Elsayed EK, ElDahshan KA, El-Sharawy EE, Ghannam NE. 2019. Reverse engineering approach for improving the quality of mobile applications. PeerJ Computer Science 5:e212 https://doi.org/10.7717/peerj-cs.212

Reverse engineering approach for improving the quality of mobile applications

Eman K. Elsayed [1] , **Kamal A. ElDahshan** [1] , **Enas E. El-Sharawy** [1] , **Naglaa E. Ghannam** [Corresp. 1]

[1] Mathematical and Computer science Department, Al Azhar University, Cairo, Egypt

Corresponding Author: Naglaa E. Ghannam
Email address: naglaasaeed@azhar.edu.eg

Background: Portable applications (Android applications) are becoming increasingly complicated by mind-boggling programming frameworks. Applications must be produced rapidly and advance persistently in order to fit new client requirements and execution settings. However, catering to these imperatives may bring about poor outline decisions on design choices, known as anti-patterns, which may possibly corrupt programming quality and execution. Thus, the automatic detection of anti-patterns is a vital process that facilitates both maintenance and evolution tasks. Additionally, it guides developers to refactor their applications and consequently enhance their quality.

Methods: We propose a reverse-engineering approach to analyze Android applications and detect the anti-patterns from mobile apps. We validate the effectiveness of our approach on a set of popular mobile apps such as YouTube, Whats App, Play Store and Twitter. The result of our approach produced an Android app with fewer anti-patterns, leading the way for perfect long-time apps and ensuring that these applications are purely valid.

Results: The proposed method is a general detection method. It detected a set of semantic and structural design anti-patterns which have appeared 1262 times in mobile apps. The results showed that there was a correlation between the anti-patterns detected by an ontology editor and OntoUML editor. The results also showed that using ontology increases the detection percentage approximately 11.3%, guarantees consistency and decreases accuracy of anti-patterns in the new ontology.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Reverse engineering approach for improving Mobile Applications' Quality

Eman K. Elsayed[1], Kamal A. ElDahshan[1], Enas E. El-Sharawy[1], Naglaa E. Ghannam[1]
[1] Mathematical and Computer science Dept., Faculty of Science, Al-Azhar University, Cairo, Egypt

Corresponding Author:
Naglaa Ghannam[1]
Cairo, Egypt
Email address: naglaasaeed@azhar.edu.eg

Abstract

Background: Portable applications (Android applications) are becoming increasingly complicated by mind-boggling programming frameworks. Applications must be produced rapidly and advance persistently in order to fit new client requirements and execution settings. However, catering to these imperatives may bring about poor outline decisions on design choices, known as anti-patterns, which may possibly corrupt programming quality and execution. Thus, the automatic detection of anti-patterns is a vital process that facilitates both maintenance and evolution tasks. Additionally, it guides developers to refactor their applications and consequently enhance their quality.

Methods: We propose a reverse-engineering approach to analyze Android applications and detect the anti-patterns from mobile apps. We validate the effectiveness of our approach on a set of popular mobile apps such as YouTube, Whats App, Play Store and Twitter. The result of our approach produced an Android app with fewer anti-patterns, leading the way for perfect long-time apps and ensuring that these applications are purely valid.

Results: The proposed method is a general detection method. It detected a set of semantic and structural design anti-patterns which have appeared 1262 times in mobile apps. The results showed that there was a correlation between the anti-patterns detected by an ontology editor and OntoUML editor. The results also showed that using ontology increases the detection percentage approximately 11.3%, guarantees consistency and decreases accuracy of anti-patterns in the new ontology.

Introduction

Mobile applications take center stage in our lives today. We utilize them anywhere, at any time and for everything. We use them to peruse websites, shop, search for everything we need and for basic administration such as banking. However, the dependability and quality of mobile applications are basic. Like any other applications, the initial design of mobile apps is affected by bug-settling and the introduction of new properties, which change results. All of these

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

elements can occasionally affect the quality of design (Parnas D. L, 1994). This aspect is known as software degeneration, which can exist in the form of design flaws or anti-patterns (Eick S. G. et al. 2001).

