Anatomy of Functionality Deletion

An Exploratory Study on Mobile Apps

Maleknaz Nayebi
iDB lab, University of Toronto
m.nayebi@utoronto.ca

Konstantin Kuznetsov
Saarland University / CISPA
kuznetsov@st.cs.uni-saarland.de

Paul Chen
SEDS lab, University of Calgary
Paul.chen@ucalgary.ca

Andreas Zeller
Saarland University / CISPA
zeller@cs.uni-saarland.de

Guenther Ruhe
SEDS lab, University of Calgary
ruhe@ucalgary.ca

ABSTRACT

One of Lehman's laws of software evolution is that the functionality of programs has to increase over time to maintain user satisfaction. In the domain of mobile apps, though, too much functionality can easily impact usability, resource consumption, and maintenance effort. Hence, does the law of continuous growth apply there? This paper shows that in mobile apps, deletion of functionality is actually common, challenging Lehman's law. We analyzed user driven requests for deletions which were found in 213,866 commits from 1,519 open source Android mobile apps from a total of 14,238 releases. We applied hybrid (open and closed) card sorting and created taxonomies for nature and causes of deletions. We found that functionality deletions are mostly motivated by unneeded functionality, poor user experience, and compatibility issues. We also performed a survey with 106 mobile app developers. We found that 78.3% of developers consider deletion of functionality to be equally or more important than the addition of new functionality. Developers confirmed that they plan for deletions. This implies the need to re-think the process of planning for the next release, overcoming the simplistic assumptions to exclusively look at adding functionality to maximize the value of upcoming releases. Our work is the first to study the phenomenon of functionality deletion and opens the door to a wider perspective on software evolution.

KEYWORDS

Deletion, Taxonomy, Functionality, Mobile apps, Survey, App store mining

ACM Reference format:
Maleknaz Nayebi, Konstantin Kuznetsov, Paul Chen, Andreas Zeller, and Guenther Ruhe. 2018. Anatomy of Functionality Deletion. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories , Gothenburg, Sweden, May 28–29, 2018 (MSR '18),* 11 pages.
DOI: 10.1145/3196398.3196410

1 INTRODUCTION

Textbooks in Software Engineering often cite Lehman's laws of software evolution [13], stating universal experiences such as software has to be continuously adapted to avoid becoming less satisfactory over time. Lehman's sixth law, however, not only is concerned with change; it also postulates *growth:* "Functional content of a program must be continually increased to maintain user satisfaction over its lifetime." For service-oriented platforms such as operating systems, this may be true; in particular, as functionality once introduced has to stay in order to maintain backward compatibility. If a program functionality is mainly invoked by users, though, an ever-increasing set of features is in sharp conflict with usability, as more and more features compete for being easy to discover and easy to use. We thus, pose the question: Is it true that functionality of a program *must* be increased over time? Or couldn't it be the case that functionality is actually *reduced* in order to improve usability? So far, functionality removal has been discussed as a spectrum that "at one end of the dimension, the whole software product is eliminated; at the other, no code is removed at all." [17].

To study these questions, the domain of *mobile applications* is particularly interesting. On mobile devices, additional functionality comes at a cost: First, the small screen severely limits the number of features that can be offered by an application. Second, computation demands and memory usage may impact battery life. Hence, developers should have an interest in *removing* functionality that negatively impacts the user experience. Finally, there is a myriad of applications to study—including their releases over time.

The first exploration of Android mobile apps shows that such *deletion* of functionality may actually be the case. Here we assume that some functionality might have been removed from the app if the size of an app release was reduced in comparison to the previous release.

Figure 1 shows the release sizes of three popular Android apps over time—the Firefox browser, the Wikipedia app, and the Flym feed reader. We see that over time, many releases had a larger size

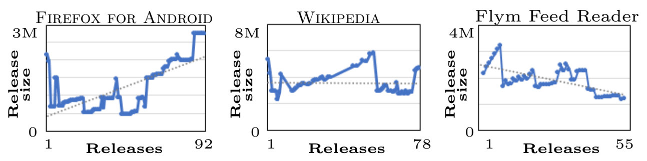

Figure 1: App size across multiple releases. Apps do grow and shrink over time.

than their predecessors; however, we also see that again and again over time, some releases showed a *reduction* in size instead.

1.1 Preliminary study

We assumed that if the size of a release is not smaller than the previous release(s), then no functionality has been removed. In order to analyze *how frequently has the size of a release been reduced?* we did a preliminary study on app size evolution with the intention to motivate anatomizing functionality deletion.

To test our assumption, in May 2017 we collected all the open source Android apps from F-Droid that are hosted on GitHub. In total, we studied 1,519 open source Android mobile apps over 20,806 GitHub releases to analyze the size trend of mobile apps. We considered two ways to measure the size of a mobile app's release: (i) the size of the app's binary file (*.apk file) as it is in the app store ($Size_{APK}$), and (ii) the size of the code base at the time of the release, available via GitHub ($Size_{codebase}$). Each of these measures only provide an approximate estimation of the size. We compared these proxies to find their relation.

$Size_{APK}$ is the most intuitive measure as the APK file is what a user downloads and installs on an Android smart device to use a mobile app. APK's file size is available along with each release both in the Google Play store and F-Droid. App stores do not maintain all versions of an app, making the continuous analysis of the release impossible. Furthermore, the APK file of the same release can vary in size with the device due to different build options such as resource minimization, compression of native libraries, and code optimization. It is possible to build an APK from sources, preserving the same build configuration. However, this approach needs significant manual effort, as the compilation of each release may require the manual update of specific settings.

The size of the code base $Size_{codebase}$ at the time of release is available on Git repositories. It can be retrieved for each and every release, providing complete information on app size evolution. However, a repository may contain various unrelated files like legacy resources, tests or additional release bundles, which affects the proper estimation of the size. To test whether these files significantly bias the $Size_{codebase}$ or not, we investigated on the association between the $Size_{APK}$ and $Size_{codebase}$.

Google Play store does not provide access to the archive of APK files. Therefore, we crawled APK files from GitHub repositories or third-party stores such as AndroZoo [1], AppCake and Androidha. To retrieve $Size_{codebase}$, we fetched each release of an app from GitHub and cached the size of the code repository at the time of the release. We mapped an APK file to the particular GitHub release based on the release date (which we retrieved from Searchman.com) and identifier [21]. In total, we could gather the APKs of 565 mobile apps and 14,237 releases. Successfully retrieving about 37% of the APK files, we found the correlation of 0.86 between $Size_{APK}$ and $Size_{codebase}$. This allowed us to use $Size_{codebase}$ as our main estimator to investigate the frequency of size reduction over releases, even not having access to all the APK files.

