Exploring the Prevalence and Evolution of Android Concerns: A Community Viewpoint

Sherlock A. Licorish*
Department of Information Science, University of Otago, PO Box 56, Dunedin 9054, New Zealand.

Manuscript submitted May 3, 2016; accepted August 12, 2016.
* Corresponding author. Tel.: 64 3 479 8319; email: sherlock.licorish@otago.ac.nz
doi: 10.17706/jsw.11.9.848-869

Abstract: In line with growing awareness of the need for systems to adapt quickly to change, there has been increasing interest in the evolution of software systems. Research has particularly considered developer-led activities change over time. Comparatively less consideration has been given to the study of software evolution as driven by the wider community of stakeholders. Although, a project's wider community is central to the feedback system and project success. We have contributed to such efforts and studied the evolution of architecture issues and non-functional requirements in the Android project, as identified by the wider Android community[1]. We mined the Android issues tracker, employing n-gram analysis in our examination of 21,547 issues. We observe that most architecture-related issues were located in Android application layer, and these issues increased with time. Additionally, usability-related concerns were reported most when they were held to be given greatest attention. Outcomes suggests that Android's open model and shared ownership have positively impacted Google's success, which could provide a useful lesson for other similar projects.

Key words: Android, android architecture, android non-functional requirements, software evolution, data mining and N-gram analysis.

1. Introduction

A large body of research has been directed to understanding various aspects of the software development process (as performed by humans), and particularly, how systems evolve. Given the challenges associated with gaining access to live teams and the extended user community, these works have tended to use repositories of software artefacts, e.g., change logs, bug and issue tracking systems, mailing list and blogs [1], focusing on within-project activities. For instance, using repository data Asaduzzaman *et al*. [2] studied the specific developer changes that introduce bugs in Android over time and found that larger code changes were likely to result in more defects. Artefacts were used by Thomas *et al*. [3] who applied topic modelling to study the evolution of JHotDraw and jEdit and found correlation between the evolution of various topics and developers' code changes. Zaidman *et al*. [4] also used artefacts and the TeMo tool to evaluate the co-evolution of test and production code.

From the examples noted here it can indeed be seen that, in the use of repositories to study the way software systems evolve there has been a general tendency to examine within-project phenomena. Accordingly, in terms of topic areas, among the subjects that were previously examined, and particularly

[1] Companies such as HTC, Samsung, LG, Sony, Sprint, Verizon, T-Mobile, AT&T, users of Android devices, other developer groups and Google.

through the use of data mining methods, researchers have considered the work of developers and the relationships between that work and evolutionary code coupling [5], defect classification [6], origin analysis and refactoring [7], software reuse [8], project communication [9] and team's behavioral processes [10]-[16]. More recent efforts have looked at understanding post release user's feedback [17]-[23]. In fact, research effort has also been dedicated towards providing a theory of software evolution [24], [25].

A much lesser degree of emphasis has been placed on investigating how systems evolve, as driven by the wider community of stakeholders, and particularly, how community members identify and report issues over time. This is despite the long held acknowledgement of these members' role as change agents and their relevance in projects' success [26]. In fact, beyond software project success, evidence has also shown that community members' reviews, which are often influenced by their experience while using a product, influence other individuals' decision to (not)use a product [27]. Issues encountered during product use affect stakeholders' experiences, and their reviews [20]. We have thus studied the evidence in issues logged about the Android operating system (OS) in this work to contribute to the study of software systems evolution, examining the evolution of issues in Android's architecture and non-functional requirements, areas central to the validation of software quality.

The Android OS has become the most widely used mobile OS [28], and so, insights into this OS evolution could be useful for honing and evolving the success of other software projects. In fact, studying the feedback of the Android community, as delivered through the identification of defects and issues, could inform within-project investigations as conducted previously [2], [4], and particularly, those related to bug prediction [29]. Such investigations could also provide insights into the impact of particular defect issues on the community. For instance, in planning remedial work on the Android platform, issues uncovered in the OS libraries may be perceived by Google to be less critical than those that are detected in the application layer. This viewpoint may be particularly relevant if the prevalence of bugs in a particular layer may have a negative impact on many members' satisfaction. Thus, if prediction models are able to accurately forecast specifically where and how severe issues are likely to be (e.g., based on the Android application architecture), such work may provide incentives to the software development community [2].

Of course, in general, before mining, modelling and predicting the incidence of software issues, it would be prudent to determine the prevalence of specific issues, as this knowledge may inform the feasibility of conducting prediction work at all [30], and should validate how reliable predictive models are likely to be. Should it be uncovered that most issues arise at the Application layer of an OS, for instance, then appropriate action may be taken to reduce these concerns.

We look to determine the prevalence and evolution of specific Android issues by mining the Android OS issue tracker. We applied the n-gram analysis technique to the study of pre-processed issues, and explored the evolution of two areas of Android: the system architecture and non-functional requirements, which have been assessed as central to systems quality and success. Building on previous works [31], [32], our contributions are twofold. We provide insights and recommendation for stakeholders of the Android community, and we assess our findings in relation to the laws of software evolution and outline suggestions for further research.

In the next section we present the study's background and motivation, and outline our specific research direction. We then describe our research setting in Section 3, introducing our analysis techniques and measures in this section. In Section 4 we present our results, and in Section 5 we discuss our findings. We next consider the limitations to our study in Section 6. Finally, we outline the implications of our results in Section 7 and highlight directions for future research.

2. Background and Motivation

We provide our background and motivation in this section. Firstly, in grounding our investigation, we consider the theory of software evolution in Section 2.1. We then examine the two areas under consideration for Android, architecture and NFR, in Section 2.2.

2.1. Theory of Software Evolution

Over four decades of research efforts have been dedicated to the study of software evolution, in part with the intent of delivering a theory. Lehman et al.'s work has been at the forefront of these efforts, and culminated in the laws of software evolution [24], [25], various metrics for studying software evolution [33] and a theory of software evolution [34]. In general, Lehman and his colleagues noted that software systems growth followed a smooth long range trend, with superimposed ripple and a region of instability or chaotic behavior [34] – this pattern has been replicated in the study of many systems of varying application areas, sizes and organizational domains over the last four decades [35]. Notwithstanding differences in software development domain characteristics (e.g., commercial versus open source developments), and the potential lack of generalizability of these outputs, independent research has indeed found tenets of Lehman *et al.'s* [35] classification scheme to be relevant for explaining software evolution [36].

While, as noted above, metrics used to study software development address issues that originate inside the development team (e.g., module size, modules added, modules changed, modules deleted, effort applied, developers' messages and commits), metrics generated by the wider community of users (e.g., bug reports or issues) are also held to be necessary for studying software evolution [36]. In fact, beyond Lehman's [24] model of software growth trend, Lehman *et al.'s* [35] eight laws of software evolution emphasize refinement as a major aspect of software systems growth. This issue, refinement, is captured under laws VII and VIII which may be assessed as user-driven. For instance, Law VII: *The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes*, and Law VIII: *E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement for other than the most primitive process*. These members (the wider community of users) interact with software systems to unearth bugs so as to maintain system quality [37]. Community-wide feedback, whether via their reporting of bugs, submission of enhancement requests through an issue tracker, or approving software requirements, also contributes to software improvements.

Given a project's community integral role in the feedback process, and particularly in relation to software functionality and quality, much could be learned about the prevalence and evolution of specific issues as reported by the community. Such outcomes may also be related more generally to previous finding in the software evolution space. We therefore studied two areas of Android OS issues evolution (architecture and NFR). These are considered in the subsequent section.

2.2. Android OS

We review literature around the two areas under consideration in this section. Firstly, we consider works related to Android software architecture in Section 2.2.1. We then consider literature around NFR is Section 2.2.2.

2.2.1. Software architecture

The Android OS is built using Java, C, C++ and XML. At the core of the Android OS stack is a modified Linux 2.6 monolithic kernel, with java applications running on a virtual machine [28] (see also: http://developer.android.com/about/index.html). Android OS modules are generally arranged in a layered structure, with the Application Layer at the top, followed by the Application Framework layer, Libraries and the Linux kernel (refer to Butler [28] for illustration). Among the software programs that are shipped as

part of the Android OS, Contacts application, Email client, Web and Map browsers and Messaging application are most frequently included. Multiple handset vendors collaborate with Google through the Open Handset Alliance (OHA), extending these applications (i.e., how features appear), and the basic Android OS, to suit their hardware offerings. Companies such as HTC, Samsung, LG and Sony are among the device manufactures that offer Android phones, while Sprint, Verizon, T-Mobile and AT&T offer services for Android devices [28]. These communities, along with user groups, other developer groups and Google use the Android OS issue tracker to report issues and request enhancement to features. Thus, the Android issue tracker provides the interface between the Android OS (as the product), the producers of the product (the developer community) and the consumers of the product (app developers and regular end-users of Android devices).

