphones. However, many developers still lack knowledge about best practices to deliver energy efficient mobile applications [1], [2]. Important efforts have been carried out to help developers ship energy efficient mobile apps [3]. Novel tools have been built to suggest energy improvements to the codebases of mobile apps [4]–[7] and to help developers measure the energy consumption of their apps [8]–[12].

Despite these efforts, improving the energy efficiency of mobile applications is not a trivial task. It requires implementing new features and refactoring existing ones [13], only for the sake of better energy usage, i.e., predominantly a non-functional rather than functional change. However, the extent to which these changes affect the maintainability of the mobile app software has not yet been studied. In this work, we are interested in studying the trade-off between the energy efficiency and the maintainability of mobile applications.

The International Standards on software quality ISO/IEC 25010 define software maintainability as "the degree of effectiveness and efficiency with which a software product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" [14]. The standard defines five core sub-characteristics of maintainability: modularity, reusability, analyzability, modifiability, and testability. The Software Improvement Group (SIG) has developed a web-based source code analysis toolset *Better Code Hub* (BCH) [15] that maps the ISO/IEC 25010 standard on maintainability into a set of 10 guidelines, such as *write short units of code* and *write code once*, derived from static analysis [15]–[18]. The code metrics used by the SIG model were empirically validated in previous work [19]. We use this toolset in our work to provide an assessment of maintainability in mobile app codebases.

Specifically, we want to explore whether there is a trade-off between applying energy efficiency patterns and keeping the maintainability of the apps, i.e., does improving energy efficiency have a negative impact on code maintainability? In this paper, we present the results of our analysis on the maintainability using 539 energy commits harvested from open source Android applications.

The key contributions of this work are:

- An empirical investigation of the impact of energy patterns in code maintainability.
- A dataset of energy commits and respective impact on maintainability.
- A software package with all scripts used in our experiments and a dataset of energy commits with respective impact on maintainability, for reproducibility. Available here: https://figshare.com/s/989e5102ae6a8423654d.

Our empirical study finds evidence that energy efficiency-oriented code changes have a negative impact on code maintainability. In particular, careful thinking is required to implement the energy patterns *Power Save Mode* and *Wakelock Addition*. Furthermore, we show that energy patterns are more likely to require maintenance than regular code changes.

This paper is structured as follows. In Section II, we introduce an example of an energy improvement from a real-world

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00013

mobile application. Section III describes the methodology we use to answer the research questions. We present the results in Section IV and discuss their implications in Section V. In section VI, we enumerate the threats to the validity of our work. Section VII describes the differences between our work and existing literature. Finally, in Section VIII we summarize the main conclusions and elaborate on future work.

II. Motivating Example & Research Questions

Improving energy efficiency of apps revolves around changing their codebases. Previous work has studied existing energy patterns for mobile applications [13]. It cataloged typical coding practices developers adopt to address energy efficiency. An example of an energy pattern is the *Power Save Mode*: the app features a mode that can be activated upon low battery and uses fewer resources while providing the minimum functionality that is indispensable to the user.

An instance of this pattern can be found in the app *NetGuard*[1] – an Android app that provides a firewall and monitors network traffic across other apps.

To improve energy efficiency, *NetGuard*'s developers decided to implement the pattern *Power Save Mode* [13]. The following snippet presents the required code changes[2]:

```
public class SinkholeService extends VpnService {
  private boolean powersaving = false;
// [snip]

  public void handleMessage(Message msg) {
+  if (powersaving) return;❶
    switch (msg.what) {
      case MSG_PACKET:
      log((Packet) msg.obj, msg.arg1, msg.arg2 > 0);
// [snip]
    }
  }

// [snip]

  private BroadcastReceiver interactiveStateReceiver =
    new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
// [snip]
      statsHandler.sendEmptyMessage(
-      Util.isInteractive(this) ? STATS_START : STATS_STOP
+      Util.isInteractive(this) && !powersaving ?
+        STATS_START : STATS_STOP  ❷
      );
    }
  };

// [snip]

+ private BroadcastReceiver powerSaveReceiver =
+   new BroadcastReceiver() { ❸
+   @Override
+   @TargetApi(Build.VERSION_CODES.LOLLIPOP) ❹
+   public void onReceive(Context context, Intent intent) {
+     Log.i(TAG, "Received " + intent);
+     Util.logExtras(intent);
+     PowerManager pm = getSystemService(
+       Context.POWER_SERVICE);
+     powersaving = pm.isPowerSaveMode();
+     Log.i(TAG, "Power saving=" + powersaving);
+     statsHandler.sendEmptyMessage(
+       Util.isInteractive(this) && !powersaving ?
+         STATS_START : STATS_STOP  ❺
+     );
+   };

// [snip]

  @Override
  public void onCreate() {
    // [snip]
+   if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) { ❻❼
+     PowerManager pm = getSystemService(POWER_SERVICE);
+     powersaving = pm.isPowerSaveMode();
+     IntentFilter ifPower = new IntentFilter();
+     ifPower.addAction(ACTION_POWER_SAVE_MODE_CHANGED);
+     registerReceiver(powerSaveReceiver, ifPower);
+   }
// [snip]
  }

  // [snip]

  @Override
  public void onDestroy() {
    // [snip]
+   if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) ❽
+     unregisterReceiver(powerSaveReceiver); ❾
// [snip]
  }
}
```

❶ Disable data logging methods to suppress output.

❷ Deactivate network speed statistics when *Power Save Mode* is activated.

❸ An instance of `BroadcastReceiver` is created to implement the handler of Power Save Mode events.

❹ A decorator is used to make sure *Power Save Mode* changes are only applied to a compatible Android version.

❺ Network speed statistics have to be deactivated upon different events. This is a duplicate of ❷.

❻ Subscribe event of *Power Save Mode* activation.

❼❽ A conditional statement is used to ascertain *Power Save Mode* is only applied to a compatible Android version.

❾ Subscribe *Power Save Mode* event.

Although the concept of creating a *Power Save Mode* is relatively simple, this example illustrates that a number of code changes have to be made that have an adverse impact on code maintainability. For instance, it requires adding duplicated code and adding conditional statements to check the version of Android, increasing cyclomatic complexity. This form of coding goes against some of the guidelines for building maintainable software [15].