We found that 98.8% of apps had decreased their size at least once over their lifetime. 61.3% of these apps had at least one release with more than 10% reduction in size compared to their previous release. In Figure 2, we plotted (for the 1,519 apps explored) the absolute and relative frequency of releases with decreased size. In 73.7% of all apps, more than half of their releases had a decreased size compared to the previous release; and 33.3% of apps even had a decreasing size trend over time (Similar to Flym app in Figure 1).

Also, we analyzed the number of Android components declared in Android manifest file as the proxy for the app functionality. To answer *how frequently functionality of a release has been reduced?* we examined the reduction in the number of Android components — *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers* — in each app and release. They are the main building blocks of any application and expose app's functionality regardless of whether it is implemented inside these components or in another part of the code. An Android activity implements a single screen with UI elements, which is visible to a user. Services run in the background and perform any long-running operations. A broadcast receiver allows the app to respond to system-wide events like a notification or a call. A content provider gives access to the app's and user's data. These building blocks must be declared in the Android manifest XML file, where we can retrieve it. If the number of activities decreases, this means that fewer screens are available, which hints at less functionality being visible to the user. In Figure 3, we plotted

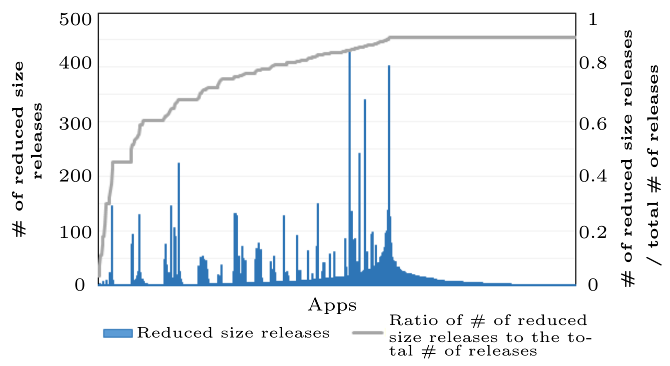

Figure 2: Frequency of releases with decreased size and their relative frequency (releases with decreasing size related to all releases per app).

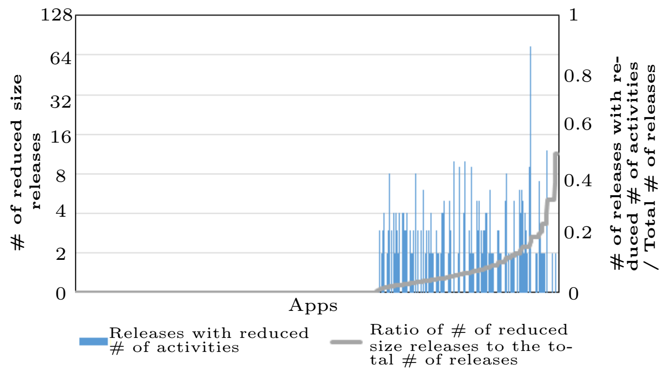

Figure 3: Frequency of releases with decreased number of activities and their relative frequency. Activities are a proxy of available functionality to the end user.

the absolute and relative frequency of releases with fewer activities (sorted by relative frequency). Again, 37.6% of apps had a decrease in activities over their lifetime. We also found that for 88.9% of releases there was no change in their services and some services were removed in 2.7% of the releases. 5.7% of releases removed one or several providers while the rest did not change in terms of that. 1.1% of releases removed receivers and 2.4% added receivers while the rest of releases remained stable.

Although Android components can be considered as a proxy for the functionality, this estimation is rather coarse. Android components represent the big parts of the app, whereas deletions may affect only small chunks of the code. For instance, Android activities represent the whole screens. Sometimes after an update, the screen remains unchanged while the UI elements and their underneath behavior is modified. To mine these changes, we used BACKSTAGE tool [3]. We investigated whether deletions affected particular UI elements with associated API calls. For each UI element, extracted by BACKSTAGE, we checked if it has been deleted in the next release or if any API triggered by it has been removed. Since BACKSTAGE requires binary `APK` file for the analysis, we could investigate only 37% of the apps. We found that 39.8% of apps had UI elements removed (and 22.7% added UI elements) in at least one release.

In almost one third of the analyzed open source Android mobile apps, we observed a decreasing trend in size, activities, and UI elements across releases.

Lehman's laws postulating continuous maintenance are certainly as true as they ever were; but if continuous growth in functionality, as stated by Lehman's laws, translates into bigger program size and more screens, then mobile apps are breaking the law. Still, changes in program size and in the number of screens may not translate into more or less functionality. Hence, in this paper, *we shed some light on the role of deleting functionality in software evolution:* Do developers delete functionality over time? Why, when, and how do they do it?

1.2 Research questions

So far, functionality removal has been discussed as a spectrum [17]. Within this paper, we perform an exploratory study to find the taxonomy of functionality deletions to detail this spectrum and to examine the feasibility of providing decision support tools for developers in this context. We study two main research questions (RQs) by performing a mixed method study:

RQ1: What type of functionality was deleted from mobile apps, and why?
We systematically selected and analyzed 8,000 commit messages from open source mobile apps to define the nature and reasons of deletions. We extracted a taxonomy of the deleted functionality and a taxonomy of the reasons for those deletions.

RQ2: How do developers perform functionality deletion and how do they perceive the extracted taxonomies?
*We performed a survey with 106 app developers and evaluated our taxonomies extracted in **RQ1**. We also asked developers how they decide for functionality deletion and to what extent they plan for it. In particular, we explored the impact of user reviews on driving the decision for functionality deletion.*

This is the first study ever examining the deletion of functionality during software evolution, and also the first study that would challenge Lehman's law of functionality growth in software evolution. It also is one of the first studies to address the evolution of mobile apps. Our results shed new light on why, when, and how developers delete functionality, notably when user reviews suggest that some functionality is annoying, unnecessary, or otherwise negatively impacts the user experience.

In the remainder of this paper, we discuss the methodology and results for both research question in Sections 2, 3, respectively. We discuss the results and their implications in Section 4. This is followed by a discussion on threats to validity (Section 5). After discussing related work in Section 6, we conclude the paper in Section 7.