While the Android issue tracker has been the subject of research, previous work has either considered a small subset of the issues, or has focused on a specific subset of architectural issues. For instance, Kumar Maji et al. [38] studied 758 bugs from four early versions of the Android OS (versions 1.1, 1.5, 1.6 and 2.0) and found most bugs to be present in the application layer. Guana *et al.* [31] classified 8,597 Android OS bugs (of 20,169 total bugs) in the last four layers of this OS, omitting those that were suspected to be in the application layer. Contrary to the findings of Kumar Maji *et al.'s* [38] study considering a much smaller sample of bugs, they found higher bug concentration in the framework and kernel layers. Guana *et al*. [31] suggested that this prevalence of bugs closer to the OS kernel may be linked to hardware compatibility issues. This notion may be challenged however; as evidence has shown that users of mobile devices focus most on web browsing and multimedia applications (for example: http://androidforums.com/nexus-5). Similarly, those involved with application development for the Android community are likely to be most concerned with how the application framework (e.g., the Location and Telephony managers for a GPS app with voice and audio features) and libraries (e.g., SQLite for data storage in a multiplayer video game that is maintaining players state) work, and so, may discover more defects in these OS segments.

Accordingly, investigating issues in the application layer of Android OS architecture would extend previous work and provide the opportunity for us to understand the real scale of issues as present in the five layers. In fact, apart from Kumar Maji et al.'s [38] small study, we are unaware of any other study dedicated towards understanding how Android architecture issues have evolved and shifted over time. Such insights would be useful for the Android community (companies, user groups, developer groups and Google), and may also provide strategic direction(s) for the project's stakeholders. For instance, information on the prominence of specific issues, how these issues evolved and how they were remedied over time would help to focus specific hiring strategies for Google, other device vendors, and maybe, the Apps development community. Such understandings may also provide useful pointers for the mobile OS community more widely. We therefore outline our first research question:

RQ1. (a) How Android architecture issues as logged on the issue tracker were distributed in the five OS layers, and (b) How the community concerns evolved from the OS initial release to its recent offerings?

2.2.2. Non-functional requirements (NFRs)

Given the high level of issues that were uncovered in the application layer of very early versions of the Android OS [38], validating if this pattern persisted in the later releases would provide useful insights into the community challenges. In particular, quality related problems (as captured via NFRs) may be especially unsettling for developers and end-users. These problems have been shown to become apparent in community members' expression of certain topic words (e.g., efficiency and usability) [39]. Thus, it would be insightful to explore how NFRs issues evolved, and the general prevalence of these issues in the Android community.

Previous work has long established that software projects are most successful when user feedback is

accommodated [40]. Therefore, apart from relating software release outcomes to established quality models (e.g., ISO-9126 – [41]), a NFRs-based enquiry could potentially explain specifically how NFRs aspects in the different Android OS layers affected the Android community. Furthermore, this insight would help us to understand how the Android community emphasis on software quality shifted over time. This awareness would also inform the current community priorities.

In fact, an approach to study the way NFRs issues change over time has been considered by previous work. Hindle et al. [32], for instance, explored the way NFRs changed over time and examined the difference in emphasis for such issues in MySQL and MaxDB systems. They found that NFRs issues decreased as the systems matured, and interests in such issues for these two systems were driven largely by external technology forces (e.g., the release of new software that needed integration into these products). Additionally, these authors found stronger interest in functionality aspects by MySQL developers, while the MaxDB developers were most interested in systems efficiency. Alipour et al. [42] also included NFRs words to aid with the detection of duplicate bugs. Hindle et al.'s [32] study, while being somewhat similar to the work that is undertaken here, only examined one version of each of two systems, and was aimed mostly at demonstrating how automated topic extraction techniques are relevant to understanding pass events in software projects. In our context we aim to understand how the Android stakeholders' quality-related concerns shifted over time:

RQ2. How have NFRs issues evolved in the Android OS?

3. Research Setting

We investigated more than six years of records in the Android OS issue tracker, comprising issues submitted by the Android community between January 2008 and the end of March 2014. Issues identified by the Android community are submitted to the Android OS issue tracker (refer to http://code.google.com/p/android/issues/list). The data that is stored for issues include: Issue ID, Type, Status, Owner, Summary description, Stars (number of people following the issue), Priority, Milestone, Attachments, Open date, Close date, Reporter, Reporter Role, Project, Component, and OS Version. We extracted 21,547 issues from the Android OS issue tracker that were reported between the dates noted above. These issues were imported into a Microsoft SQL database, and thereafter, we performed data cleaning by executing previously written scripts to remove all HTML tags and foreign characters [43] to avoid confounding our text analysis.

Through these techniques we observed that the issues were labelled by those adding them as defect (15,750 issues), enhancement (5,354 issues) and others (5 issues); and 438 issues had no type (being null). Issues had one of six statuses, being new (18,891 issues), assigned (2,001 issues), unassigned (476 issues), needsinfo (143 issues), accepted (32 issues), and resolvedbyuser (1 issue). Three issues also had null status. Issues had 140 different owners, with various numbers of stars and five priorities (low, medium, high, critical and blocker). There were 14,958 reporters of issues to the Android OS issue tracker, comprising mostly users (9,006 issues) and developers (7,804 issues); with some 4,737 issues having a null role. Issues were reported about 13 different components of the OS, however, for most of the issues reported this field was left blank (15,711 issues altogether). We observed that none of the issues had a Close date, suggesting that, although issues were addressed, developers did not invest effort in updating this field. This pattern was also noted in Kumar Maji et al. [38], whose study of 758 Android OS bugs revealed that some fields were not updated regularly. In fact, we observed that only 2,816 issues had the version field updated (out of the total 21,547 issues), while the others were left blank. Given this evidence, we did not perform extensive analyses on data columns with missing values. Of note also is that whether issues logged were recorded as defects or enhancements (or new features), we anticipate that the lodgment of issues denotes stakeholders'

need for added fulfilment, and the subsequent developers' extension of the system denotes an addition which aids in the system's evolution.

We examined the data associated with each issue in our database to correlate these with the official releases of the Android OS (refer to http://www.android.com). We observed that the Android OS's first commercial release was in September 2008, while the first issue was logged in the issue tracker in January 2008 (see: http://android-developers.blogspot.co.nz/2008/09/announcing-android-10-sdk-release-1.html). This suggests that the community was already actively engaged with the Android OS after the release of the first beta version in November 2007, with issues being reported two months after the first beta release (see: http://android-developers.blogspot.be/2007/11/android-first-week.html). Given this pattern of active engagement and issue identification even before the official Android OS release, we therefore partitioned the database of issues based on the Android OS release date and major name change to perform our temporal analysis. So for instance, all of the issues from January 2008 (the date of the first issue that was entered) to February 2009 (the date of one of the Android releases before a major name change was made) were labelled as early versions, reflecting the period of the Android OS releases 1.0 and 1.1 (which were both deployed without formal names). The subsequent partition comprised the period between Android OS version 1.1 and Cupcake (Android version 1.5), and so on.

Table I provides a brief summary of the numbers of issues logged between each of the major releases, from the very first commercial release (and using the release date of the first beta version to compute the first entry) to Android OS's KitKat – Android version 4.4. In column three of Table I (Number of days between release) it is noted that the time taken between the delivery of most of Android OS's major releases (those involving a name change) was between 80 and 156 days, with only two formal releases (Gingerbread and Jelly Bean) falling outside of this range. The fourth column of Table I (Total issues logged) shows that the number of issue reported generally increased as development of the Android OS progressed, with this rise being more evident when the mean number of issues reported per day for each release is considered (refer to the values in the fifth column for details). Over the six years of Android OS's existence, on average, 9.6 issues were logged every day. We studied the details in these issues using text analysis techniques; these methods and the detail around how we operationalized the aforementioned issues are presented next.

3.1. N-Gram Analysis

Natural language processing (NLP) techniques are often used to explore and understand language usage within groups and societies, both from static and temporal perspectives. The motivation for investigating word use, for instance, is rooted in the notion that frequently used terms are often important and meaningful to those that utter them during their discourses [44, 45]. As a result, NLP approaches have attracted a significant amount of interest in the software engineering and mining repository communities. Such investigations are often focused on understanding various areas of software development, including detecting defects [46], extracting requirements [47] and source code analysis [48].