We are concerned that, while improving energy efficiency, developers are decreasing the maintainability of their projects, and consequently increasing technical debt. In this work, we use a dataset of energy efficiency-oriented changes to measure the difference in maintainability incurred in Android applications when those changes were applied. Therefore, in this work, we want to answer the following research questions.

> ***RQ1:*** *What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps?*

Why: Energy efficiency often requires to change codebases and even the features of a mobile application. If maintainability is not addressed, these improvements may significantly increase technical debt and require rework during the lifetime of the project.

[1]More information about the app NetGuard on Google Play app store: https://play.google.com/store/apps/details?id=eu.faircode.netguard&hl=en (Visited on July 26, 2019) .

[2]Commit taken from NetGuard project's Github repository, available at: https://github.com/M66B/NetGuard/commit/2e70a038970d6efe9f74e5719e7648f91de30498 (Visited on July 26, 2019)

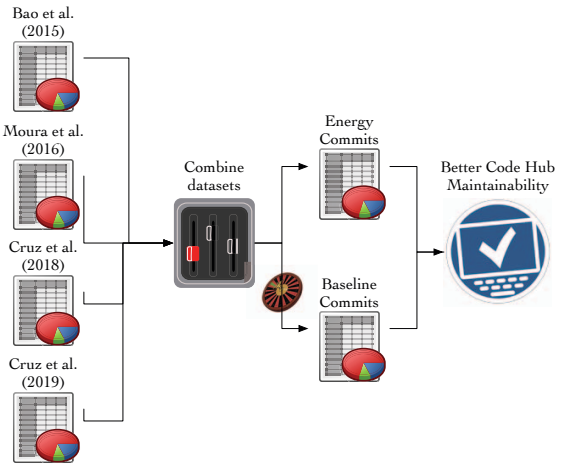

Fig. 1. Methodology for data collection.

How: We analyze a combination of previous datasets with 539 energy-oriented commits. We compute the maintainability score of these commits using the online tool BCH. We apply the same approach to a dataset of regular commits to use as baseline and compare results.

> ***RQ2:*** *Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps?*

Why: Some energy patterns might be more complex to implement than others. By understanding which patterns are more likely to introduce maintainability issues, we bring awareness to mobile app and mobile SDK developers of code changes that require more attention.

How: We use the classification of developers activities made in previous work [4], [13], [20], [21] to group energy-oriented commits and analyze maintainability independently.

> ***RQ3:*** *What are typical maintainability issues introduced by energy-oriented code changes?*

Why: By using examples of typical maintainability issues in real energy-oriented commits, practitioners and researchers will have a more tangible concept of how energy efficiency may hinder maintainability.

How: First, we select energy-oriented commits that yielded low maintainability. Then, we manually inspect these commits and discuss the potential issues entailed by energy efficiency improvements.

III. Methodology

We use the approach illustrated in Fig. 1 to analyze how energy commits affect the maintainability of Android applications. It comprises the following steps:

1) Combine the datasets from related work that classifies the activities of developers addressing energy efficiency in mobile apps [4], [13], [20], [21].
2) Collect regular commits from Android apps to be used as baseline.
3) Compute the impact of energy-oriented commits on maintainability, using BCH.

TABLE I
Datasets that were combined from previous work.

Authors	Ref.	# Commits	Platforms
Moura et al. (2015)	[20]	2188	Android, iOS, non-mobile
Bao et al. (2016)	[21]	468	Android
Cruz et al. (2018)	[4]	59	Android
Cruz et al. (2019)	[13]	431	Android, iOS

A. Dataset

Our work uses the data collected in four previous studies [4], [13], [20], [21] to assess the impact of energy management-oriented changes on the maintainability of Android software. The datasets are summarized in Table I and explained below.

Moura et al. (2015) mined more than 2000 commits to understand energy management activities in general-purpose applications [20]. Their findings suggest that energy efficiency techniques have to be carefully chosen to ascertain that the correctness of the software remains intact. In an extension of this work [21], Bao et al. (2016) used a similar approach to focus exclusively on Android apps, having mined 468 energy management commits. They found that apps in different categories typically have different approaches to energy efficiency.

Cruz et al. (2018) have provided energy efficiency patterns in an automatic refactoring tool [4]. The tool was used to analyze 140 open-source Android apps. As an outcome, the authors submitted 59 pull-requests containing energy improvements to the official repositories of open-source Android applications. In another work, Cruz et al. (2019) proposed a catalog with 22 energy patterns to help developers design energy efficient mobile applications [13]. The authors mined the commits, issues, and pull requests of 1027 Android apps and 726 iOS apps to understand how developers address energy efficiency issues. The catalog can be used to help novice developers learn advanced energy management techniques from existing practices.

From all the data collected, we only select commits from Android projects. Changes from other platforms, such as iOS and Desktop software, were filtered out. Moreover, we cleansed the dataset by filtering out projects that have been deleted and by updating projects that have moved their repositories to a different location. In addition, datasets [20] and [21] include commits that have not been manually validated – we only include commits that the authors manually ascertained as proper energy changes.

In addition, we reuse the categorization of the energy changes defined in the original datasets. Despite similar, different datasets use different labels to indicate the same pattern. For example, the same pattern is labeled as *PowerConditionalStrategy:PowerSaveMode* by Bao et al. (2016) [21] and as *Power Save Mode* by Cruz et al. (2019) [13]. We map these and other identical categories into unique labels[3]. In sum, energy commits are classified into seven categories:

- **Bug Fix & Code Refinement.** Changes related to fixing energy bugs, or refactoring code that already implements energy management features.

[3]The whole set of identical categories can be found in the replication package:https://figshare.com/s/16397140e8183708d248 (Visited on July 26, 2019).

- **Power Awareness.** Have a different behavior when the device is connected/disconnected to a power station or has different battery levels.
- **Power Save Mode.** Implementation of an energy efficient mode in which some features are deactivated to improve better energy usage.
- **Power Usage Monitoring.** Developers add UIs or configurations to inform users about the status of the battery and let them make informed decisions about their interaction with the application.
- **Wakelock Addition.** Wakelocks are used when apps execute tasks that may take longer to execute and need to prevent resources from getting into a sleep state (e.g., screen, network, audio, etc.).
- **Wakelock Optimization.** Inappropriate usage of wakelocks may incur into unnecessary energy usage. Thus, often developers have to optimize wakelock behavior, or even replace them with other techniques (e.g., event handlers).
- **Miscellaneous.** This comprises several categories of energy commits. Since we perform hypothesis tests to statistically validate results, we need to have at least 20 commits per category. Thus, when a category comprises less than 20 commits, we label it as *Miscellaneous*.