2 WHAT WAS DELETED, HOW AND WHY?

We start with **RQ1** and retrospectively explore mobile apps to examine what was deleted and why. To this end, we gathered commit messages and analyzed their content.

2.1 Data: Commit messages

For this study, we used open source Android mobile apps. We obtained apps from `F-Droid`, an open source app repository, and the `Google Play` Store. We extracted the app name, package name, and the address of source code repository by crawling `F-Droid`. To unify the process of mining the repositories, we filtered out the apps that were not on `GitHub` (461 out of 1,980 apps). We gathered 1,519 remaining `GitHub` repositories and collected app and release information from `GitHub` logs. We used this data to mine the commits associated with deletion. Figure 4 details the process of mining commits related to functionality deletion (or deletion commits) along with the number of apps and commits retrieved in each step.

In `GitHub`, each commit is associated with added or deleted lines of code. If a developer changes a variable name, this change is shown as one line addition and one line deletion on `GitHub`. A commit shows the changed lines of code in addition to a message that the developer adds to explain her changes. We used this data to select commit messages. We filtered out commits that were not associated with any deleted lines of code. With the aim to anatomize functionality deletion, we focused on the non-trivial changes that a developer applied consciously. For example, we considered the change of variable name which is associated with deleting one line of code as trivial and out of the scope of this study. To systematically separate trivial and non-trivial deletions, two authors manually analyzed 1,500 commit messages of eight apps.

Considering the number of commits of the apps, we picked two apps from each quartile. Then, we randomly select commit messages and two of the authors manually and independently tagged a commit as trivial or non-trivial. The Cohen's Kappa [9] agreement between the two authors in this tagging process was 0.92 which shows almost perfect consensus. Then three of the authors looked into non-trivial commits (994 commits). As a result we considered

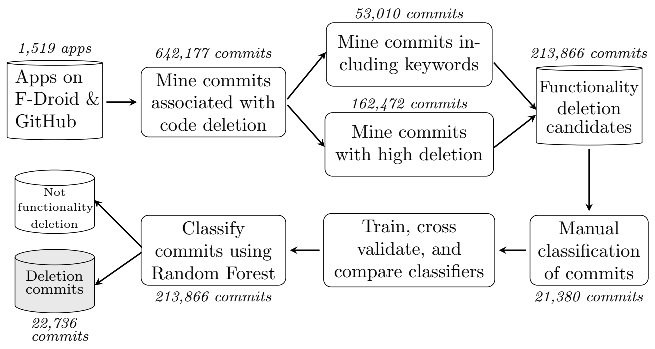
1,519 apps
Apps on F-Droid & GitHub
642,177 commits
Mine commits associated with code deletion
53,010 commits
Mine commits including keywords
162,472 commits
Mine commits with high deletion
213,866 commits
Functionality deletion candidates
Manual classification of commits
21,380 commits
Train, cross validate, and compare classifiers
Classify commits using Random Forest
213,866 commits
Not functionality deletion
Deletion commits
22,736 commits

Figure 4: Selection of deletion commits

commits that fall into the below categories as functionality deletion commits:

Commits containing a keyword: We lemmatized commit messages and select commit messages that include any of the keywords "remove", "revert", "roll back", "discard", "eliminate", "erase", "disable", "delete", and "rm". We selected keywords using our observations during manual analysis. We denote this set by $\text{CM}_{keywords}$.

Commits with high deletion: We gathered deleted lines of code associated with each commit message per app. Based on this data for each app, we calculated the quartiles related to the number of deleted lines of code per commit. To reduce the total number of commits studied, we considered only commits with the size of deletions being in the upper quartile (at least 75%) of the size of all commit deletions. We hypothesized that deleting a big portion of code is intentional and non-trivial. We denote this set by $\text{CM}_{highdeletion}$.

We considered the union of the commits that were found by above proxies for functionality deletion. In this exploratory study, we persisted on keeping our neutral perspective about what is/is not functionality deletion. At this point of our initial analysis we state that functional deletions are a subset of all commits found from keyword search or being in the upper quartile of deletions:

$$\text{Functionality Deletion} \subset \{CM_{keywords} \cup CM_{highdeletions}\}$$

This process resulted in 213,866 commits that are considered candidates for expressing functionality deletion. 1,616 commits were intersections between the two categories. This selection is broad that bring noise (false-positives) into dataset as some of the messages are not related to deletion of any functionality. For example *"Fixed bad synchronization of files removed from another client"* or *"httpfileupload rewritten to use Smack"*. To reduce the noise as such, we trained and evaluated three classifiers being Naive Bayes, Random Forest, and Support Vector Machine (SVM). We randomly selected 21,380 commits (10% of the total set) for training the classifiers. Two of the authors manually labeled these commit messages either as a "functionality deletion" or "not a functionality deletion" (Cohen's Kappa = 0.73). We present the cross-validation results in Table 1. Among the tested classifiers, Random Forest and SVM performed almost equally. However, we selected Random Forrest as it has a slightly better recall for classifying all the commit messages. As a result, we ended up with 22,736 deletion commits.

Table 1: Accuracy of different machine learning techniques for classifying commits related to functionality deletion.

CrossValidation	**10-Fold**			**Leave One Out**		
Classifier	Precision	Recall	F1	Precision	Recall	F1
Naive Bayes	0.64	0.67	0.65	0.70	0.72	0.70
Random Forest	0.74	0.76	0.74	0.79	0.80	0.79
SVM	0.74	0.75	0.74	0.79	0.80	0.79

2.2 Method: Card Sorting

We used card sorting to extract the taxonomies for **RQ1**. Card sorting is an exploratory technique and can be used to derive themes from the text [27, 36]. This technique is used for deriving mental models and taxonomies from qualitative data. Two types of card sorting exist [5, 36]: "open" — where the taxonomy categories can be openly defined and added, and "closed" — where the taxonomy categories are predefined. For our study, we took a hybrid model by categorizing 2,300 cards by an open sorting process to identify categories. We followed by a subsequent closed sorting of 8,000 cards. Figure 5 shows four software developers that assisted us in a card sorting session.

These four developers are professional in at least two programming languages and have been working for a significant time (minimum of two years) with distributed version control systems. They have been trained for 12 hours in the context of this study to understand the context, expectations, commit message analysis, feature and functionality evolution. We also performed a workshop with the four developers having the first author as the facilitator to create a shared understanding of the commit messages and objective of this study. Within this workshop, each developer was assigned randomly to some commit messages and reflected which part of the app has been impacted by the commit. The reflection was discussed in the group and settled once developers get a shared understanding.