N-gram analysis, considered to be an NLP and computational linguistic method, is often recommended for analyzing large volumes of text to identify frequently occurring words usage patterns [49]. The n-gram is defined as a sequence of n words (or characters) in length that is extracted from a larger stream of elements [49]. Take for instance the statement: "wifi connection stopped working". This statement has four 1-grams in the context of the words "wifi", "connection", "stopped" and "working". A 2-gram analysis of this statement would result in "wifi connection", "connection stopped" and "stopped working". A similar pattern continues if 3, 4, 5, and so on, are substituted for n. (In the example above, considering that the statement has only four words, a 5-gram analysis (or larger) would not be feasible). Inherent in the example noted here is how progressive n-grams become useful in interpreting text. For instance, the words "wifi", "connection",

"stopped" and "working" on their own do not convey much information, perhaps apart from knowing that these words individually were of interest to the communicator. Considering the output from the 2-gram analysis - "wifi connection", "connection stopped" and "stopped working", each token (e.g., "connection stopped") is unearthing interesting information that could help us to piece together a clearer picture about a non-functional wifi connection. A 3-gram analysis would reveal further details (i.e., "wifi connection stopped" and "connection stopped working"), providing additional context that would likely aid in some form of remedial action or intervention, resulting from the availability of clearer contextual information. Thus, taken together, n-gram analysis can reveal interesting patterns in written text.

Table 1. Android OS Issues over the Major Releases

Version (Release)	Last release date	Number of days between release	Total issues logged	Mean issues per day
Early versions (1.0, 1.1)	09/02/2009	451	262*	0.6
Cupcake (1.5)	30/04/2009	80	101	1.3
Donut (1.6)	15/09/2009	138	266	1.9
Éclair (2.0, 2.01, 2.1)	12/01/2010	119	464	3.9
Froyo (2.2)	20/05/2010	128	490	3.8
Gingerbread (2.3, 2.37)	09/02/2011	265	1,291	4.9
Honeycomb (3.0, 3.1, 3.2)	15/07/2011	156	897	5.8
Ice Cream Sandwich (4.0, 4.03)	16/12/2011	154	1,127	7.3
Jelly Bean (4.1, 4.2, 4.3)	24/07/2013	586	12,148	20.7
KitKat (4.4)	31/10/2013	99	4,501	45.5
		$\sum = 2{,}176$	$\sum = 21{,}547$	$\bar{x} = 9.6$

* Total number of issues logged between the first beta release on 16/11/2007 and Android version 1.1 released on 09/02/2009

In fact, n-gram analysis techniques have been employed in knowledge mining very large volumes of text by major technology sector companies, including for example, Google (see: http://books.google.com/ngrams) and Microsoft (see: http://web-ngram.research.microsoft.com). Other researchers in software engineering have also used this approach in more granular works to study the shifting focus of topics (e.g., [50]).

We applied this technique to study the 21,547 extracted Android OS issues in our corpus. In order to check the suitability of the n-gram technique for analyzing the issues that were logged by the Android OS community we first randomly selected 15 issues logged between each of the major Android OS release in Table I for a pilot analysis (i.e., 150 issues). The outcome of this analysis revealed that for most of the issues that were logged (across versions), while there were numerous tokens for each list, tokens appeared infrequently, and particularly when those beyond 3-grams were considered (i.e., 4-grams, 5-grams, and so on). We therefore decided to stop at 3-grams, which we thought would represent the issues of most interest to the community [50]. We believe that this analysis helps us to study the meaningful issues (given the mass interest) that were reported by the Android OS community over time.

Accordingly, informed by insights from the pilot study, we converted all characters in our corpus to their lowercase equivalent to avoid n-grams duplication. For instance, "BUG", "Bug" and "bug" would all be treated as different terms and a 1-gram analysis would return three different tokens (as against one token with a count of three), had we performed the n-gram analysis with case sensitivity. Such an analysis would have skewed our results [50]. We analyze the Android OS issues that were logged between each of the major release in our corpus (refer to Table I for summary). In order to do this we used the open source Online NGram Analyzer (ONA) (http://guidetodatamining.com/ngramAnalyzer). During the analysis process punctuations were treated as part of the preceding token as these were not meaningful for our interpretation. Three sets of analyses were performed for each set of issues (1-gram, 2-gram and 3-gram). A

summary of the n-grams and tokens is provided in Table 2, which shows that there were 182,760 tokens altogether, resulting in 40,658 different 1-grams, 142,247 different 2-grams and 172,758 different 3-grams in our corpus (355,663 n-grams in total). The output from the ONA included the n-gram or token, count (number of times the token appeared) and frequency (in percentage). Given that issues were of varying lengths (number of words), the percentage value was used to normalize the n-grams across issues. We also used the number of followers to normalize our outcomes, on the basis that the user base of the feature would impact the number of issues uncovered (e.g., 50 users reporting a specific bug would probably be less significant in terms of revision than the same group of users reporting 50 different bugs) [51]. This normalized value allowed us to assess the relative use of the specific terms in issues across versions, and to perform meaningful comparisons of Android OS versions, regardless of the number of words used by various individuals in the community to describe issues, and the volume of followers specific issues attracted [51].

We analyze these n-grams to explore the way the Android OS architecture and NFRs issues affected the Android community over the life of the OS. We consider how each of these study areas was operationalized in the following two subsections.

Table 2. N-Gram Summary for Android OS Issues

Version (Release)	Total tokens (words)	1-grams	2-grams	3-grams
Early versions (1.0, 1.1)	2,090	1,123	1,960	2,069
Cupcake (1.5)	792	473	746	787
Donut (1.6)	2,245	1,016	2,078	2,228
Éclair (2.0, 2.01, 2.1)	3,481	1,787	3,282	3,461
Froyo (2.2)	4,456	1,414	3,412	3,816
Gingerbread (2.3, 2.37)	10,811	3,467	9,103	10,107
Honeycomb (3.0, 3.1, 3.2)	7,289	3,167	6,736	7,230
Ice Cream Sandwich (4.0, 4.03)	9,487	3,486	8,502	9,308
Jelly Bean (4.1, 4.2, 4.3)	104,132	17,894	77,393	97,536
KitKat (4.4)	37,977	6,831	29,035	36,216
∑	182,760	40,658	142,247	172,758

3.1.1. Classifying architecture issues

Android OS architecture issues were classified according to the Android architecture topic words. As noted in Section 2.2.1, Android OS modules are generally arranged in a layered structure, with the Application Layer at the top, followed by the Application Framework layer, Libraries, Android runtime and the Linux kernel. Specific keywords are contained in the Android architecture documents, for instance, the Android runtime layer has *Core libraries*, *Delvac* and *Virtual machine.* Similarly the Android Linux kernel layer is made up of the *Display driver*, *Camera driver*, *Flash memory driver*, *Binder (IPC) driver*, *Keypad driver*, *Wi-Fi driver*, *Audio driver* and *Power management*. We followed the approach used by Guana et al. [31], and mined the n-grams for the Android architecture topic words for each of the OS layers to assess the distribution and evolution of architecture issues in the OS. However, given that Guana et al. [31] had omitted issues that were suspected to be in the application layer during their work, we extended their classification to accommodate issues in the Android OS application layer. The following terms (and their plural forms) were used to classify issues in the application layer: *Home*, *Contacts*, *Phone*, *Browser*, *Map*, *Email*, *SMS/MMS/Message*, *Application/App*, *Calendar*, *Music/Album*, *Photo*, *Video/Movie*, *Home Screen*, *Game*, *Search*, *Call*, *GPS, Wifi* and *Bluetooth*.

3.1.2. Classifying non-functional requirements (NFRs) issues

Multiple taxonomies were previously designed for assessing NFRs and software product quality. While some of these taxonomies were developed to reflect the realities of a specific project [52], others also consider multiple software projects [53]. From a formal standard perspective, the ISO-9126 quality model describes six NFRs that are considered to be relevant to all projects [41]: efficiency, functionality, maintainability, portability, reliability and usability. Further, Boehm et al.'s model [54] has also been widely considered during the assessment of NFRs and software quality issues. In order to evaluate how NFRs issues evolved in the Android OS and to understand whether and how the community's priorities and focus on stakeholders' concerns shifted over time, we considered a combination of these models as used by Hindle et al. [32] to classify Android's NFRs related n-grams. In the taxonomy we used each of the six ISO-9126 terms (efficiency, functionality, maintainability, portability, reliability and usability), resulting in an extended list of words under each category. For instance, the words (and associated adjectives, nouns and verbs, e.g., maintain, maintainable and maintainability) *testable*, *changeable*, *analyzable*, *stable*, *maintainable*, *modular*, *modifiable*, *understandable*, *interdependent* and *dependable* were captured under maintainability.

3.2. Qualitative Analysis

We supplement our quantitative n-gram analysis with bottom up manual inspections where there were revealing results, and particularly in examining usability issues. In order to do this we used the issue as the unit of analysis. We performed selective coding on specific issues, before then engaging in constant comparison of the subsequent codes in order to provide a higher level of abstraction to facilitate further interpretation of aspects of our quantitative results [55]. This process first involved reading the selected issue and tagging the issue with a specific code (e.g., dissatisfied with feature or redundant feature), before grouping the identified codes and comparing these with all others in the group to then come up with a particular theme (e.g., feature unusable or feature should be removed). We present our results in the following section.

