Identifying Architectural Technical Debt in Android Applications through Automated Compliance Checking

Roberto Verdecchia*†
*Gran Sasso Science Institute, L'Aquila, Italy
†Vrije Universiteit Amsterdam, The Netherlands
roberto.verdecchia@gssi.it

ABSTRACT

By considering the fast pace at which mobile applications need to evolve, Architectural Technical Debt results to be a crucial yet implicit factor of success. In this research we present an approach to automatically identify Architectural Technical Debt in Android applications. The approach takes advantage of architectural guidelines extraction and modeling, architecture reverse engineering, and compliance checking. As future work, we plan to fully automate the process and empirically evaluate it via large-scale experiments.

KEYWORDS

Software Architecture, Technical Debt, Android, Compliance Checking

1 INTRODUCTION

In the past decade a drastic media consumption shift towards mobile devices took place. It is hence not surprising that the development of mobile apps experienced an exponential growth in recent times. The shift towards mobile development was supported by the advent of *app stores*, such as Google Play and Apple App Store, where millions of mobile apps are available nowadays. Mobile application development results to be a highly competitive business, which can lead to high profits, but is also sensible to the introduction of errors with tremendous financial impact [4]. The mobile application business model is tightly coupled with users satisfaction, who can efficiently express their opinions through reviewing systems. It is hence paramount, to ensure the user satisfaction and revenue of apps, to be able to promptly and efficiently release new versions to introduce new features, fix bugs, and rapidly adapt to users' needs.

By considering the fast pace at which mobile apps need to evolve, Architectural Technical Debt (ATD) results a crucial yet implicit factor of success. This is confirmed by the recent release of a set of Android architectural components aimed to lower the complexity of Android apps and provide a recommended Android architecture [1].

ATD is defined as sub-optimal design decisions hindering the evolvability and maintainability of software apps over time. By identifying, resolving, and monitoring ATD of mobile apps, it is possible to enable them to rapidly adapt according to users' needs. While ATD has high impact on overall software quality, its presence is hard to uncover due to its complexity and lack of tools [3].

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3198442

In this research we present a novel approach, based on architecture reverse engineering and compliance checking, for identifying ATD hotspots in Android apps. The presented approach is, to the best of our knowledge, the first one conceived specifically to identify ATD in Android applications.

2 APPROACH

In the literature several techniques aimed to identify ATD can be found [5]. Among these, a typology of approaches focuses on identifying ATD by comparing the architecture of the implemented software applications with a reference architecture. Occurrences where the implemented architecture is non-compliant to the envisioned reference architecture are regarded as potential ATD Items (ATDIs) [1] This typology of approaches is particularly interesting as design guidelines can be directly embedded in the reference architecture. Building on such concept, the reference architecture can even be composed exclusively of architectural guidelines aimed at avoiding ATD. The approach presented in this research, conceived to identify ATDI of Android apps, is based on such intuition. The approach consists of five steps, namely: (1) Android architectural guidelines extraction, (2) Android reference architecture establishment, (3) reverse engineering of implemented architectures, (4) compliance checking, and (5) quantitative assessment of compliance violations. In the reminder of this section the steps constituting the approach, depicted in Figure 1, are further detailed.

Step 1: Android architecture guideline extraction. The first step of the approach consists in the identification of architectural guidelines that Android apps should adhere to in order not to incur in potential ATD. The data sources adopted for the extraction of architectural guidelines to construct the Android reference architecture are complementary and heterogeneous, in order to be as encompassing as possible, and consist of:

- **Official Android Guidelines:** Official Android documentation available online, such as the *Guide to App Architecture*[2];
- **Academic researches:** Peer-reviewed research papers considering architectural guidelines of Android apps, e.g. the study of Bagheri et al. [2];
- **Grey literature:** Non-academic writings on the topic available online, e.g. articles featured in Android related websites and blogs;
- **Developer interviews:** Semi-structured interviews with Android developers to validate and complement the data extracted from the above mentioned data sources.

Quality assessment is conducted on the extracted guidelines to ensure the soundness of the findings, remove duplicates, etc. The

[1] As an example, ATDIs in Android may arise when developers need to maintain different versions of the same Android API call (e.g., to request permissions) in order to (i) keep up with new releases of Android and (ii) maintain backwards compatibility.
[2] https://developer.android.com/topic/libraries/architecture/guide.html; Accessed 27 February 2018.