One of the most important factors in the development of software systems is improving software quality. The success of software design depends on the availability of quality elements such as maintainability, manageability, testability, and performance. These elements are adversely affected by anti-patterns. Anti-patterns are bad practice in software design that impede maintenance and degrade performance. Many tools and methods have been introduced for measuring the quality of software products. The automatic detection of anti-patterns is a good way to support maintenance, uncomplicate evolution tasks and improve usability and software quality. We noted many other approaches were interested in detecting code smell or other code anti-patterns. although it has been noted that anti-pattern detection at the design level reduces many code anti-patterns and is more general.

According to Raja, V. (2008), engineering is the process of designing, manufacturing, assembling, and maintaining products and systems. Engineering has two types, forward engineering and reverse engineering. The term Reverse Engineering (RE) according to our approach, refers to the process of generating UML diagrams followed by generating OWL ontologies of mobile apps through importing and analyzing the source code.

Generally, we can use ontology re-engineering for direct incorporation as an ontology development method (Obrst et al., 2014) by allowing the designer to analyze the common components dependence. The low barriers between ontologies provide reusability.

Designing a high-quality mobile application pattern remains an ongoing research challenge. The proposed approach aims to detect structure and semantic anti-patterns in the design of a high-quality mobile application pattern, and to show which method is better for the integration of apps.

Motivated by the research mentioned above, the major contributions of this paper are seven-fold:

• Presenting a new method for generating OWL Ontology of mobile apps.
• Presenting a general method for designing a high-quality mobile application pattern.
• Illustrating how the proposed method can detect both structure and semantic anti-patterns in the design of mobile applications.
• Describing how we evaluate the proposed method in 29 publicly available mobile applications. Showing how it detects and treats 15 designs' semantic and structure anti-patterns that appeared 1262 times.
• Presenting the integration of mobile apps using two different scenarios to improve the contents and quality of the apps.
• Showing how semantic integration among mobile apps decreases the accuracy of anti-patterns in the generated OWL Ontology pattern when compared to the original apps.
• Analyzing the relationships among the object-oriented anti-patterns.

In the rest of the paper, we subsequently present the related work. Next, we present some basic definitions, and the details of the proposed approach is described. After that, the empirical validations of the proposed method are presented, followed by the results and discussion. And, finally, the concluding remarks are given, along with scope for future work, in the last section.

Related Work

RE methodology is important because it offers good benefits. RE allows for the understanding of the construction of the user interface and algorithms of applications. Additionally, we can know all of the properties of the app, its activities, permissions and can read the Mainfest.xml of the apps. RE methodology has been used in many approaches for many purposes. According to Song, L. et al. (2017), the RE technique was used to improve the security of Android apps. They introduced the AppIS system that can effectively enhance the app's security and its strength against repackaging and cumulative attack. Zhou, X. et al. (2018) used the RE technique to detect logging classes, and to remove logging calls and unnecessary instructions. Arnatovich, Y. L et al. (2018) used RE to perform a program analysis on a textual form of the executable source, and to represent it with an intermediate language (IL). This (IL) has been introduced to represent applications executable Dalvik (dex) bytecode in a human-readable form.