4. Results

We present our results in this section. Firstly, we provide the results for the distribution and evolution of Android OS architecture issues in Section 4.1, then those for the NFRs issues in Section 4.2. As a means of validation we revisit the outcomes of our earlier work [56] about the correlation of issues raised with features that were released by Google towards the end of Section 4.2.

4.1. Architecture Issues (RQ1)

We extracted n-grams in keeping with the classification procedure outlined in Section 3.1.1. The resulting percentage distribution of Android OS architecture issues is presented in Fig. 1 (values originate from the mean percentage n-gram for the ten major Android releases in Table I). Here it is shown that, overall, nearly 60% of the issues that were logged by the Android community over the lifetime of the OS related to the Android OS application layer, while the kernel and application framework layer had the second and third largest number of concerns (19.0% and 16.0% respectively). Fig. 1 reveals that far fewer issues were raised in regard to the libraries and Android runtime layers (only 4.8% and 0.3% respectively). These results confirm the findings of the smaller scale study of Kumar Maji *et al*. [38], and justify the need for further investigation of issues as evident in the Android OS application layer.

An ANOVA test was conducted in order to ascertain whether the differences observed in Fig. 1 were statistically significant. Our dependent variable, frequency of issues, was normally distributed for four of the five types of architecture issues in Fig. 1 as assessed by the Shapiro-Wilk test, with the exception being

the frequency of runtime issues, which was slightly skewed. The variance across groups was significantly different ($p < 0.01$) as assessed by Levene's test for equality of error variances. We observed significant differences between the frequency of architecture issues that were reported on the Android issue tracker, $F(4, 45) = 28.842$, $P < 0.01$. Tukey HSD pair-wise comparisons confirmed that there were significantly more issues reported for Android applications than the other four types ($p < 0.01$ for all comparisons with the other issues).

In keeping with our objective to understand how architecture issues evolved over the life of Android, we examined the prevalence of architecture issues (*overall*). Fig. 2 shows that the largest percentage of architectural concerns were generated between the release of Donut (version 1.6) and the final Gingerbread release (version 2.37), and between the final release of Ice Cream Sandwich (version 4.03) and Jelly Bean (version 4.3). In fact, if we use a proxy value of 10.0% for the mean percentage of Android system architecture issues (given our partitioning of the ten major release in Section 3), Fig. 2 demonstrates that between the final release of Ice Cream Sandwich (version 4.03) and Jelly Bean (version 4.3) there were actually twice (19.7%) as many issues. Additionally, between Donut (version 1.6) and Éclair's final release (version 2.1) there were nearly one and a half times (13.5%) the mean percentage of issues reported. Given this pattern of result we examined the actual n-grams (percentage values) to see how the five forms of architecture issues were distributed in the different releases in Fig. 3.

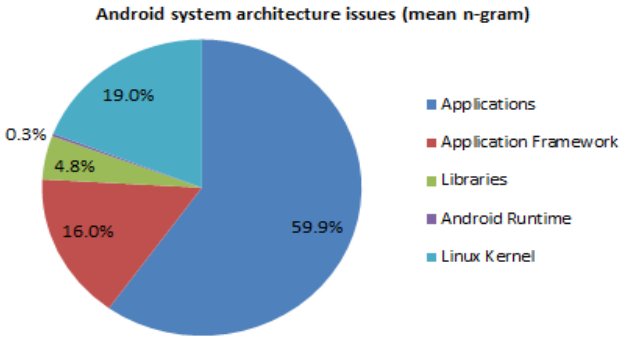

Fig. 1. Android OS architecture issues.

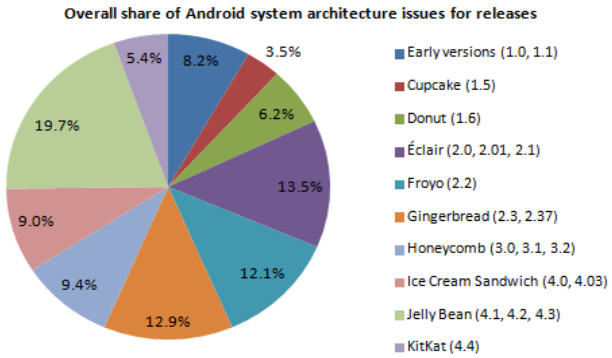

Fig. 2. Share of Android OS architecture issues over time.

Fig. 3 demonstrates that most issues were reported about the Android OS application layer across all releases. In addition, our n-gram analysis across releases confirmed that very few issues were reported about the Android runtime layer (0.3% overall in Fig. 1). Furthermore, in line with the pattern noted in Fig. 1, there were also substantial proportions of issues reported about the kernel and application framework layers across releases. With the exception of the releases between Froyo (version 2.2) and Gingerbread (version 2.37), and between Jelly Bean (version 4.3) and KitKat (version 4.4), there were on average more

kernel issues than application framework issues.

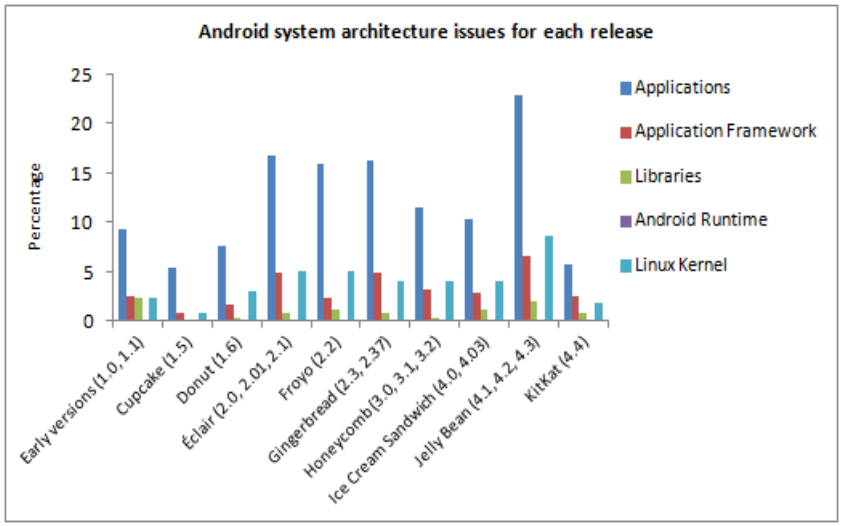

Fig. 3. Android OS architecture issues across releases.

Finally, we extracted the top 10 n-grams between four releases where the highest incidence of architecture issues were recorded (refer to Fig. 2) to probe the issues further. These measures are provided in Table III, which confirm the highest prevalence of issues in the application layer (though, issues may manifest themselves in the application layer, but their origin may be deeper). Between Éclair 2.01 and 2.1 Table III reveals that the top 10 issues were primarily application related. This pattern was maintained between Éclair 2.1 and Froyo 2.2. However, the Android community also expressed some dissatisfaction with how the OS was functioning on the "nexus" device in the latter three periods that were examined. The later releases, while maintaining the pattern of dissatisfaction with issues in the application layer (e.g., see the n-grams "sms", "app", "screen" and "browser" in the latter two release ranges in Table III), also generated concerns about how the OS was functioning on specific mobile devices. Table III shows for instance that between Jelly Bean 4.1 and 4.2, beyond the Nexus device, issues related to the Galaxy were also frequently reported.