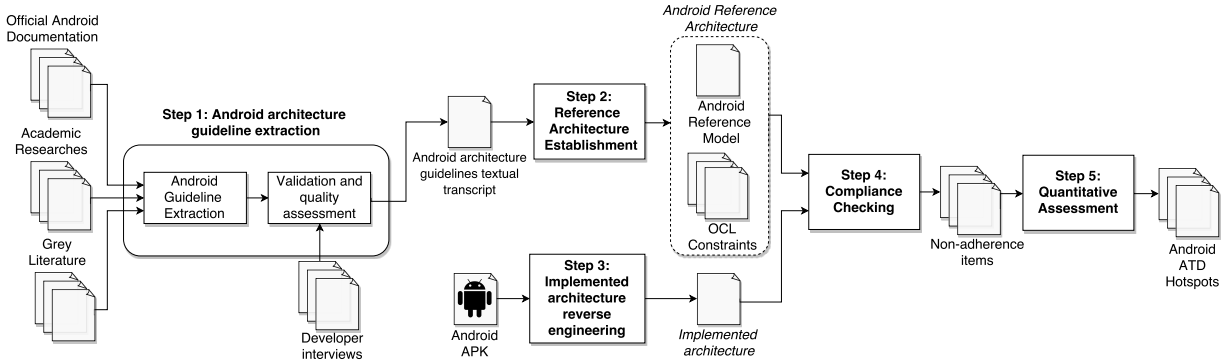

Figure 1: Android ATD hotspot identification approach overview

output of this step is a textual transcript of the identified Android architectural guidelines.

Step 2: Android reference architecture establishment. This step consists of the formalization of the textual transcript produced in Step 1. This process is required in order to effectively format the data for the subsequent automated analysis described in Step 4. Specifically, this step consists in developing a software model which conveys the information of the architectural guidelines extracted in Step 1. The resulting model is referred to as *Android reference model*. The reference model conforms to a chosen architecture description language (ADL). Specifically the Acme ADL[3] results suited for this process due to technology constraints dictated by Steps 3-4.

In addition to the Android reference model, in order to complement it with the information which cannot be exhaustively represented in form of a software model, a set of constraints expressed through the Object Constraint Language (OCL) are defined. The combination of the reference model and OCL constraints is what is jointly referred to in this document as *Android reference architecture*. The *output* of this step is an Android reference architecture composed of Android architectural guidelines in form of a software model and complementary OCL constraints.

Step 3: Reverse engineering of implemented architecture. This step consists in the retrieval of the architecture of an implemented Android application through the analysis of its source code or APK. Specifically, this step is constituted by the automated reverse engineering of the most prominent Android architectural components (i.e. *Activities*, *Services*, *Content providers*, and *Broadcast receivers*) of an Android application and the relations between such building blocks in terms of connectors and ports. This process was first proposed by Bagheri et al. [2], who also provided empirical evidence of its effectiveness. Due to the potential complexity of this process, this step has to be carried out by utilizing dedicated architecture reverse engineering tools. In particular, this can be achieved by adopting the *ACME-Generator* tool[4], which is a dedicated tool for reconstructing specifically the architecture of Android apps. Note that to ease the compliance checking process (Step 4) both the ANdroid reference architecture and the implemented architecture must either (i) adhere to the same metamodel or (ii) be linked by a suitable model-to-model transformation being able to bridge models conforming to them. The *output* of this step is the reverse engineered architecture of an implemented Android application.

Step 4: Compliance checking. Subsequent to the establishment of the Android reference architecture and the implemented architecture, a compliance checking process is carried out. During this process, items of non-adherence of the implemented architecture w.r.t. the Android reference architecture are identified. Due to its complexity, this step has to be carried out (semi-)automatically. In order to carry out a sound compliance checking process and reduce the number of potential false positives, the compliance checking has to consider a semantic comparison logic. This can be achieved by utilizing the model comparison tool EMFCompare[5], which offers a vast range of extension and customization mechanisms through which the *ad-hoc* comparison process can be implemented. The *output* of this step is the set of the non-adherence items of the implemented architecture w.r.t. the Android reference architecture.

Step 5: Quantitative assessment of compliance violations. Once the set of non-adherence items is computed, it is possible to analyze the gathered data to identify which architectural elements of the implemented architecture violate the highest number of Android architectural guidelines. Such identified items, referred to as *Android ATD hotspots*, are stored for a final manual inspection to prioritize them, select which require refactoring, etc. The *output* of this step is the set of Android ATD hotspots, i.e. the components of the implemented architecture which contain the highest number of non-adherence items w.r.t. the Android reference architecture.

3 CONCLUSION AND OUTLOOK

In this research we present a novel approach to identify ATD of Android apps based on architectural guidelines extraction and modeling, architecture reverse engineering, and compliance checking. We plan to automate the process and extensively evaluate it on large set of Android apps. The approach enables us to conduct evolutionary studies on the ATD of Android apps, through which a higher precision of the approach could be achieved, e.g. by considering code churn to rank more precisely ATDIs.

REFERENCES

[1] Android and Architecture. In *Android Developers Blog*. https://android-developers.googleblog.com/2017/05/android-and-architecture.html Accessed 18 March 2018.

[2] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. 2016. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software* 119 (2016), 31–44.

[3] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

[4] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 477–487.

[5] Roberto Verdecchia, Ivano Malavolta, and Patricia Lago. 2018. Architectural Technical Debt Identification: The Research Landscape. In *International Conference on Technical Debt (TechDebt)*.

[3] http://www.cs.cmu.edu/~acme/

[4] https://github.com/arsadeghi/ACME-Generator. Accessed 27th February 2018.

[5] https://www.eclipse.org/emf/compare/. Accessed 27th February 2018.