We mined two taxonomies in two separate sessions to study "What functionality was deleted" and "Why functionality was deleted". We limited the number of cards we sorted in each session in a way that each session takes no more than eight hours. These are the four steps that we followed during each session:

Figure 5: Card sorting session with defined categories (closed card sorting).

1- **Preparing cards:** We used cards for open sorting from the set of manually labeled commits we prepared for training classifiers in Table 1. For closed card sorting, we randomly selected a subset of commit messages which were identified by the classifier as deletion commits.
2- **Open sorting:** We allocated 180 minutes (3 hours) for this session. The four software developers acted in groups of two to categorize cards and identified the categories, independent of the other group. Each group categorized 1,150 cards in this session. Then, for 45 minutes the session moderator (first author) discussed the mutual and different categories and the team agreed on a set of categories.
3- **Close sorting:** We allocated 300 minutes (5 hours) to this session. Four software developers categorized 8,000 unique cards into the categories defined by closed card sorting. 800 cards were categorized by all the four team members to calculate their degree of conformance. We used Fleiss’ Kappa measure [9] for this purpose. Fleiss’ Kappa was 0.9 on average for the two taxonomies we mined. This shows an almost perfect agreement.
4- **Taxonomy design:** Two of the authors grouped the low-level categories and drew the relation between different categories, independent from each other. Then we discussed the relation and agreed on a high-level taxonomy. We solved disagreements by discussing it with another author as the facilitator.

2.3 Results: Taxonomies

Four developers analyzed 8, 000 commit messages to identify “What was deleted from apps?”, “Why the deletion happened?”, and “How the deletion happened?”. Among 8, 000 commit messages, 10% (800 commit messages) were categorized by all four developers to evaluate the degree of conformance between them. The extent of conformance using Fleiss’ kappa for the “nature of deletions” (“what” taxonomy) was 0.88 and for the “causes of deletion” (“why” taxonomy) was 0.93, which shows almost perfect conformance [9].

⧉ **How was functionality deleted?** By analyzing the commit messages within card sorting process, we identified six different ways that functionality was deleted. The functionality might have been “removed””, “replaced”, “moved”, “reverted”, “re-factored”, or “temporary commented”. As the result of these actions, some app functionality was missed or became inaccessible.

The notion of what constitutes *Functionality deletion* is not seen consistently between developers. By mining the commit messages, we found that deletions happened along with the above development actions. In Section 3, we report the results of a survey with app developers to better understand the What?, How? and Why? of deletions.

⧉ **What functionality was deleted from apps?** By analyzing commit messages, each team (a pair of two developers) identified categories, of which 14 were common between the teams. The first author acted as a facilitator and discussed the rest of the categories and all agreed on eight other categories. We moved the cards around to match with the 22 approved categories. We used the 22 categories for closed sorting. We defined a two-level taxonomy in Table 2. The four high level categories include “security and privacy elements”, “communication bridges”, “User interface elements”, and “Development artifacts”. These categories cover the 22 classes of low-level taxonomy. We provided examples for each of these categories. We believe that the names of the categories are intuitive, and elaborate on a few of the more unclear ones to save space. *Feature* refers to a particular functionality which is usually defined as a phrase for example “removed *vibrate in silent mode*”. The *code* refers to any part of a code that was not specified by the

Table 2: Taxonomy of the nature of functionality deletion to answer “what was deleted?” by analyzing (i) 8,000 commits through card sorting ⧉, and (ii) surveying 106 mobile app developers 👥.

High level taxonomy	Low level taxonomy	% of Cards	Avg. % by developers	Examples
User interface elements	Audio	0.76	3.92	*Remove native audio mixer*
	Themes/Background	1.03	4.93	*Remove Android Holo themes/Remove Osmarender as a background*
	Text	1.51	5.82	*Remove ”.” from menu text*
	Notification	2.51	5.29	*Remove on phone SMS notification*
	Image/icon/animation	3.09	5.76	*Removed old pictures/Remove old vote icons/remove custom slide animations*
	Feature	7.81	6.93	*Remove remember last share location; Remove vibrate on silent mode*
	Other UI elements	12.51	9.32	*Remove relative link in webview; Remove action button; Remove fast scroll*
Development artifacts	GPU	0.34	2.74	*Removed the Sync GPU option from the F-Zero GX ini*
	Log	0.56	4.25	*Removed all sync call added log*
	Document	1.18	4.02	*Remove flipping from the release notes*
	Database	1.25	3.79	*Remove dbLock from HostDatabase*
	Binary	1.26	3.71	*Remove binary on Lollipop*
	Test	1.30	5.18	*Remove unstable integration tests*
	File	2.09	4.63	*Delete obsolete files; Remove files of old nav drawer*
	Config	2.36	3.5	*Remove handling configChanges; Removed Travis config*
	Code	53.91	6.35	*Remove FIFO supplicant state pattern; Removed tiling method polygon*
Security & licensing	License	0.16	1.70	*Remove Apache license*
	Permission	0.61	1.71	*Removed camera permission as unnecessary in Manifest Maps*
	Accounts and access	0.81	3.21	*Removing all references to the username/password stored in the preferences*
Communication bridges	Library/API	0.41	6.97	*Remove sslwebsocket lib/Remove broadcast action from APIEndpoint*
	Plugin	1.39	2.41	*Remove IndeterminateProgress plugin*
	Network/web	2.84	3.86	*Remove dhcp/Remove manual DNS resolution of default servers*

Table 3: Taxonomy of the reasons for functionality deletion "why it was deleted?" by analyzing (i) 8,000 commits through card sorting, and (ii) surveying 106 mobile app developers. 58.78% of commit messages did not include a reason for deletion.