Table 3. Top 10 n-Grams and Percentage Frequency for Four Selected Releases

Between Éclair 2.01 and 2.1		**Between Éclair 2.1 and Froyo 2.2**		**Between Gingerbread 2.3 and 2.37**		**Between Jelly Bean 4.1 and 4.2**	
email	2.0	**sms**	**1.1**	**sms**	**1.3**	android 4.1	1.6
app	**1.3**	calendar	0.7	android 2.3	1.1	nexus	0.8
calendar	0.8	phone	0.7	**app**	**0.7**	**app**	**0.7**
call	0.6	**screen**	**0.6**	phone	0.7	jelly bean	1.1
display	0.6	**app**	**0.6**	nexus	0.6	**screen**	**0.5**
browser	**0.6**	call	0.6	**browser**	**0.5**	galaxy	0.5
music	0.6	nexus	0.6	call	0.5	phone	0.4
screen	**0.6**	contacts	0.5	keyboard	0.5	**browser**	**0.3**
exchange	0.6	email	0.4	**screen**	**0.4**	**sms**	**0.3**
sms	**0.5**	**browser**	**0.4**	emulator	0.3	wifi	0.3

Bold values denote n-grams that appear in the top 10 list across all four releases

4.2. Non-functional Requirements (RQ2)

We performed similar analyses across the six NFRs issues (refer to Section 3.1.2 for description). High-level results are depicted in Fig. 4, which demonstrates that usability-related issues have dominated

the concerns of interest to the Android OS community (i.e., 86% of NFRs issues were usability related). Fig. 4 shows that 8.2% of the NFRs issues were functionality related, 2.6% were maintainability related and 2.9% were reliability related. The Android OS community expressed very few concerns about efficiency and portability. We conducted formal statistical testing to inspect these results for statistical significance, first examining the distributions for normality with the Shapiro-Wilk test. Findings revealed that only the distributions of functionality and usability issues were normal, the others violated normality ($p < 0.05$). We thus conducted a Kruskal-Wallis test which confirmed that there was a statistically significant difference between the frequency with which NFRs issues were raised by Android stakeholders ($H(5) = 34.582$, $P < 0.01$), with a mean rank of 18.6 for efficiency, 35.9 for functionality, 27.3 for maintainability, 18.1 for portability, 28.2 for reliability and 50.9 for usability. Pair-wise *post-hoc* comparisons at the Bonferroni adjusted level of 0.003 (i.e. 0.05 divided by 15 comparisons) confirmed that the Android community raised significantly more usability related issues than any other NFRs issues ($p < 0.003$ for all comparisons). None of the other pair-wise comparisons involving the other NFRs issues showed a significant difference.

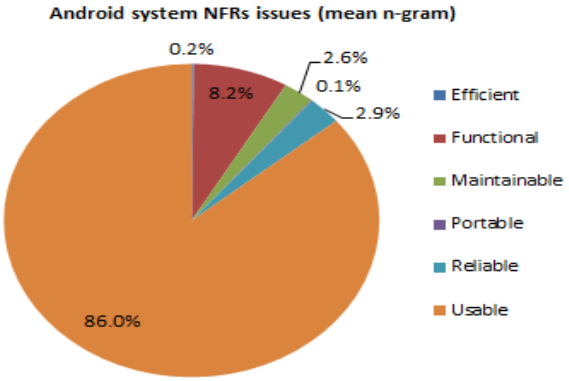

Fig. 4. Android OS NFRs issues.

We then examined the way NFRs concerns were raised across all releases and depict the results in Fig. 5. In using the mean percentage issue of 10% as a proxy for comparison (as done in Section 4.1), Fig. 5 shows that the highest proportion of NFRs issues were reported between the Froyo (version 2.2) and Jelly Bean (version 4.3) releases. (*Note* that the community was actively populating the Android issue tracker for the KitKat (version 4.4) release at the time of our data extraction, and so, the measures uncovered for this release may not represent a complete picture of the issues that are likely to be reported.) We plot the incidence of reported usability issues (given its dominance) across releases to properly examine the trend of usability complaints in Fig. 6. Again, in Fig. 6 it is shown that apart from between the release of Android version 1.1 and Cupcake (version 1.5) and between the last version of Honeycomb (version 3.2) and Ice Cream Sandwich (version 4.0), the Android community expressed increasing concerns about the software usability, with issues peaking in the Jelly Bean releases (version 4.1, 4.2 and 4.3) – notwithstanding that we did not consider these results against Android handset ownership and usage data, as reliable data is not available.

We performed deeper analysis on a random sample of issues (60 altogether) that were raised across releases where usability concerns were the highest (i.e., between Froyo (version 2.2) and Gingerbread (version 2.37), between Gingerbread (version 2.37) and Honeycomb (version 3.2), and between Ice Cream Sandwich (version 4.03) and Jelly Bean (version 4.3)). As noted in Section 3.2, we performed selective coding, before then engaging in constant comparison of the subsequent codes in order to provide a higher level of abstraction. Overall, a range of issues were reported across these Android OS releases, at times due to the OS features being unusable (did not work or produced errors), lacking utility (did not always work or

could be improved) or not useful (should probably be removed).

For instance, between Froyo (version 2.2) and Gingerbread (version 2.37) issue "*13370: Gingerbread SIP Receive Calls functionality does not work on data connection*", demonstrates a community member's unhappiness with the subject feature not working (or it was unusable). During the same period issue "*14246: Keyguard Manager Reenable Keyguard doesn't always work*" shows that, given that the Keyguard feature expected behavior was not always consistent (did not always work), the reporter's opinion was that there was general need for this feature to be improved.

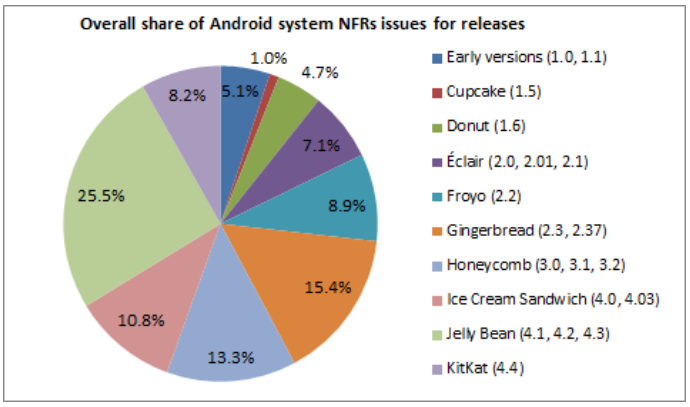

Fig. 5. Share of Android OS NFRs issues over time.

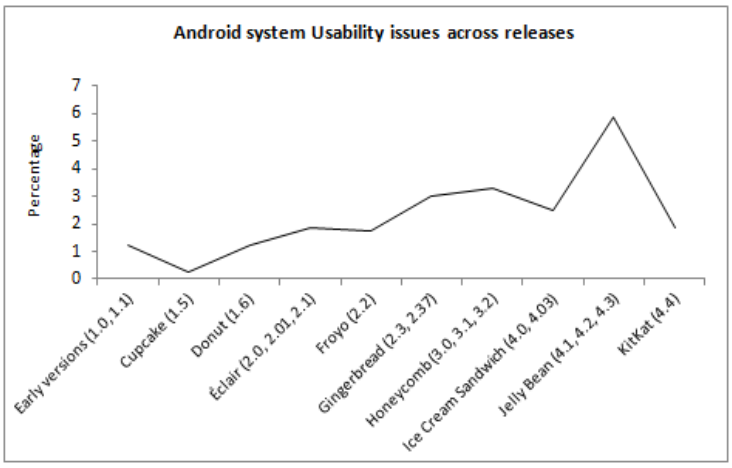

Fig. 6. Android OS usability issues across releases.

Between Gingerbread (version 2.37) and Honeycomb (version 3.2) issues "*15760: Add unknown phone number to existing contact from call log doesn't always work*" and "*16549: Touch panel cannot work sometimes*" pointed to a lack of utility in the required functionalities. In the same period issue "*17370: Do not retry to authenticate network with bad credentials*" indicated that it was not useful to have this specific feature (should be removed).

Finally, between the release of Ice Cream Sandwich (version 4.03) and Jelly Bean (version 4.3) issue "*39890: Bluetooth completely unusable in 4.2*" indicated that a required feature did not work (was unusable). Further, issue "*40991: Contact groups no longer usable"*, suggests that a previously usable feature had stopped working.

We next examined how Google's remedial efforts on the Android OS correlated with the issues stakeholders logged, in order to triangulate our findings above [56]. We gathered release notes from the

official Android developers' portal[2]. Google provides, for most OS versions, two types of notes: *users'* and *developers'* release notes. We obtained and combined all available release notes for both users and developers for each version. In correlating issues logged against what was released we observed that requests and responses significantly correlated for two of the ten versions of the Android OS, Ice Cream Sandwich and Jelly Bean. These correlations were strong for the Jelly Bean releases (r = 0.57, p = 0.01), and upper medium over Ice Cream Sandwich releases (r = 0.49, p = 0.01) [56]. These outcomes validate that the issue log provided a reliable avenue for capturing community members requests, and also that such requests were responded to by Google (refer to Licorish et al. [56] for further in-depth findings). We discuss these findings next.

5. Discussion

Our results support the potential for delivering insights to the software engineering community by studying the way software projects evolve as driven by the wider community of stakeholders. Such analysis is particularly relevant to understanding the advancement of software projects [24, 35, 36]. While the n-gram analysis of issues that is used in this work is somewhat divergent to those that are typically used when studying software evolution, we indeed note the relevance of aspects of Lehman's laws for explaining multiple characteristics of our results (considered in detail below). Additionally, although not entirely replicating Lehman's pattern of results [34, 35], in examining Fig. 6 it is observed that frequencies in stakeholders' concerns exhibited some level of instability or chaotic behavior after some specific releases, with much smoother trends between others.

We examine this issue further during our discussions in the remaining subsections, first presenting a summary of our results for Android architecture issues (in Section 5.1). We next summarize our results for Android NFRs in Section 5.2, and consider how Google released features towards the end of this section.