B. Baseline Commits

Although we want to assess the maintainability of energy commits, there is no evidence in previous work on how regular commits affect the maintainability of Android projects. E.g., if energy-oriented commits hinder maintainability, we need to understand whether this result is in fact different from general purpose commits. Thus, in parallel with energy commits, we also analyze the maintainability of all other commits and use these as a baseline to answer RQ1 and RQ2.

The baseline dataset is collected as follows: for each energy commit, we obtain all the commits of the respective project and randomly select one. In addition, we randomly select 20 commits to validate that commits are similar in terms of complexity. By using the dataset of energy commits as input for our baseline dataset we make sure that differences in maintainability in the two datasets are not originated by the specificities of different Android projects (e.g., different contribution policies, coding guidelines, app categories, etc.).

C. Maintainability Analysis

We make use of the Software Improvement Group's web-based source code analysis service *Better Code Hub* (BCH for short[4]) to collect maintainability reports of the projects. BCH delivers a maintainability model based on 10 guidelines [15]:

1) **Write short units of code.** Long units are hard to test, reuse, and understand.
2) **Write simple units of code.** Keeping the number of branch points low makes units easier to modify and test.
3) **Write code once**. When code is duplicated, bugs need to be fixed in multiple places, which is inefficient and prone to errors.
4) **Keep unit interfaces small**. Keeping the number of parameters low makes units easier to understand and reuse.
5) **Separate concerns in modules**. Changes in a loosely coupled codebase are much easier to oversee and execute than changes in a tightly coupled codebase. This is computed based on the total fan-in of all methods in a module. Note that a module in Java and other object-oriented languages translates to a class.
6) **Couple architecture components loosely**. Independent components ease isolated maintenance.
7) **Keep architecture components balanced**. Balanced components ease locating code and foster isolation, improving maintenance activities.
8) **Keep your codebase small**. Small systems are easier to search through, analyze, and understand code.
9) **Automate tests**. Automated testing makes development predictable and less risky.
10) **Write clean code**. Code without code smells is less likely to bring maintainability issues.

For each guideline, BCH evaluates the compliance against a particular guideline by setting boundaries for the percentage of code allowed to fall in each of the four risk severity categories (*low risk*, *medium risk*, *high risk*, and *very high risk*). If the thresholds are not violated, the project is considered to be compliant with the guideline. According to BCH, the guideline thresholds are calibrated yearly based on a representative benchmark of closed and open source software systems. Being compliant with a guideline means that the project under analysis is at least better than 65% of the software systems in BCH's benchmark.

The BCH report of the app *NetGuard* for a non-compliant guideline can be seen in Fig. 2. This was extracted from the report of the app *NetGuard*, used in the motivating example of Section II. The green bar represents the percentage of compliant lines of code. These lines of code are considered to be compliant with ISO 25010 standard for maintainability [22]. The yellow, orange and red bars represent non-compliant lines of code with *medium*, *high*, and *very high* severity levels, respectively. Along the bars, there are also marks that refer to the compliance thresholds for each severity level. The report is equivalent to the information reported in Table II: a set of thresholds, number of lines of code (LOC), and percentage of the project for each severity level. Nonetheless, thresholds provided by BCH do not sum to 100%: non-compliant levels are provided in a cumulative way (e.g., the threshold for the medium level includes high and very high levels); the compliant-level threshold is the complement of the medium-level threshold.

Since we want to analyze maintainability regression, we use BCH to compute maintainability in two different versions of the Android app: a) the version of the project before the energy commit (v_{E-1}) and b) the version immediately after

[4]*Better Code Hub*'s website available at https://www.bettercodehub.com/ (Visited on July 26, 2019)

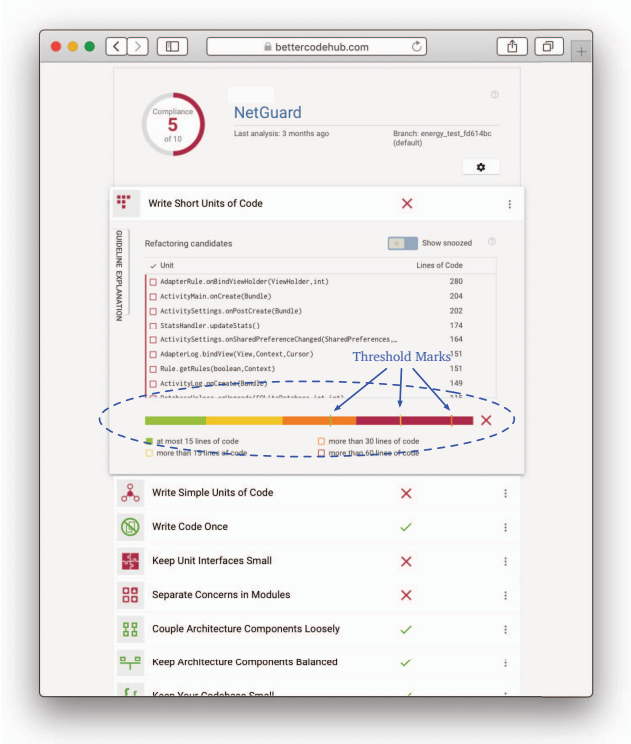

Fig. 2. BCH's maintainability report of the app *NetGuard* for the guideline *Write short units of code*. The app does not comply with the guideline because the bars are not reaching the threshold marks.

TABLE II
EXAMPLE OF A BCH REPORT FOR A NON-COMPLIANT CASE.

Level	Threshold (%)	LOC	Percentage of Code (%)
(Low)	(56.3)	(1353)	(18.6)
Medium	43.7	1683	23.2
High	22.3	1622	22.4
Very High	6.9	2588	35.7

the energy commit (v_E). This is illustrated in Fig. 3.