High level taxonomy	Low level taxonomy	% of cards	Avg. % by developers	Examples
Improving user experience	Negative User Feedback	0.23	14.65	*Remove "customer should quite" message suggested by Max*
	Security improvement	0.81	5.26	*Remove check for root access for write permissions prior to installation*
	Usability improvement	4.44	13.57	*Remove rotate and flip controls as it broke the layout*
	Fix a bug/broken functionality	5.79	11.31	*Removed the hack for loading games this fixes the launcher*
Improving quality of existing code	Duplicated functionality	1.16	8.17	*Removes duplicate ifdef_Win32 from VKToString*
	Improving code structure	5.49	5.05	*Undid formatting havoc. We should stick with one code formatter*
	Eliminate deprecated code	7.1	6.6	*DrawContext: Remove the old way of setting uniforms*
	Unused/unneeded functionality	8.62	10.27	*Removed unused ModalDialog style/ Remove useless support* methods.*
Better use of resources	Battery draining	0.1	2.55	*Removed local keep alive to save battery*
	Performance optimization	0.78	3.73	*Switched to WKWebView, performance improvement leveraging Safari.*
	Code optimization	1.53	4.38	*GameRun passes from many functions, removed to reduce loop usage*
Communications	Better management of data	0.75	5.04	*Removed DBTasks, DBWriter is unified method for communication*
	Compatibility issues	4.42	6.86	*Remove save support for Android < 4.4 as there is no further support*

Distribution of the 22,736 commits across categories of what and why taxonomies were almost the same. Hence we did not provide the machine learning results in these tables.

developer in a commit message, and we could not infer any further details about this functionality.

Why was functionality deleted? In a separate session, four developers analyzed the same 8,000 commit messages and categorized them by answering "why did this deletion happen?". The analysis resulted in 16 categories gained from open card sorting. After discussion, the team agreed on 13 categories of reasons for functionality deletion which were grouped into "improving user experience", "improving the quality of the existing code", "Better use of resources", and "Better communication" within the high-level taxonomy that covers these 13 detailed categories. We show the complete taxonomy in Table 3. Commits that were categorized as "Unknown" did not have any description on why the deletion happened, for example "deleted battery indicator".

In our retrospective analysis, 29.98% of functionality deletions were related to UI elements, and 11.27% of functionality deletions were intended to improve users' experience.

Using the results of commit message analysis, we provide insight on the relationship between nature of deletions (what was deleted?) and reasons for deletion (why it was deleted?) in a heat-map shown in Figure 6. The highest frequency is related to deletion of a piece of code without stating any particular reason, which happens because the commit messages are not very informative. Also, the deletion of unused or unneeded functionality happened frequently.

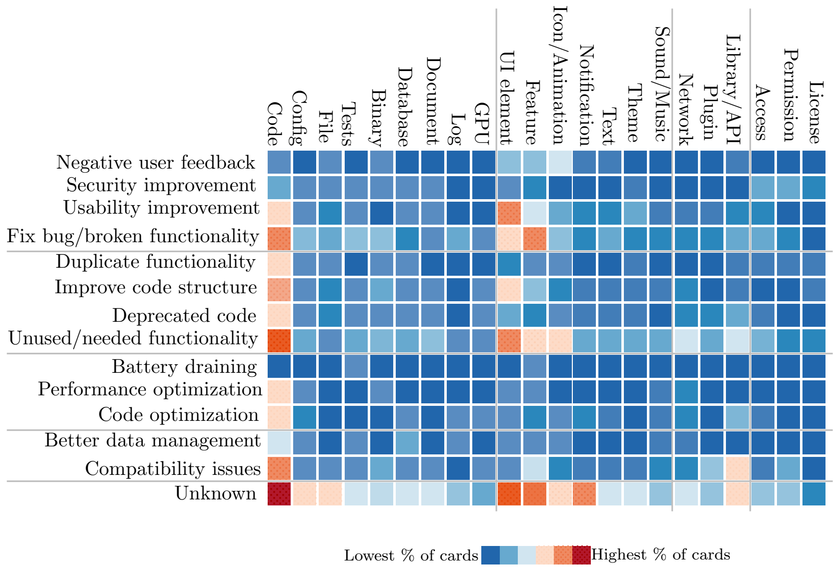

Figure 6: Heatmap showing what is deleted (columns) and why it is deleted (rows) as the result of analyzing 213,866 commit messages.

Automation potential: To classify all the 22,736 commit messages related to functionality deletion, we experimented with Naive Bayes, SVM, and Random Forest classifiers, frequently used in academia. We used our manual card sorting classification to train these classifiers. We found that Random Forest performed best with $F1\ score = 0.83$. However, the percentage of cards in each taxonomy categories remained almost the same. Hence, we only included the percentages of cards from manual classification in Table 2 and Table 3.

Instead, we further evaluate these taxonomies with developers in **RQ2**.

3 HOW DO DEVELOPERS VIEW FUNCTIONALITY DELETION? - A SURVEY

For the purpose of evaluating the results of **RQ1** and for better understanding the role of deletion in software evolution, we performed a survey to get the perspective of developers on our extracted taxonomies. We present the questions and results of this survey along with the results of **RQ2** in the next section.

3.1 Survey design

We designed a survey with 12 questions, five of them related to subject classification. We used convenient sampling [12] for this survey. We invited 130 mobile app developers that we knew from our personal contacts to respond to our survey. These developers have been working in 61 different companies at the time of the survey. Finally, 114 developers participated in the survey (response rate = 87%). Eight responses were incomplete, and we took them out of our analysis. The majority of these developers (67.0%) had more than three years of software development experience. In terms of mobile app development, the majority (53.8%) had more than three years of experience and 81.1% of them developed more than three apps, at the time of the survey.

We performed a survey with the objective to:

(1) Understand the frequency and extent of deletions;
(2) Evaluate and complete our results from analysis of commit messages (what, how, and why functionality is deleted);
(3) Grab developers opinion on if the extracted items in the "what" taxonomy relates to functionality deletion or not;
(4) Understand if and to what extent deleting functionality is decided in advance and planned, compared to adding a new functionality or fixing bugs;
(5) Explore the trade off between excluding a functionality versus fixing or improving it; and
(6) Understand the role of users in triggering functionality deletions.

3.2 Survey results

106 developers participated in our survey. The majority of these developers had more than three years experience in mobile app development (53.8%) and developed more than three apps (81.1%). The detailed information on the experience of the participants can be found in Table 4, Q1 to Q3. We first validated with them the taxonomies we mined in **RQ1** and then asked how they decided on functionality deletion and compared the importance of functionality growth (Lehman's law) with functionality deletion. We presented the results of evaluating taxonomies along with card sorting results in Table 2 and Table 3. Further details of the survey results are presented in Table 4.

How do developers delete functionality? We presented all six categories of deletions we mined in **RQ1** to developers. We asked them to "*Select all the actions that may result in eliminating a functionality from apps.*". 100% of the developers selected "removed", "reverted", and "temporary commented". 95.2% also selected "refactored". On the other side, 7.5% of developers (8 out of 106) and 12.2% of developers (13 out of 106) did not consider "moved" and "replaced" as functionality deletion, respectively.