5.1. Software Architecture (RQ1)

(a) How Android architecture issues as logged on the issue tracker were distributed in the five OS layers, and (b) How the community concerns evolved from the OS initial release to its recent offerings? We observed that the majority of Android OS issues and stakeholders' dissatisfaction were related to the software applications (we caution however that some of these issues could be associated with handset providers' versions of customized software applications). This finding confirms that of Kumar Maji et al.'s [38], who observed a similar distribution of defects in their smaller scale study. In fact, our findings also vindicate our position about the relevance of studying issues in all of the OS's layers, as against a subset of issues [31]. We observed that there were also substantial numbers of concerns raised with the kernel and application framework layer, a finding also reported by Guana et al. [31] in their classification of issues in four layers. Our finding for the detection of issues in the application layer is understandable given that most mobile device users tend to interface heavily with features in this layer (e.g., web browsers and SMS). Similarly, developers of mobile Apps generally interface with frameworks and libraries (including those that are provided by third parties), and thus, in the process are likely to uncover inadequacies in these layers.

Results in Section 4.1 show that architecture issues in the Android OS were not evenly distributed across releases. We also note in Table I that with increasing capability of Android devices, so too are there increases in the number of concerns in the OS (see trend in fifth column). This pattern of higher prevalence of issues in the later software releases also previously correlated with increasing software use [36]. In fact, for Jelly Bean and KitKat, the community has recorded 20.7 and 45.5 issues per day respectively. While numerous reasons may be responsible for the increases in Android OS issues raised over time, the evolution

[2] http://developer.android.com/about/index.html

and growth in Android supported mobile hardware is likely to be an integral factor. For instance, the early T-Mobile G1 (Android 1.0) device possessed basic hardware and software capability, and had no on-screen keyboard or multi-touch capability, whereas the recent Nexus 5 (Android 4.4) provides these capabilities, along with advanced resource management and optimization (for CPU, memory and I/O), multi-mode processing, enhanced security and across the board application integration (e.g., Contacts, Gmail and SMS). The community's large membership helps to detect issues. These assessments are all plausible, particularly when considered in relation to Lehman et al.'s [35] laws for software quality and feedback (laws VII[3] and VIII[4] introduced in Section 2.1).

We see an increasing focus on device-specific issues. For instance, while application related issues dominated after Android OS early releases, more device-specific (e.g., about Nexus and Galaxy) issues were reported for recent releases. This evidence suggests that Google's OHA collaboration is shaping recent directions in the Android OS. Here we see relevance of Lehman et al.'s [35] law VII, and particularly in terms of Google's need to adapt to environment changes.

5.2. Non-functional Requirements (RQ2)

How have NFRs issues evolved in the Android OS? Stakeholders in the Android community were primarily concerned with the OS's usability. Notwithstanding that many device manufacturers tend to replace aspects of Android OS's interfaces (e.g., the camera interface) with their own skins, recurrence of usability issues has been held to affect user interest in software projects [57]. In the Android OS context, however, we see very rapid growth in the use of Android devices [28], suggesting that these issues were not encumbering to the community. One potential rationale for such growth is Android's open model, which likely influenced device manufacturers and community developers to feel a sense of shared ownership of the OS. Evidence has shown that stakeholders are more committed to software project success when they share in its ownership [58]. Enhancements in hardware capability and an ever expanding range of Android OS features are also likely to enhance community tolerance. Thus, promoting community-wide ownership and continuous adaptation probably had a positive impact on Android OS versions and devices sustained relevance [59, ch. 7].

The Android community was also concerned about the OS's functionality. Issues related to compliance, accuracy, interoperability and practicality were considered under this category. We observed that the most NFRs issues were raised after the release of Froyo (version 2.2), and particularly during the Jelly Bean releases. Strikingly however, there were said to be many major bug fixes and usability related improvements delivered as part of the Jelly Bean releases (Android versions 4.1, 4.2 and 4.3) (http://www.techradar.com/reviews/pc-mac/software/operating-systems/android-jelly-bean-1087230/review). We indeed noticed a degree of reduction in NFRs issues after the release of Jelly Bean 4.3. Although limited in scope, our deeper analysis revealed that Android's community raised quality-related concerns that were noteworthy. For instance, there were many features delivered between releases that were unusable (did not work or produced errors), some features lacked utility (did not always work or could be improved), and some features were not useful (should probably be removed).

Such issues, understandably, would have varying degrees of effect on the community. While users of mobile devices may not be interested in the OS's maintainability, and so, such complaints may not affect this section of the community, these members are likely to express unhappiness with slow software or OS features that open with errors. Accordingly, such occurrences, even when reported infrequently, should be

[3] The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
[4] E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement for other than the most primitive process.

of note to Android OS development teams, [59] (refer to Chapter 18).

In assessing our findings in relation to previous work, we note that, contrary to the evidence reported by Hindle *et al*. [32], Android's NFRs issues increased with time. Additionally, notwithstanding the single context that was considered for the MySQL and MaxDB systems in Hindle *et al.* [32], we see divergence in focus for the three communities (MySQL, MaxDB and Android). For instance, there were stronger interests in systems functionality and efficiency for MySQL and MaxDB respectively, while members of the Android community were most bothered by usability related issues. This divergence in focus shows that we need to be cautious about generalizing from outcomes for studies considering software systems evolution while neglecting the specific domain characteristics. That said, we see some convergence in terms of focus, where, like the MySQL and MaxDB systems in Hindle *et al*. [32], there was increasing emphasis on external technology-related issues as these systems matured (refer to the previous section for discussion).

In fact, one very positive observation for the Android community relates to our evidence of very few concerns raised in relation to efficiency and portability (refer to Section 4.2). Speed and power had been emphasized for Nexus 5 and KitKat (Android 4.4) (http://www.google.com/intl/all/nexus/5). However, beyond this Android OS release, our evidence suggests that the Android community was generally happy with the speed and efficiency of the OS over the years. The developer community has also expressed few concerns about portability. These benefits may trade-off the other concerns, and provide some form of balance for the community's discontent.

In fact, we observed that there was a strong correlation between stakeholders' requests for enhancement and Android developers' responses to those requests across specific releases [56], suggesting that these developers actually peruse the issue log in evolving the OS. In addition, this evidence confirms the multi-level, multi-loop, multi-agent feedback environment that characterizes the software evolution process. We consider the threats to this study next (in Section 6), prior to examining the implications for the work's findings in Section 7.

6. Limitations

We concede that this study, like any other case study [60], suffers from limitations that may present threats to the work's generalizability. We thus address this issue in this section. We follow the guidelines of Yin [61] and Runeson and Host [60] in discussing the limitations of this work. These authors suggest that the findings of case studies are meaningful when they are presented with an assessment of construct validity, internal validity, external validity and reliability – as each considered next.

Construct Validity: Construct validity seeks to consider the adequacy of the constructs for measuring what was intended. In this study we used the Android issue tracker to examine stakeholder concerns and the OS's defects. Although the Android issue tracker is publicly hosted (refer to Section 3), and so, this log is likely to capture most of the community concerns, issues may also be informally communicated and addressed within the development teams at Google. Of course our goal in this work was also to study the issues as provided by the wider Android community, thus, we believe that we have achieved this goal. In fact, we observed that features released for Android OS correlated with the requests of the community [56]. This evidence somewhat validates that the issue tracker provided reliable data, and was also monitored and used by Google in their remedial efforts. We focused, as far as possible, to include all terms, their synonyms and associated adjectives, nouns and verbs to examine the concepts under consideration. However, we accept that there was a possibility that we could have missed some terms. That said, our approach to study multiple forms of issues, and the convergence of our results triangulated our classification schemes somewhat, and suggests that our approach was robust. We separated the issues based on the dates of the Android OS releases. Given that device manufactures have been shown to delay upgrading their hardware

with recent Android OS releases [62], there is a possibility that some issues reported between specific releases 'belonged' to earlier releases. However, this issue was not detected by our EDA and small scale qualitative analysis. Finally, while we normalized our measures using percentage scores and the number of followers, customer demand (device usage) may also impact the number of issues reported. We have no access to this information however.

Internal Validity: Internal validity refers to the control of factors that are likely to affect the main variables under consideration. Given that case studies are generally uncontrolled, such studies invoke internal validity concerns [61]. In this work the archival data studied were not originally prepared for the purpose of research, and thus, there were many missing values in the issues studied (refer to Section 3). This issue was observed by previous research that considered subsets of the same data source [38]. Nevertheless, we did not observe any missing text in the description summary field of any of the 21,547 issues that were examined, which formed the core of our analysis. Additionally, previous studies considering Android OS artefacts and the issue tracker have confirmed the representativeness and appropriateness of these artefacts for studying various aspects of the Android OS [38, 63]. We concede that words may sometime possess dual meaning [50], and so, incidence of such words in our analysis is likely to skew our results. For instance, the word "security" may be used to describe being free from threat or being fulfilled (e.g., as with job security). Such dual meanings are likely to be negligible (if not unlikely) in the context of our data source and the analyses performed here.