Although BCH provides a detailed report of the maintainability of the project, it does not compute a final score that we can use to compare maintainability amongst different projects. Thus, based in previous work [18], we designed an equation to capture the distance between the current state of the project and the standard thresholds. We have adjusted the equation to meet the following requirements:

- **The maintainability difference between two versions of the same project is not affected by its size.** In this work, we want to evaluate the identical energy patterns occurring in different projects. Thus, the metric cannot use normalization based on its size – we convert percentage data to the respective number of lines of code.
- **Distance to the thresholds in high severity levels is more penalized than in low severity levels.** We use weights based on the severity level to count lines of code that violate maintainability guidelines.

We compute the mean average of the maintainability score $M(v)$ for all the selected guidelines, as follows:

$$M(v) = \sum_{g \in G} M_g(v) \quad (1)$$

where:
G = selected maintainability guidelines from BCH (e.g., *Write short units of code*, etc.)
v = version of the app under analysis.

The maintenance M based on the guideline g for a given version of a project is computed with the following equation:

$$M_g = \frac{1}{|L|} \sum_{l \in L} C(l), \; L = \{medium, high, veryHigh\} \quad (2)$$

where:
C = compliance with the maintainability guideline for the given severity level (medium, high, and very high)
L = severity levels of maintainability infractions.

The compliance C for a given severity level l is derived by:

$$C(l) = LOC_{compliant}(l) - w(l) \cdot LOC_{\neg compliant}(l) \quad (3)$$

where:
$LOC_{compliant}(l)$ = lines of code that comply with the guideline at the given severity level l
$LOC_{\neg compliant}(l)$ = lines of code that do not comply with the guideline at the given severity level l
$w(l)$ = weight factor to boost the impact of non-compliant lines in comparison to compliant lines.

Finally, the term $w(l)$ is calculated as follows:

$$w(l) = \frac{1 - T(l)}{T(l)} \quad (4)$$

where:
$T(l)$ = threshold in percentage of the lines of code that are accepted to be non-compliant with the guideline for the severity level l. This is a standard value defined by BCH, as illustrated in Fig. 2 and Table II.

In other words, the factor w is used in Eq. 3 to highlight the lines of code that are not complying with the guideline. For instance, the threshold for the severity level *veryHigh* is defined in Table II as $T(veryHigh) = 6.9\%$, which derives to a weight of $w(veryHigh) = 13.5$. This means that, in this example, one non-compliant guideline is decreasing maintainability score by 13.5 points while a compliant guideline is increasing by 1.0 point. In addition, a version that is perfectly aligned with the standard thresholds has a maintainability score of zero.

Then, we compute the difference of maintainability (ΔM) between the energy commit (v_E) and its parent commit (v_{E-1}), as illustrated in Fig. 3.

Statistical validation: To validate the maintainability differences in different groups of commits (e.g., baseline and energy commits) we use the Paired Wilcoxon signed-rank test with the significance level $\alpha = 0.05$. In other words, we test the null hypothesis that the maintainability difference between pairs of versions v_{E-1}, v_E (i.e., before and after an energy-commit) follows a symmetric distribution around 0. This test

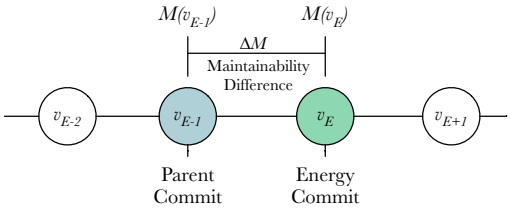

Fig. 3. Maintainability difference for the energy commit v_E.

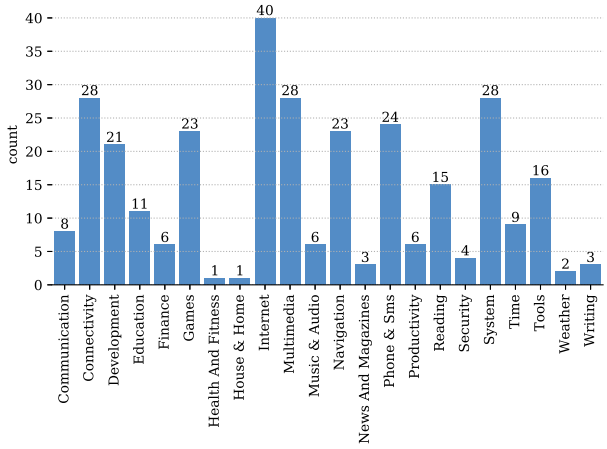

Fig. 4. Categories of apps included in our study with the corresponding app count for each category.

does not capture the absolute value of the maintainability differences. Thus, it is not affected by confounding factors, such as the size of the code changes in different groups.

To understand the effect-size, as advocated by the Common-language effect sizes [23], we compute the mean difference, the median of the difference, and the percentage of cases that reduce maintainability.

D. Typical Maintainability Issues

From the results collected in our dataset, we select the most evident examples of maintainability issues that arise from improving energy efficiency. We manually analyze these energy-oriented commits by examining its message and code changes. The most evident cases are then discussed and presented to illustrate common maintainability issues and bring awareness on how to avoid common issues.

IV. Results

We evaluated a total of 539 energy commits and 539 baseline commits. These commits comprise 306 apps distributed among 22 categories, as depicted in Fig. 4. In this section, we present the results for each proposed research question.

A. What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps? (RQ1)

The results on the impact of different categories of commits in software maintainability are presented in the plot bar of Fig. 5. The plot presents the results for two groups of software changes: **energy commits**, and **baseline commits**. For each

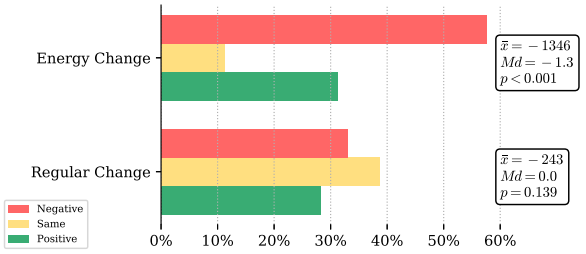

Fig. 5. Maintainability differences for energy commits and baseline commits.

group, the figure provides three bars with the percentage of commits which 1) decrease maintainability (on top, colored in red), 2) do not change maintainability (in the middle, colored in yellow), and 3) increase maintainability (in the bottom, colored in green). In addition, the figure provides, for each group, the mean ($\bar{x}$) and the median (Md) of the maintainability difference, and the p-value of the Wilcoxon signed-rank test (p).