What do developers delete? We presented the 22 categories of the low-level taxonomy as mined in **RQ1** (see Table 2) and asked developers to:

"*Distribute 100 points considering the extent and frequency of deletion for each of the below elements using your experience. Assign zero to the category if you do not consider it as functionality OR you never have deleted that element.*"

Table 4: Results of the survey with 106 app developers. Questions marked with * were open questions.

Demographics

	Q1: General development experience	**Q2:** App development experience
Less than a year	5.7%	14.2%
Btw. one and three years	27.4%	32.1%
More than three years	67.0%	53.8%

	Q3: Number of developed apps
One app only	4.7%
Two to three apps	14.2%
More than three apps	81.1%

Q4: To what extent you decide in advance (plan) for the below tasks?

	Never	Rarely	Sometimes	Often	Always
Adding functionality	2.8%	3.8%	18.9%	42.5%	32.1%
Fixing bug/broken functionality	4.7%	9.4%	24.5%	27.4%	34%
Deleting functionality	8.5%	14.25	19.8%	33%	24.5%

Q5: Deleting functionality from apps is .... adding functionality.

Less important than	21.7%
As important as	48.1%
More important than	30.2%

Q6: In what cases did you decide to delete a functionality instead of fixing it?*

Complex, Time consuming and hard to fix	75.5%
Function is unneeded, unused, or unnecessary	71.7%
Negative reviews on the broken functionality	54.7%
Incompatibility with 3^{rd} parties	19.8%
Fix needs functionality recreation	17.0%
Alternative functionality available	13.2%
Broken functionality cascades other errors	4.7%

Q7: Which type of user reviews provoke you to remove functionality?*

Annoyed reviews	86.9%
Repeated by different users	83.7%
Associated with low rating	81.6%
Precise reporting	46.6%
Replicable crashes	12.7%
UI and UX related	7.4%
Reviews associated with uninstalls	3.7%
Asking for refund	3.2%

Developers could add categories to complete our extracted taxonomy. We presented the average score for each element among all the 106 developers' responses in Table 2. Three developers added "paywall" and "advertisement". They stated that they delete advertisement or paywalls with the average frequency of 0.09%.

In our survey, we have asked the developers to score an artifact according to its perceived degree of membership to *functionality deletion*. The *Avg. % of opposed developers* in Table 2 shows the percentage of developers stated that the artifact in our mined "what" taxonomy does not relate to functionality deletion. Considering the scores per developer and artifact, we showed the extent of disagreement between developers about the association of each artifact to the *what?* taxonomy. "License", "test", and "log" are the top three most controversial artifacts between developers in that taxonomy.

Why do developers delete functionality? We asked developers to evaluate the low level taxonomy of Table 3:

"Based on your experience, distribute 100 points among the possible reasons for deleting apps' functionality."

We presented the average of the points for each reason in Table 3. Developers could also define new categories and assign points to them. Seven developers pointed to the "size of the update (release)" with the average score of 8.5% among the seven developers. Also, one developer pointed to *"solving technical debt"* as a reason for functionality deletion.

Developers stated that 41.97% functionality deletions are related to UI elements, and 44.79% of deletions have the purpose of improving users' experience.

If and to what extent do developers plan for deletion? We asked developers to state how often they decide in advance about deleting functionality versus adding functionality and fixing a functionality (Q4 - Table 4). We also asked them to state how important this planning is in comparison to planning for adding functionality (Q5 - Table 4). 57.5% of developers usually plan for deleting functionality, 19.8% sometimes plan for it, and the rest never or rarely plan for it. However, adding a feature is decided in advance for 74.6% of cases, and this number of bug fixes and maintenance tasks is 61.4%. While functionality deletion has been planned less in comparison to adding functionality or fixing a bug, the majority of developers still *plan* for deleting functionality.

Mostly, developers plan for functionality deletions.

Besides, 78.3% of developers believe that planning functionality is as important or even more important than adding functionality to an app (Q5 - Table 4). They supported their opinion based on (i) the effort and resources that have been put on developing a functionality, (ii) users and reaction, and (iii) opinion of their teammates. 45.2% of developers did not state any reason to support their selection on this question.

Developers do not plan for functionality deletions as often as addition and fixing it, but they believe planning for deletion is as important or more important than additions.

Do developers delete a functionality instead of fixing it? In an open question, we asked developers how they decide if they should fix a broken functionality versus deleting it. Each developer could state several reasons. Two of the authors categorized the results and extracted themes using open card sorting. We agreed on seven themes as presented in Q6 - Table 4. 75.5% of developers mentioned the complexity, time and effort needed for fixing the functionality as the main reason. Usage of the functionality was the second driving factor with 71.1% of developers considering it. Repetitive and extremely negative reviews are the third trigger (with 54.7%) for developers. As one of the developers said:

"When a belief about a crash or bug goes viral, we usually shut it (down) by removing the functionality, whether the report is correct or not."

Developers delete a broken functionality instead of fixing it mainly because of maintenance cost and complexity.

What type of user reviews intrigue developers to delete a functionality? Developers stated several reasons for this question. By analyzing these results, we identified seven reasons. The annoyed and angry reviews were the most important reason to consider deletion with 86.9% of participants mentioning it. Repeated blames by different users and the reviews which are associated with a low rating are the second and third reasons for deleting functionality (83.7% and 81.6% of participants pointing to it resp.). As one of the developers pointed:

"In case of frequent and angry feedback about certain annoying functionality and roughly above 30% app uninstalls, we remove the functionality."

Table 5: Degree of conformance between developers. % of opposed developers shows the number of developers assigning zero to an artifact in low level taxonomy of Table 2.

Low level taxonomy	% of opposed developers	Rank of controversy
Audio	0	–
Themes/Background	0	–
Text	3.77	
Notification	0	–
Image/icon/animation	0	–
Feature	0	–
Other UI elements	0	–
GPU	0	–
Log	29.2	3
Document	11.3	5
Database	7.5	6
Binary	4.7	7
Test	35.8	2
File	15.1	4
Config	0	–
Code	0	–
License	39.6	1
Permission	0	–
Accounts and access	0	–
Library/API	0	–
Plugin	0	–
Network/web	0	–

Multiple factors are involved in making a decision about functionality deletions. Complexity and needed maintenance effort, extent of usage, and user reviews with specific attributes are the top three most important factors.