External Validity: External validity considers the generalizability of the findings derived from research. We have studied the issues raised regarding the Android OS as logged by the community. In fact, the issues under consideration reflect stakeholders' dissatisfactions with features from the very first Android beta release to Android KitKat (version 4.4). This makes our findings generalizable to Android OS, in alignment with our goal. That said, and as noted above, there may also be other avenues for issues to be logged, and particularly between Google developers. Such issues may not be reflected in our analysis. Furthermore, Android's very rapid evolution has not been replicated by other mobile OS projects [28], and so, this may affect the generalizability of our findings. That said, the classification scheme used for studying NFRs issues is applicable to all software projects [64]. Furthermore, although the issue tracker of many mobile OSs are not publicly available, and the distribution of these OS issues may not be similar to what is observed in this work for Android OS, mobile OS like Microsoft Windows, Apple iOS, Symbian and BlackBerry are all likely to follow release-maintenance cycles similar to that of Android OS in order to improve their offerings.

Reliability: Reliability assesses the repeatability of research, the likelihood of which is often enhanced by adopting benchmarks [65]. In this study context, the n-gram technique used is an established approach used in computational linguistics for analyzing large volumes of text to facilitate the identification of frequently occurring words usage patterns [49] – refer also to Google's project at http://books.google.com/ngrams/. In addition, this approach was used previously for studying texts as produced in software development contexts (e.g., [50]). We also used previous classification schemes [31, 32], and so, there are strong claims for the validity and reliability of these schemes. In addition, we have applied an incremental approach to our analysis, taking into account the unique nature of the Android dataset (refer to Section 3). That said, the suitability of the classification schemes may still be debated, particularly given that previous work has noted variations in NFRs issues across projects [52, 53]. Notwithstanding this evidence, our approach to study multiple issues and to perform informal qualitative analysis provides a form of triangulation which acts as a countermeasure against reliability and validity threats [60].

Overall then, while there are indeed potential threats to the findings derived from the research conducted and reported here, extensive effort has been expended to ensure that the findings in this study are as robust

as possible.

7. Implications and Future Work

Given our evidence of active community participation from the very early Android beta releases we believe that active stakeholders' participation may have a positive effect on project evolution. Findings indicating the highest number of issues were reported about the Android OS application layer suggest that it would be prudent to retain members responsible for developing and maintaining such modules. Additionally, open source developers wishing to become quickly recognized in the Android OS community may also find it useful to study the workings of the code base and modules in the application layer, as their contributions in this layer would likely have the most impact on the community (given the higher prevalence of application layer issues). Of course, such members may also be quickly recognized if they are able to provide useful fixes for issues in the application framework and kernel layers. Furthermore, those who have contributed to the Linux kernel, and so, possess prior knowledge of its workings, may be able to offer tangible fixes and/or extensions for issues in this layer for the Android OS community.

There has been a steady increase in issues reported by the Android community with each successive release. This evidence goes against the position of previous researchers (e.g., [38]), which suggest that issues generally reduce as a project matures and the community members becomes larger. In fact, some 45.5 issues per day have been reported after the KitKat (Android 4.4) release to the end of March 2014, the largest of any release block in Table I. Android of course is less than a decade old, and given the volatility of the hardware of mobile devices, it can hardly qualify as a mature project. The community should therefore be prepared for these issues.

We observed that even with the prevalence of NFRs issues, growth in the Android community has still surpassed that of Android's main rivals. This, we contend, is linked to Android's open model, which promotes shared ownership in the community's offerings. Such levels of user tolerance may not be evident for commercial software products (and for closed models), which no doubt need to possess highly usable features that are commensurable with product costs. Promoting shared software ownership may also, more generally, influence community buy-in and stakeholders' commitment to software projects. This may particularly be the case if there are incentives to be gained by the community members. For instance, the Android app developer community may be willing to identify and possibly fix or provide workarounds for the OS issues which are likely to affect (perhaps through delays) their delivery of potentially highly marketable Apps. Community members are also likely to await fixes for bugs as a trade-off for newer, faster, more energy efficient, more durable and sleeker hardware. Of course such compromises may be less evident for those defects that are excessively prohibitive.

Given continued growth in Android device use, so too are community members' demands likely to grow. The recent move to modularize the Android OS could enable rapid delivery of small fixes, as against a big bang approach to the OS upgrade which may possibly be undesirable to some stakeholders both due to a negative effect on internet cost and mobile device performance during phone updates.

Research efforts aimed at exploring the techniques that are likely to deliver the best precision in detecting duplicate issues offer fruitful areas for future research [66]. Such techniques may be context-specific, where results may vary across OS layers. Context-specific defect detection is likely to be particularly necessary given that some OS issues are likely to be more critical than others, and thus, need fixing more urgently. Approaches that provide the best performance, in terms of detecting duplicate issues, and predicting bugs more generally, in the various subsystems of the OS would have great utility. Research employing text mining approaches and other contextual analysis techniques would also supplement these enquiries. Contextual analysis techniques in particular could help to reveal latent insights and to probe manifest

patterns revealed through the use of text mining approaches.

Acknowledgment

Thanks to Google for making the Android issues publically accessible to facilitate the analyses that are performed in this study. Thanks also to Professor Stephen MacDonell for his detailed and insightful comments on the early version of this work.

References

[1] Hassan, A. E. (2008). The road ahead for mining software repositories. *Proceedings of the Frontiers of Software Maintenance* (pp. 48-57).

[2] Asaduzzaman, M., Bullock, M. C., Roy, C. K., & Schneider, K. A. (2012). Bug introducing changes: A case study with Android. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (pp. 116-119).

[3] Thomas, S. W., Adams, B., Hassan, A. E., & Blostein, D. (2014). Studying software evolution using topic models. *Science of Computer Programming*, 80 (Part B), 457-479.

[4] Zaidman, A., Van, R. B., Van, D. A., & Demeyer, S. (2011). Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, *16(3)*, 325-364.

[5] Bieman, J. M., Andrews, A. A., & Yang, H. J. (2003). Understanding change-proneness in OO software through visualization. *Proceedings of the 11th IEEE International Workshop on Program Comprehension* (pp. 44-53).

[6] Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who should fix this bug? *Proceedings of the 28th International Conference on Software Engineering* (pp. 361-370).

[7] Dig, D., & Johnson, R. (2006). How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, *18(2)*, 83-107.

[8] Selby, R. W. (2005). Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, *31(6)*, 495-510.

[9] Licorish, S. A., & MacDonell, S. G. (2013). The true role of active communicators: An empirical study of Jazz core developers. *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering* (pp. 228-239).

[10] Licorish, S. A., & MacDonell, S. G. (2013). How do globally distributed agile teams self-organise? Initial insights from a case study. *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*.

[11] Licorish, S. A., & MacDonell, S. G. (2014). Combining text mining and visualization techniques to study teams' behavioral processes. *Proceedings of the 4th ICSME Workshop on Mining Unstructured Data* (pp. 16-20).

[12] Licorish, S. A., & MacDonell, S. G. (2014). Relating IS developers' attitudes to engagement. *Proceedings of the 25th Australasian Conference on Information System*.

[13] Licorish, S. A., & MacDonell, S. G. (2015). Communication and personality profiles of global software developers. *Information and Software Technology*, *64*, 113-131.

[14] Keertipati, S., Licorish, S. A., & Savarimuthu, B. T. R. (2016). Exploring decision-making processes in Python. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*.

[15] Licorish, S. A., & MacDonell, S. G. (2014). Personality profiles of global software developers. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*.

[16] Licorish, S. A., & MacDonell, S. G. (2014). Understanding the attitudes, knowledge sharing behaviors and task performance of core developers: A longitudinal study. *Information and Software Technology*, *56(12)*, 1578-1596.

[17] Patel, P., Licorish, S., Savarimuthu, B. T. R., & MacDonell, S. (2016). Studying expectation violations in socio-technical systems — A case study of the mobile app community. *Proceedings of the European Conference on Information Systems (ECIS)*.

[18] Martin, W., Sarro, F., Jia, Y., Zhang, Y., & Harman, M. (2016). A survey of app store analysis for software engineering.

[19] Lee, C. W., Licorish, S. A., Savarimuthu, B. T. R., & MacDonell, S. G. (2016). Augmenting text mining approaches with social network analysis to understand the complex relationships among users' requests: A case study of the android operating system. *Proceedings of the 49th Hawaii International Conference on System Sciences (HICSS 2016)*.