In the case of the regular commits, used as a baseline, 33.0% decrease maintainability (183 cases), 38.7% do not change maintainability (215 cases), and 28.3% improve maintainability (157 cases). Since the p-value of the Wilcoxon signed-rank test ($p = 0.139$) is not below the significance level ($\alpha = 0.05$), there is no statistical significance of the impact of regular commits on maintainability.

On contrary, we observe clear changes for energy commits: 57.1% (310 cases) decrease software maintainability, 10.7% do not change maintainability (61 cases), and 31.2% improve maintainability (168 cases). The results for the Wilcoxon signed-rank test show statistical significance that energy commits decrease the maintainability of Android applications ($p < 0.001$).

B. Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps? (RQ2)

Results of the maintainability impact per category of energy changes are presented in Fig. 6. The Wilcoxon signed-rank test yields statistical evidence that the categories *Miscellaneous* (p = 0.021), *Power Save Mode* (p = 0.012), and *Wakelock Addition* (p = 0.003) significantly decrease the maintainability of Android projects.

The remaining patterns, (i.e., *Bug Fix & Code Refinement*, *Power Awareness*, *Power Usage Monitoring*, and *Wakelock Optimization*) yielded more cases in which maintainability was negatively affected. However, for these patterns, results are not statistically significant.

In the category *Miscellaneous*, 53.1% of changes (77 cases) have decreased maintainability, while 15.9% (23 cases) did not bring any impact, and 31.0% (45 cases) have improved maintainability. The impact is more evident in the category *Power Save Mode*, decreasing maintainability in 78.3% of changes (18 cases), leaving 4.3% unaffected (1 case), and 17.4% (4 cases) with an observed improvement in maintainability. Finally, in the category *Wakelock Addition*, 65.5% have hindered maintainability (55 cases), 6.0% (5 cases) have not

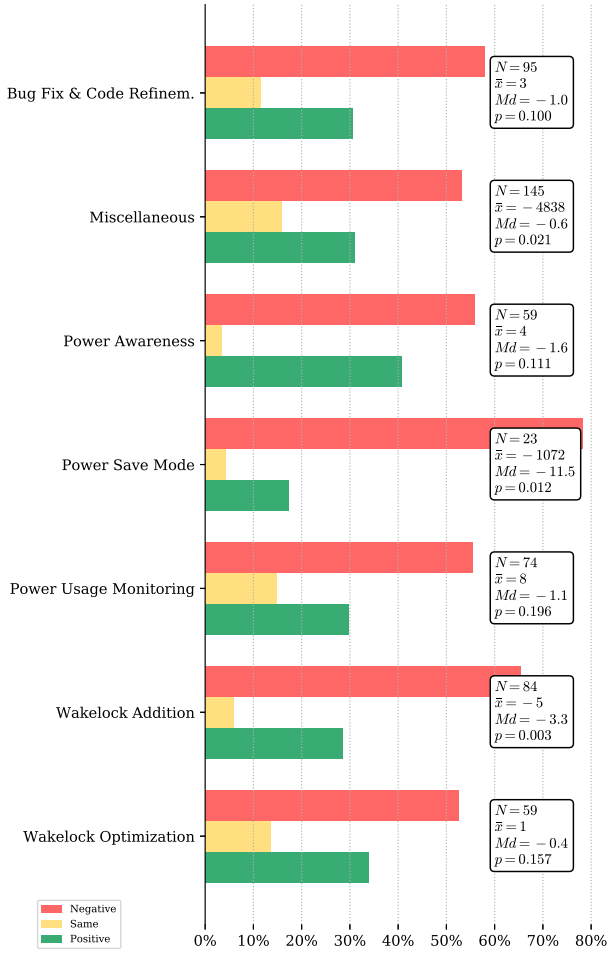

Fig. 6. Maintainability differences among different types of energy commits.

yielded any difference, and 28.6% (24 cases) have registered an improvement.

C. What are typical maintainability issues introduced by energy-oriented code changes? (RQ3)

The following examples illustrate a subset of the maintainability issues we encounter that originate from energy-oriented changes (Maintainability Instances 1–5).[5]

Maintainability Instance 1

Git Repository: https://github.com/ccrama/Slide
Commit: `070c2c6`
Change: Merge two different categories of notifications in the same operation. This is a common approach to improve energy efficiency, coined as *Batch Operations*[6] [13].
Maintainability issue ($\Delta M = -949$)***:*** While coalescing different tasks, methods ended up being extremely large. As a best practice, Java methods should not go over 15 lines of code [15]. Thus, the guideline *Write Short Units of Code* was violated in method `SubredditView.onOptionsItemSelected()`, which ended with 209 lines of code. Several small helper methods should have been implemented to keep this method short.

Maintainability Instance 1 shows an example of maintainability issues that were likely introduced by the lack of awareness by developers on best practices for maintainability. Before applying the code change, the project already had 30 methods with over 200 lines of code. Extracting issues that are strictly related to energy-efficiency improvements is not straightforward. Thus, we skip examples in which this distinction was not clear and opted for selecting maintainability issues that arise from improving energy efficiency in projects with a positive maintainability score.

Maintainability Instance 2

Git Repository: https://github.com/mozilla/MozStumbler
Commit: `37819d9`
Change: New behavior to update the current GPS location. When the user is not moving – i.e., the accelerometer is not sensing any movement – the GPS is turned off and the location is assumed to be constant. When the user moves again, the GPS location updater is reactivated. This instance is an example of energy pattern and *Sensor Fusion* [13].
Maintainability issue ($\Delta M = -60$)***:*** Although this new behavior for GPS updates was added by default in the mobile application, the previous behavior remained as an option in the codebase. This entailed some code duplications: the logic needed to read data from GPS satellites is exactly the same in both behaviors. This violates the *Write Code Once* guideline.

Maintainability Instance 3

Git Repository: https://github.com/mozilla/MozStumbler
Commit: `6ea0268`
Change: Added support for a power save mode [13], in which the app stops scanning cell towers and Wi-Fi networks. This change required adding extra logic in the `onCreate` method of `MainActivity` class. In short, the method was changed to verify whether the battery level was low and whether the *Power Save Mode* was enabled in the app.
Maintainability issue ($\Delta M = -20$)***:*** Although the idea seems trivial, developers had to add 18 extra lines of code to the already existing `MainActivity.onCreate()` method. The method ended up with 45 lines of code, violating the *Write Short Units of Code* guideline.