4 DISCUSSION

Evolution of functionality is an integral part of software evolution. Software development includes both addition and deletion of functionality. This study is the first attempt to structure the broad spectrum of functionality deletion. So far, functionality deletion has been seen [17] as ranging from *no code being deleted* to *the whole software product will be removed.* Our preliminary study showed that there is a high probability that functionality would be deleted from a mobile app. We found that 98.8% of apps had decreased their size at least once over their lifetime. 61.3% of these apps had at least one release with more than 10% reduction in size compared to their previous release. To understand the nature of functionality deletions, we examined over 20,000 consecutive releases of more than 1,500 Android applications. The analysis of commit messages resulted in three taxonomies that we later validated with app developers.

What was deleted? It is hard to come up with an unambiguous short definition for software functionality deletion, as the term is broad and interpreted in different ways. In our *What?* taxonomy we identified 22 software artifacts. We cross-validated the mined taxonomy with app developers to show the accountability of our results. Some of the artifacts in this taxonomy (such as “file”) may appear surprising for some readers. However, as an exploratory case study, our goal was to find out the state of the practice, searching for new insights and creating ideas and hypotheses for new research [29]. We did not prune any artifacts from the taxonomy. Instead, to interpret the results we rely on the percentages and the degree of conformance by experts as presented in Table 5. As expected there are contradictory ideas which call for cautious interpretation of our results.

Why was it deleted? Commit messages did not reveal the clear reason for the majority of deletion activities. In the deletion distribution heat-map (Figure 6), the most frequent item is the code that has been deleted for an unknown cause. This is an indication of insufficient documentation for the majority of the analyzed commit messages. For the rest of commit messages, our analysis of commit messages showed that 29.98% of functionality deletions are related to UI elements and 11.27% of functionality deletions are intended to improve users’ experience. Our survey with the app developers confirmed the importance of users’ feedback and experience. This cannot be generalized without further investigation to other software applications.

How was it deleted? The results of the *How* taxonomy created the strongest controversy to our perception and the understanding of the surveyed developers. There were disagreements on whether “re-factoring”, “moving” and “replacing” are activities that indeed result in functionality deletion. We reported these disagreements along with the taxonomy.

We see our taxonomies as the comprehensive anatomy of functionality deletions which can be adapted and contextualized [6]. Understanding the domain is the first step toward decision support in the process of guiding the evolution of software functionality. The results of our survey with developers showed that developers plan for software deletion as well as additions. Moreover, for the majority of the participants, deletion is at least as important as adding functionality. The first step toward assisting developers with deletion decision is understanding the nature of these deletions and possible root causes as we did in this paper.
So far, release planning in general [28] and in particular for mobile apps [15, 34] is exclusively focused on feature addition. Planning in consideration of both addition and deletion of functionality requires revisiting the planning objective(s). Clearly, deletion consumes development effort as well. This is important for planning and making decisions for the next release(s). The exponential growth of maintenance effort independence of the number of function points (as a measure of the functional size of software) was for example confirmed by

5 THREATS TO VALIDITY

Throughout the different steps of the process, there are various threats to the validity of our achieved results. We discuss them in the sequel:

Are we measuring the right things? We performed a combination of machine learning and manual analysis to identify “functionality deletion commits” as the non-trivial deletions of the code. There is a threat that this set included other deletions as well and that we may miss some functionality deletions. We believe that our selection of commits was broad enough to cover the existing categories. However, the analysis of commits should be interpreted along with the developers’ survey results to fairly reflect functionality deletion. In the survey of **RQ2** with app developers, only two other categories were identified. We used half of the cards related to functionality deletion for card sorting in **RQ1** and only two of the commits were not related to functionality deletion. As a result, the threat of having irrelevant commits is low.

Another *construct validity* threat relates to the creation of the two taxonomies. Finding “fully correct” taxonomies is inherently difficult. We applied a hybrid approach trying to combine the strengths of both open and closed card sorting. The high Fleiss’ Kappa value showed conformance between developers. Having this along with the results of the survey showed that we created “good enough” taxonomies.

Are we drawing the right conclusions about treatment and outcome relation? We contacted developers from our personal contacts whom they have a considerable amount of contributions and years of experience. 106 developers participated in the survey of **RQ2** with a response rate of 87%. In comparison to studies in the context of mobile apps, this is considered high participation. Nevertheless, we selected developers by convenient sampling, which might bias the conclusions drawn.

Can we be sure that the treatment indeed caused the outcome? Our findings in **RQ1** and **RQ2** are the result of the procedure that we described in detail. We designed our protocol considering the broad definition existing in the literature [17]. While the results sound controversial in some cases, we believe this in particular, makes the results interesting and triggers further studies. We kept our position unbiased and did not prune any part of results from mining. Using app developers to validate the results of our mining and presenting the outcome of the survey minimized the existence of this type of threat to validity.

Can the results be generalized beyond the scope of this study? Our retrospective analysis was performed on open source mobile apps. The number of apps, reviews, and commits analyzed is considered high, indicating that results are significant at least for open source mobile apps.

6 RELATED WORK

In this study, we challenged Lehman's law of growth by investigating functionality deletion as a specific activity in the development process. Software evolution has been the subject of several studies, but *deletions* have been investigated only in few cases such as [16]. We focused on the mobile apps because the device resources are limited and the size of the release has been introduced as a decisive factor for release decisions [20]. Feature and functionality deletion for software products, in general, have been discussed mostly on the model level which triggered us to widely investigate the nature and reasons of functionality deletion in **RQ1** and **RQ2**. We discuss the literature on most related works on functionality deletion in Section 6.1. We briefly discuss the existing taxonomies in software engineering and the taxonomies that exist for mobile apps in Section 6.2.

6.1 Code Deletion in Software Engineering

Development activities have been discussed in software engineering as addition, deletion, and modification [26, 30]. However, addition and modification have been discussed more in comparison to deletion. Adding functionality is the scope of release planning. All the different approaches developed for that just consider addition [10, 28, 31] or applying change requests [2, 34]. Murphy-Hill et al. [17] differentiated between addition and deletion in the context of a bug fix and considered *functionality deletion* as the extent of a feature that is removed during a bug fix. They found that 75% of their participated developers remove functionality to fix a bug, which is aligned with our findings of **RQ1**. The heatmap in Figure 6 shows that deletion of code or feature to fix a bug is a common reason for functionality removal. The number of studies used code churn specifically in the context of defect prediction [19, 37]. However, code churn is the number of added or deleted lines of code together and does not differentiate between them.