[20] Palomba, F., Linares-Vásquez, M., Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2015). *User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps*.

[21] Maalej, W., & Nabil, H. (2015). *Bug Report, Feature Request, or Simply Praise? On Automatically Classifying App Reviews*.

[22] Licorish, S. A., MacDonell, S. G., & Clear, T. (2015). Analyzing confidentiality and privacy concerns: insights from Android issue logs. *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (Nanjing, China).

[23] Keertipati, S., Savarimuthu, B. T. R., & Licorish, S. A. (2016). Approaches for prioritizing feature improvements extracted from app reviews. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Limerick, Ireland).

[24] Lehman, M. M. (1979). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, *1*, 213-221.

[25] Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998). On evidence supporting the FEAST hypothesis and the laws of software evolution. *Proceedings of the 5th International Software Metrics Symposium* (pp. 84-88).

[26] Baroudi, J. J., Olson, M. H., & Ives, B. (1986). An empirical study of the impact of user involvement on system usage and information satisfaction. *Communications of the ACM*, *29(3)*, 232-238.

[27] Zheng, X., Zhu, S., & Lin, Z. (2013). Capturing the essence of word-of-mouth for social commerce: Assessing the quality of online e-commerce reviews by a semi-supervised approach. *Decision Support Systems*, *56*, 211-222.

[28] Butler, M. (2011). Android: Changing the mobile landscape. *IEEE Pervasive Computing*, *10(1)*, 4-7.

[29] Klein, N., Corley, C. S., & Kraft, N. A. (2014). New features for duplicate bug detection. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 324-327.

[30] Licorish, S. A., MacDonell, S. G., & Clear, T. (2015). Analyzing confidentiality and privacy concerns: insights from android issue logs. *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*.

[31] Guana, V., Rocha, F., Hindle, A., & Stroulia, E. (2012). Do the stars align? Multidimensional analysis of Android's layered architecture. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (pp. 124-127).

[32] Hindle, A., Ernst, N. A., Godfrey, M. W., & Mylopoulos, J. (2011). Automated topic naming to support cross-project analysis of software maintenance activities. *Proceedings of the 8th Working Conference on Mining Software Repositories* (pp. 163-172).

[33] Ramil, J. F., & Lehman, M. M. (2000). Metrics of software evolution as effort predictors — A case study.

In *Proceedings of the International Conference on Software Maintenance* (pp. 163-172).

[34] Lehman, M. M., & Ramil, J. F. (2001). An approach to a theory of software evolution. *Proceedings of the 4th International Workshop on Principles of Software Evolution* (pp. 70-74).

[35] Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998). Implications of evolution metrics on software maintenance. *Proceedings of the International Conference on Software Maintenance* (pp. 208-217).

[36] Gala-Perez, S., Robles, G., Gonzalez-Barahona, J. M., & Herraiz, I. (2013). *Intensive Metrics for the Study of the Evolution of Open Source Projects: Case Studies from Apache Software Foundation Projects*.

[37] Hsu, J. S.-C., Chan, C.-L., Liu, J. Y.-C., & Chen, H.-G. (2008). The impacts of user review on software responsiveness: Moderating requirements uncertainty. *Information and Management*, *45(4)*, 203-210.

[38] Kumar Maji, A., Kangli, H., Sultana, S., & Bagchi, S. (2010). Characterizing failures in mobile oses: A case study with android and symbian. *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering* (pp. 249-258).

[39] Hindle, A., Godfrey, M. W., & Holt, R. C. (2009). What's hot and what's not: Windowed developer topic analysis. *Proceedings of the IEEE International Conference on Software Maintenance* (pp. 339-348).

[40] McKeen, J. D., Guimaraes, T., & Wetherbe, J. C. (1994). The relationship between user participation and user satisfaction: An investigation of four contingency factors. *MIS Quarterly*, *18(4)*, 427-451.

[41] *ISO/IEC 9126-3: Software engineering - Product quality - Part 3: Internal metrics*. ISO/IEC, Japan.

[42] Alipour, A., Hindle, A., & Stroulia, E. (2013). A contextual approach towards more accurate duplicate bug report detection. *Proceedings of the 10th Working Conference on Mining Software Repositories*.

[43] Licorish, S. A., & MacDonell, S. G. (2013). What can developers' messages tell us?: A psycholinguistic analysis of Jazz teams' attitudes and behavior patterns. *Proceedings of the 22th Australian Conference on Software Engineering* (pp. 107-116).

[44] Berelson, B. (1952). *Content Analysis in Communication Research*. Free Press, Glencoe, Illinois.

[45] Holsti, O. R. (1969). *Content Analysis for the Social Sciences and Humanities*. Addison Wesley, Reading, MA.

[46] Runeson, P., Alexandersson, M., & Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. *Proceedings of the 29th International Conference on Software Engineering* (pp. 499-510).

[47] Sharma, V. S., Ramnani, R. R., & Sengupta, S. (2014). A framework for identifying and analyzing non-functional requirements from text. *Proceedings of the 4th International Workshop on Twin Peaks of Requirements and Architecture* (Hyderabad, India).

[48] Thomas, S. W., Adams, B., Hassan, A. E., & Blostein, D. (2014). Studying software evolution using topic models. *Science of Computer Programming*.

[49] Manning, C. D., & Schtze, H. (1991). *Fundations of Statistical Natural Language Processing*. MIT Press, London.

[50] Soper, D. S., & Turel, O. (2012). An n-gram analysis of Communications 2000-2010. *Communications of the ACM*, *55(5)*, 81-87.

[51] Licorish, S. A., Lee, C. W., Savarimuthu, B. T. R., Patel, P., & MacDonell, S. G. (2015). They'll know it when they see it: Analyzing post-release feedback from the android community. *Proceedings of the 21st Americas Conference on Information Systems*.

[52] Cleland-Huang, J., Settimi, R., Xuchang, Z., & Solc, P. (2006). The detection and classification of non-functional requirements with application to early aspects. *Proceedings of the 14th IEEE International Conference on Requirements Engineering*.

[53] Mockus, A., & Votta, L. G. (2000). Identifying reasons for software changes using historic databases. *Proceedings of the International Conference on Software Maintenance.*

[54] Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering*.

[55] Glaser, B. G., & Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine Publishing Company, Chicago.

[56] Licorish, S. A., Tahir, A., Bosu, M. F., & MacDonell, S. G. (2015). On satisfying the android OS community: users' feedback still central to developers' portfolio. *Proceedings of the 24th Australasian Software Engineering Conference* (pp. 78-87).

[57] Lin, W. T., & Shao, B. B. M. (2000). The relationship between user participation and system success: A simultaneous contingency approach. *Information and Management*, *37(6)*, 283-295.

[58] Foster, S. T., & Franz, C. R. (1999). User involvement during information systems development: a comparison of analyst and user perceptions of system acceptance. *Journal of Engineering and Technology Management*, *16*, 329-348.

[59] Lehman, M. M., & Belady, L. A. (1985). *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA.

[60] Runeson, P., & Host, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14(2)*, 131-164.

[61] Yin, R. (2002). *Case study Research: Design and Methods*. Sage Publications, Inc, Thousand Oaks, CA.

[62] Wimberly, T. (2010). *Top 10 Android Phones, Bestselling Get Software Updates First*. Android and Me, City.

[63] Vidas, T., Votipka, D., & Christin, N. (2011). All your droid are belong to us: A survey of current android attacks. *Proceedings of the 5th USENIX Conference on Offensive Technologies*

[64] Bhattacharya, P., Ulanova, L., Neamtiu, I., & Koduru, S. C. (2013). An empirical analysis of bug reports and bug fixing in open source android apps. *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*.

[65] Layman, L., Williams, L., Damian, D., & Bures, H. (2006). Essential communication practices for extreme programming in a global software development team. *Information and Software Technology*, *48(9)*, 781-794.

[66] Tort, A., Olivé, A., & Sancho, M.-R. (2011). An approach to test-driven development of conceptual schemas. *Data and Knowledge Engineering*, *70(12)*, 1088-1111.

Sherlock A. Licorish is a lecturer in the Department of Information Science at University of Otago, in Dunedin, New Zealand. His research includes work on software development process modelling and assessment, the development and provision of software tools, and empirical software engineering and analytics. He employs data mining, data visualization, statistical analysis and other quantitative methods (e.g., social network analysis and linguistic and sentiment analysis) in his work. He has also used qualitative methods in his research, including qualitative forms of content analysis and dilemma analysis. These techniques (both quantitative and qualitative) are often applied to large repositories and software artefacts. Dr. Licorish is an active member of the Software Engineering and Information Systems community, and reviews articles for the main journals and conferences including: IEEE's TSE, I&M, JSS, I&ST, ACI, HICSS, EASE, AAAI, AMCIS, ECIS, ACIS, ACSW.