Maintainability Instance 4

Git Repository: https://github.com/einmalfel/PodListen
Commit: `2ed5a65`
Change: Add a preference in which users can opt to download new content (i.e., podcasts) only when the smartphone is connected to the charger. This is an implementation of the energy patterns *User Knows Best* and *Power Awareness* [13].

[5]The whole instances can be found in the replication package: https://figshare.com/s/16397140e8183708d248 (Visited on July 26, 2019)

[6]More information about *Batch Operations* and other energy patternhttps://tqrg.github.io/energy-patterns/#/patterns/Batch_Operation (Visited on July 26, 2019).

Maintainability issue ***($\Delta M = -20$):*** By adding this new user-defined setting, conditional logic was added to the beginning of the affected methods (e.g., method `DownloadReceiver.updateDownloadQueue`) to verify the preferences and the phone charging status. This leads to a higher number of branch points per method (maximum recommended of four [15]), violating the maintainability guideline *Write Simple Units of Code*. In these cases, the recommended approach is to split the method into simpler ones.

MAINTAINABILITY INSTANCE 5

Git Repository: https://github.com/horn3t/PerformanceControl
Commit: `cb3080e`
Change: Based on the battery level of the smartphone, adjust the power leveraged to CPU and GPU cores. This a very low level code change that resorts to the execution of bash commands to control the hardware of a smartphone device. This example does not implement a documented energy pattern.
Maintainability issue ***($\Delta M = -37$):*** Although the nature of the change implies adding code with poor readability, there are other maintainability issues that should have been avoided. In particular, the class `GPUClass`, which was added to control GPU power, violates the guideline *Separate Concerns in Modules*. The methods of this class have a high number of references through the code (i.e., high fan-in). A typical approach to address this issue is to split the class in separate concerns [15].

V. DISCUSSION

In this section, we answer our research questions, discussing the implications of the analysis of results.

A. What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps? (RQ1)

The majority of energy efficiency-oriented changes hinder the maintainability of Android projects. Results presented in Fig. 5 shows a decrease in maintainability in 57% of the cases. This raises a new tradeoff when developers need to address energy efficiency in their projects.

Previous work found evidence that developers struggle to improve the energy efficiency of their software, lacking the knowledge and tools to aid in this problem [1]. Our work corroborates by showing that developers may have to reduce maintainability for the sake of energy efficiency.

In our perspective, developers need to be able to create energy efficient code without potentially ruining the maintainability of their projects. Otherwise, they may not apply such fixes or come with too many negative code maintenance consequences. We understand that this problem needs to be addressed at several levels:

- **Mobile frameworks** need to feature energy patterns out-of-the-box without requiring too many changes in the software codebases.
- **Documentation** of mobile libraries and frameworks need to provide developers with the best practices to implement energy patterns.
- **Programming languages** should provide coding mechanisms to easily implement energy patterns without compromising maintainability. Previous work has already started addressing energy-efficiency concerns in programming languages [24], [25]. Hopefully, these efforts can be ported to the official mobile programming languages (e.g., Java, Kotlin, Swift, etc.).
- **Mobile Developers** have to look out for maintainability issues when implementing energy patterns. Online services such as BCH, that play well in a continuous integration pipeline, can help developers to be more aware of the maintainability issues introduced by their changes. By bringing awareness, developers can put more effort on improving the maintainability of their code and avoid common issues (e.g., code duplication).

B. Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps? (RQ2)

Energy patterns **Miscellaneous, Power Save Mode, *and* Wakelock Addition *significantly decrease the maintainability of Android projects.*** Although the remaining patterns *Bug Fix & Code Refinement*, *Power Awareness*, *Power Usage Monitoring*, and *Wakelock Optimization* seem to reduce maintainability, no statistical evidence was found.

This is particularly disconcerting because *Power Save Mode* and *Wakelock Addition* are recommended as power management solutions in the official documentation of the Android SDK[7]. However, it seems that more support is needed in order to implement patterns without compromising the maintainability of Android projects.

Documentation should be enriched with more examples and best practices to implement these patterns. We were not able to find those in the official Android documentation. Moreover, the documentation does not consistently refer to the *Power Save Mode* pattern by this name, referring to it as *Battery Saver* in a few cases[8].

In addition, different Android versions feature different mechanisms to these patterns. However, developers need to make sure their software runs efficiently in different versions of Android [26], [27]. Thus, this requires adding specific logic for each API level, adding more complexity to the code and making it less maintainable.

Along with the implications from RQ1, we find that improving support to *Power Save Mode* and *Wakelock Addition* would immediately help developers ship maintainable and energy-efficient mobile software. Actually, tools providing support to automatically apply these patterns while preserving maintainability would be of great benefit.

C. What are typical maintainability issues introduced by energy-oriented code changes? (RQ3)

Although cases with the highest maintainability difference have clear examples of bad maintainability, they are not

[7]Documentation for *Power Save Mode* and *Wakelocks*: https://developer.android.com/reference/android/os/PowerManager (Visited on July 26, 2019).

[8]Android documentation using inconsistent names for *Power Save Mode*: https://developer.android.com/about/versions/pie/power#battery-saver (Visited on July 26, 2019).

entirely affected by the energy improvement per se. That is, other factors, such as the low experience level of the developer, may be the cause of the maintainability issues. This problem is illustrated in the Maintainability Instance 1.

On the contrary, the examples presented in Maintainability Instances 2–5 reveal maintainability issues that are intrinsically related to the strategy used to improve energy efficiency. E.g., in the Maintainability Instance 2, developers created an additional approach to collect sensor data but left the original one as an option. Since the efficacy of the two approaches is different, developers decided to feature both approaches in their app: one more effective but less efficient and the other less effective but more efficient. Given that the app needs to run under many different scenarios with different constraints, mobile apps often support different approaches to the same feature. While this decision may be necessary, the nature of these changes is prone to maintainability issues.

In the Maintainability Instance 4, a number of contextual pre-conditions related to the battery level of the smartphone were checked before granting the execution of particular actions. Mobile development frameworks should provide mechanisms to support typical battery-level scenarios out of the box. For instance, using Java annotations, particular actions could be postponed until power-related requirements are met.