The results of **RQ1** showed that refactoring and moving a feature usually includes functionality deletion and the majority (but not all) of developers agreed with that. For refactoring multiple aspects such as mutual deletion of code [11] and moving strategies for methods and statements (move methods) [4, 32] to eliminate bad smells [14] are considered. While we found permissions and licenses in our taxonomy, Calciati and Gorla [7] analyzed the evolution of user permissions in mobile apps and showed that permissions were rarely removed. In the process of permission evolution, Wei et al. also considered replacement of permissions as removing [35].

6.2 Taxonomies in Software Engineering

Usman et al. [33] performed a systematic literature review and analyzed 270 papers to describe the state of the art research in software engineering taxonomies. They found that most studies rely on a qualitative procedure to classify the subjects. They found that 33.9% of taxonomies did not demonstrate any usability. Only 19.58% of these studies showed the usefulness of their taxonomies by case study, survey, or experimentation. We used both the survey in **RQ2**. We used card sorting on 8,000 commit messages to extract these hierarchies [36]. Similar to us, 96.68% of the studies obtain taxonomies based on qualitative methods. However, 83.76% of them do not provide an explicit description of their procedure which motivated us to provide details about our study.

In the context of mobile app reviews, Pagano and Maalej [23] defined categories of user reviews by manually coding 1,100 reviews. Panichella et al. [24] introduced a taxonomy of user reviews related to app maintenance. Three authors manually inspected 300 emails within two software project mailing lists and mapped the extracted taxonomy to the user review categories defined by Pagano and Maalej [23]. Di Sorbo et al. [8] defined an intention based and topic-based categories of user reviews. Two of the paper authors analyzed 438 reviews manually and defined a two-level taxonomy; they used a classifier to categorize 952 reviews. Pathak et al. also provided a taxonomy for bugs related to energy consumption [18, 25].

7 CONCLUSION

Lehman's law on continuous growth of functionality does not universally apply. In the domain of mobile apps, developers frequently delete functionality—be it to fix bugs, to maintain compatibility, or to improve the user experience. This study confirms these trends by analyzing the evolution of 1,519 open source mobile apps, including top apps such as Firefox and Wikipedia; and by surveying 106 app developers.

This is the first study to investigate the phenomenon of functionality deletion in software evolution, and one of the first studies to examine app evolution over time. It opens the door towards a better understanding of software evolution, in particular in an important domain such as app development. In the days of Lehman's studies, features such as user experience, screen space, or energy consumption were not as crucial as they are today; it may be time to revisit and refine Lehman's findings.

Our future work will focus on involving *app users* to confirm the potential value of deletions also from their perspective in terms of improved usability and performance. We need to further explore the definition of functionality. More comprehensive empirical evaluation and analytical work are needed for that. The extension of existing mobile app release planning models towards consideration of platform mediation [22] and limited resources should be considered. For deciding on the functionality of evolving apps, we also target to perform a trade-off analysis balancing maintenance effort, usability, and functionality deletions. Overall, the main goal of our future research will be to better understand today's software evolution, especially for apps, and including deletion of functionality as a natural part of its evolution.

ACKNOWLEDGMENT

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada, NSERC Discovery Grant RGPIN-2017-03948.

REFERENCES

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471.

[2] David Ameller, Carles Farré, Xavier Franch, and Guillem Rufian. 2016. A Survey on Software Release Planning Models. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*. Springer, 48–65.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting behavior anomalies in graphical user interfaces. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 201–203.

[4] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694.

[5] Andrew Begel and Thomas Zimmermann. 2014. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 12–23.

[6] Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. 2017. The case for context-driven software engineering research: Generalizability is overrated. *IEEE Software* 34, 5 (2017), 72–75.

[7] Paolo Calciati and Alessandra Gorla. 2017. How do apps evolve in their permission requests?: a preliminary study. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 37–41.

[8] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Junji Shimagaki, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2016. What would users change in my app? Summarizing app reviews for recommending software changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 499–510.

[9] Jean Dickinson Gibbons and Subhabrata Chakraborti. 2011. *Nonparametric statistical inference*. Springer.

[10] Des Greer and Guenther Ruhe. 2004. Software release planning: an evolutionary and iterative approach. *Information and software technology* 46, 4 (2004), 243–253.

[11] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. 2007. Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 10–18.

[12] Barbara Kitchenham and Shari Lawrence Pfleeger. 2002. Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Software Engineering Notes* 27, 5 (2002), 17–20.

[13] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology*. Springer, 108–124.

[14] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 381–384.

[15] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering* (2016), 1–1.

[16] Audris Mockus, Roy T Fielding, and James Herbsleb. 2000. A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering*. Acm, 263–272.

[17] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2013. The design of bug fixes. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 332–341.

[18] Meiyappan Nagappan and Emad Shihab. 2016. Future trends in software engineering research for mobile apps. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 5. IEEE, 21–32.

[19] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 284–292.

[20] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. Release Practices for Mobile Apps–What do Users and Developers Think?. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 552–562.

[21] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. 2016. Analysis of marketed versus not-marketed mobile app releases. In *Proceedings of the 4th International Workshop on Release Engineering*. ACM, 1–4.

[22] Maleknaz Nayebi, Homayoon Farrahi, and Guenther Ruhe. 2017. Which Version Should be Released to App Store?. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 324–333.

[23] Dennis Pagano and Walid Maalej. 2013. User feedback in the appstore: An empirical study. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*. IEEE, 125–134.

[24] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can I improve my app? Classifying user reviews for software maintenance and evolution. In *Software maintenance and evolution (ICSME), 2015 IEEE international conference on*. IEEE, 281–290.

[25] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. 2011. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 1–6.

[26] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2015. The uniqueness of changes: Characteristics and applications. In *Mining Software Repositories (MSR), 2015*. IEEE, 34–44.

[27] Gordon Rugg and Peter McGeorge. 1997. The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems* 14, 2 (1997), 80–93.

[28] Günther Ruhe. 2010. *Product release planning: methods, tools and applications*. CRC Press.

[29] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[30] Emad Shihab, Christian Bird, and Thomas Zimmermann. 2012. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 301–310.

[31] Mikael Svahnberg, Tony Gorschek, Robert Feldt, Richard Torkar, Saad Bin Saleem, and Muhammad Usman Shafique. 2010. A systematic review on strategic release planning models. *Information and software technology* 52, 3 (2010), 237–248.

[32] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[33] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology* (2017).

[34] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. 2016. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 14–24.

[35] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 31–40.

[36] Thomas Zimmermann. 2016. Card sorting: From text to themes. In *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 137–141.

[37] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*. IEEE Computer Society, 9.