Preliminary related work has proposed programming environments that address energy-efficiency [28], [29]. We show that such solutions are relevant in the context of mobile app development. Moreover, related work has improved the specification of data types to select the most energy-efficient type for a given context [30], [31]. Nevertheless, these solutions address energy efficiency decisions at low-level, lacking support for typical design patterns to address the energy efficiency of mobile apps [13].

The analyzed examples show that maintainability issues lie mainly on the lack of awareness by developers and the insufficient support of energy-efficiency patterns from mobile platforms. New approaches ought to be delivered to help developers assess the maintainability of their code changes when tackling energy-efficiency requirements. For instance, continuous integration and continuous development is a promising approach to address this issue. Although it is known to promote software best practices [32], they are rare in the mobile app world [33], [34]. In addition, results suggest that energy-related changes ought to be tackled by developers with additional care (e.g., code reviews).

VI. Threats to Validity

A. Construct

We use metrics derived from static code analysis to assess software maintainability. However, this is a broad-scoped attribute that may not be fully capture maintainability in its five sub-characteristics: modularity, reusability, analyzability, modifiability, and testability. Nonetheless, previous work has found high correlation between maintainability sub-characteristics and BCH guidelines [19].

In addition, different projects and contexts may require different maintainability standards. Nonetheless, we use statistical hypothesis testing to mitigate confounding factors. Moreover, BCH uses a representative benchmark of closed and open source software systems to compute the thresholds used in each maintainability guideline [15], [17]. This benchmark is updated every year [15].

B. Internal

Maintainability may be affected by different coding styles and experience level from developers of the same project. We do not evaluate differences at that level. In addition, we do not evaluate the maintainability difference for all regular commits in a project. Evaluating all the commits in a project would not be feasible using our methodology. Thus, we assume that the size of the dataset (539 commits) is enough to mitigate random variations in the maintainability differences of the baseline.

The nature of baseline commits scopes general-purpose commits that may be different to energy-oriented commits in a number of characteristics (e.g., lines of code). We assure the two datasets are comparable by collecting the baseline set using a random selection. Moreover, we do not analyze the maintainability difference in terms of absolute values. In other words, we only evaluate whether the maintainability was improved, not changed, or worsen. In future work, we plan to address specific categories of changes in mobile apps.

C. External

The collection of energy-oriented commits used in this work comprises open source apps. Our methodology requires access to data that is not publicly available for commercial apps. The extent to which this findings generalize to commercial apps with non-open source licenses is not assessed. Still, the maintainability challenges pinpointed in our work are relevant to mobile app projects regardless of their license.

We only analyze Android apps. Different platforms and programming languages may require different coding practices to address energy efficiency. We did not study how our findings generalize to other mobile platforms.

We resort to a set of energy changes that were collected from four previous works [5], [13], [20], [21]. These works use the commit message provided by developers to classify a given commit as an energy change. This approach discards energy changes that did not have a commit message describing them as such. Since extending our datasets to these commits is not trivial, we limit the scope of this study to energy-oriented commits with an explicit commit message. Finally, all the energy commits in this work are described in English.

VII. Related Work

In this section, we discuss related works on code maintainability, energy efficiency patterns, and anti-pattern detection.

A. Code maintainability

Previous work has studied the evolution of maintainability issues during the development of Android apps [35]. The authors have observed that maintainability decreases over time, being code duplication the most common maintainability issue. In addition, they found evidence for the fact that maintainability issues in Android apps occur independently of the

type of development activities performed by developers. Their work uses a dataset from related work [36] with an under-represented sample of energy activities, counting with only 12 occurrences. In this work, we focus on a larger sample, counting with 539 energy activities to analyze how energy activities affect the maintainability of Android projects.

A use case study on the Java framework *JHotDraw* suggests that the adoption of design patterns is highly correlated with the maintainability of a project – i.e., the usage of design patterns do improve code maintainability [37]. On contrary, related work shows that some design patterns should be used with caution, since they may bring maintainability and evolution issues to software projects [38]. Our work studies how these findings apply in the case of energy patterns, for open-source Android apps.

Previous work studied the effect of programming languages in the quality of code found [39]. It was found that language design has a significant yet moderate impact on software quality. The authors have used the number of defects as a construct of software quality. Our work analyzes software quality in terms of code maintainability to study how it is affected by energy efficiency-oriented commits.

B. Energy efficiency patterns

Previous works have studied the impact of different energy efficiency patterns on mobile apps. Offloading heavy computation tasks to a cloud server was found to reduce energy consumption up to 50% in mobile apps [40]. Other patterns comprise featuring dark user interface themes achieve better energy usage on mobile devices [41], [42]. Other approaches have improved energy efficiency by finding the optimal number of display updates in a mobile app [43]. Another work has used regular expression representations to assure an optimal usage of the energy intensive resources of mobile devices [44].

The impact of logging practices of developers on the energy consumption of Android apps has also been studied [45]. From the 24 Android apps in this study, 19 exhibited at least one version in which logging statements had a medium or large effect size on energy consumption.

Energy patterns for mobile apps have been widely studied in the literature [13], [20], [21]. Our work acknowledges the importance of using energy patterns to improve energy efficiency. However, we take a step further and study the impact of these patterns on the quality of the app in terms of code maintainability. In addition, we study the change-proneness of these techniques in mobile app codebases.

C. Detecting Anti-patterns in Mobile Apps

Related work has studied how anti-patterns affect the overall energy consumption of Android apps. Previous work on 60 Android apps have studied the influence of 9 Android-specific code smells on energy efficiency. Results showed energy savings up to 87 times after fixing all code smells [46]. Another work has studied the impact of eight performance-based code smells on the energy efficiency of mobile apps [47]. It was found significant differences, up to 5%, on energy consumption by fixing five of the studied code smells. Not only code smells have been studied in this context. The impact of picture smells on energy usage has also been assessed [48]. It was found evidence that significant energy savings incur from using an optimal image compression and format.

These works endorse the importance of using refactoring techniques to improve energy efficiency. In fact, anti-pattern detectors and automatic refactoring tools have been delivered to help developers ship energy efficient code [49]. Cruz et al. have implemented an automatic refactoring tool for Android apps to fix five performance issues that also increase energy usage [4]. Palomba et al. proposed an automated tool to identify 15 Android-specific code smells [50]. These code smells had been flagged by previous work as a potential threat to the maintainability and the efficiency of Android apps [51]. Our work differs by 1) identifying code changes that hinder maintainability and 2) using code changes that already been labeled has an energy improvement.

VIII. Conclusion and Future Work

In this work, we present an empirical study on the impact of energy commits on the maintainability of mobile apps. We used the toolset BCH to collect maintainability reports from a dataset of 539 energy commits of open source Android apps.

We have found evidence that energy-oriented commits significantly decrease software maintainability in open source Android apps: 57% of energy commits were observed to reduce code maintainability. Conversely, no particular influence on maintainability can be observed. In particular, we show that the change on maintainability is more evident for the patterns *Power Save Mode* and *Wakelock Addition*, in which maintainability decreases in 78% and 66% of cases.

Our findings have direct implications for different stakeholders of mobile app development. We highlight that mobile development frameworks should provide mechanisms to implement energy patterns without hindering the maintainability of mobile apps.

As future work, our empirical study can be extended in different ways: analyze which maintainability guidelines are more affected from energy commits; analyze how results stand for different categories of mobile apps; expand our methodology with other software quality properties (e.g., reliability). Furthermore, it would be interesting to validate our findings with other mobile platforms (e.g., iOS), and also with desktop and server applications.

Acknowledgements

We thank SIG's *Better Code Hub* team for all the support as well as help in validating our methodology.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia with reference UID/EEA/50014/2019, the GreenLab Project (ref. POCI-01-0145-FEDER-016718), and the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017).

REFERENCES

[1] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software?" *PeerJ PrePrints*, vol. 3, p. e886v1, 2015.

[2] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 36.

[3] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, 2018.

[4] L. Cruz and R. Abreu, "Using automatic refactoring to improve energy efficiency of android apps," in *XXI Ibero-American Conference on Software Engineering (CIbSE, Best Paper Award)*, 2018.

[5] ——, "Measuring the energy footprint of mobile testing frameworks," in *Software Engineering Companion (ICSE-C), 2018 IEEE/ACM 38th International Conference on*. IEEE, 2018.

[6] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "Multi-objective optimization of energy consumption of guis in android apps," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 14:1–14:47, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3241742

[7] D. Li, Y. Lyu, J. Gui, and W. G. Halfond, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 249–260.

[8] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "Greenscaler: training software energy models with automatic test generation," *Empirical Software Engineering*, pp. 1–44, 2018.

[9] S. Boonkrong and P. C. Dinh, "Reducing battery consumption of data polling and pushing techniques on android using gzip," in *Information Technology and Electrical Engineering (ICITEE), 2015 7th International Conference on*. IEEE, 2015, pp. 565–570.

[10] S. A. Chowdhury and A. Hindle, "Greenoracle: Estimating software energy consumption with energy measurement corpora," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 49–60.

[11] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Petra: a software-based tool for estimating the energy profile of android applications," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 3–6.

[12] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, 2017.

[13] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, Dec 2019.

[14] International Organization for Standardization, "Systems and software engineering: Systems and software quality requirements and evaluation (SQuaRE): System and software quality models," ISO/IEC, Geneva, Switzerland, Standard, 2011.

[15] J. Visser, S. Rigal, R. van der Leek, P. van Eck, and G. Wijnholds, *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.", 2016.

[16] T. Kuipers, I. Heitlager, and J. Visser, "A practical model for measuring maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)(QUATIC)*, vol. 00, 09 2007, pp. 30–39. [Online]. Available: doi.ieeecomputersociety.org/10.1109/QUATIC.2007.8

[17] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[18] M. Olivari, "Maintainable production: Mapping software quality change to source code contributions," Master's thesis, University of Amsterdam, 2018.

[19] 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.

[20] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining energy-aware commits," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 56–67.

[21] L. Bao, D. Lo, X. Xia, X. Wang, and C. Tian, "How android app developers manage power consumption?: An empirical study by mining power management commits," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 37–48.

[22] O. internationale de normalisation, *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. ISO/IEC, 2011.

[23] K. O. McGraw and S. Wong, "A common language effect size statistic." *Psychological bulletin*, vol. 111, no. 2, p. 361, 1992.

[24] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2017, pp. 256–267.

[25] W. Oliveira, R. Oliveira, and F. Castor, "A study on the energy consumption of android app development approaches," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 42–52.

[26] H. Muccini, A. D. Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*, 2012, pp. 29–35. [Online]. Available: https://doi.org/10.1109/IWAST.2012.6228987

[27] K. An, N. Meng, and E. Tilevich, "Automatic inference of java-to-swift translation rules for porting mobile applications," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: ACM, 2018, pp. 180–190. [Online]. Available: http://doi.acm.org/10.1145/3197231.3197240

[28] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2018, pp. 41–53.

[29] H. S. Zhu, C. Lin, and Y. D. Liu, "A programming model for sustainable software," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 767–777.

[30] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 831–850.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[32] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," *32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[33] L. Cruz, R. Abreu, and D. Lo, "To the attention of mobile software developers: Guess what, test your app!" *Empirical Software Engineering*, Feb 2019.

[34] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[35] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago, "How maintainability issues of android apps evolve," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 334–344.

[36] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di 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*, 2018.

[37] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or reality? analyzing the effect of design patterns on software maintainability," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. Springer, 2012, pp. 138–145.

[38] F. Khomh and Y.-G. Gueheneuce, "Do design patterns impact software quality positively?" in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 274–278.

[39] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[40] Y.-W. Kwon and E. Tilevich, "Reducing the energy consumption of mobile applications behind the scenes," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 170–179.

[41] T. Agolli, L. Pollock, and J. Clause, "Investigating decreasing energy usage in mobile apps via indistinguishable color changes," in *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, 2017, pp. 30–34.

[42] M. Linares-Vásquez, C. Bernal-Cárdenas, G. Bavota, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Gemma: multi-objective optimization of energy consumption of guis in android apps," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 11–14.

[43] D. Kim, N. Jung, Y. Chon, and H. Cha, "Content-centric energy management of mobile displays," *IEEE Transactions on Mobile Computing*, vol. 15, no. 8, pp. 1925–1938, 2016.

[44] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 139–150.

[45] S. Chowdhury, S. Di Nardo, A. Hindle, and Z. M. J. Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1422–1456, 2018.

[46] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.

[47] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, 2017, pp. 46–57.

[48] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, p. 10.

[49] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: an energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2018.

[50] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 487–491.

[51] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.