Many empirical studies have demonstrated the negative impact of anti-patterns on change-proneness, fault-proneness, and energy efficiency (Romano et al., 2012; Khomh et al., 2012; Morales et al., 2016). In addition to that, Hecht et al. (2015); Chatzigeorgiou & Manakos (2010); Hecht et al. (2016) observed an improvement in the user interface and memory performance of mobile apps when correcting Android anti-patterns. They found that anti-patterns were prevalent in the evolution of mobile apps. They also confirmed that anti-patterns tend to remain in systems through several releases, unless a major change is performed on the system. Many efficient approaches have been proposed in the literature to detect mobile applications' anti-patterns. Alharbi et al. (2014) detected the inconsistency anti-patterns in mobile applications that were only related to camera permissions and similarities. Joorabch et al. (2015) detected the inconsistency anti-patterns in mobile applications using a tool called CHECKCAMP that was able to detect 32 valid functional and data inconsistencies between app versions. Hecht et al. (2015) used the Paprika approach to detect some popular object-oriented anti-patterns in mobile applications. Linares-Vásquez et al. (2014) detected 18 OO anti-patterns in 1,343 java mobile apps by using DÉCOR. This study focused on the relationship between smell anti-patterns and application domain. Also, they showed that the presence of anti-patterns negatively impacts software quality metrics, in particular, metrics related to fault-proneness. Yus, R., & Pappachan, P. (2015) analyzed more than 400 semantic Web papers, and they found that more than 36 mobile apps are semantic mobile apps. They showed that the existence of semantic helps in better local storage and battery consumption. So we believe that the detection of semantic anti-patterns will support these factors in some way. Palomba et al. (2017) proposed an automated tool called A DOCTOR. This tool can identify 15 Android code smells. They made an empirical study conducted on the source code of 18 Android applications, and revealed that the proposed tool reached 98% precision and 98% recall. A DOCTOR detected almost all the code smell instances existing in Android apps. Hecht et al. (2015) introduced the PAPRIKA tool to monitor the evolution of mobile app quality based on anti-patterns. They detected the common anti-patterns in the code of the analyzed apps. They detected seven anti-patterns, three of them were OO anti-patterns and four were mobile anti-patterns.

Ontology and Software Engineering

According to the IEEE Standard Glossary (1990), software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software".

Also, from the knowledge engineering community perspective, computational ontology is defined as "explicit specifications of a conceptualization". According to Calero et al. (2006); Happel. J., & Seedorf, S. (2006), the importance of sharing knowledge to move software to more advanced levels requires a explicit definition to help machines interpret this knowledge. So they decided that ontology is the most promising way to address software engineering problems.

El-sayed et al., 2016 proofed the similarities in infrastructures between UML and ontology components. They proposed checking some UML quality features using ontology and ontology reasoning services in order to check consistency and redundancies over UML models. This would lead to a strong relationship between software design and ontology development.

In software engineering, ontologies have a wide range of applications, including model transformations, cloud security engineering, decision support, search and semantic integration (Kappel et al., 2006; Aljawarneh et al., 2017; Maurice et al., 2017; Bartussek et al., 2018; De Giacomo et al. 2018). Semantic integration is the process of merging the semantic contents of multiple ontologies. The integration may be between applications that have the same domain or have different domains in order to take the properties of both applications. We make ontology integration for many reasons: to reuse the existing semantic content of applications, to reduce effort and cost, to improve the quality of the source content or the content itself, and to fulfill user requirements that the original ontology does not satisfy.

Proposed Method

In this section, we introduce the key components of the proposed method for analyzing the design of mobile apps to detect design anti-patterns, and for making semantic integration between mobile apps via ontology reengineering.

Anti-pattern Detection

The proposed method for anti-pattern detection consists of three phases and is summarized in 'Fig. 1'.

1. **The first phase** presents the process of reformatting the mobile application to Java format.
2. **The second phase** presents the reverse-engineering process. In this phase, we used RE to reverse the Java code of the mobile apps generating UML class diagram models. Additionally, a lot of design anti-patterns are detected.

3. **The third phase** completes the anti-pattern detection and correction processes. We detect and correct semantic anti-patterns to reverse the high-quality code again, that is, to get high-quality Android patterns.

The Integration of Mobile Apps

Merging mobile apps is a good step in mobile app development. The proposed method introduces two ways of integrating mobile apps. The first way begins after decompiling the APK of the apps. We use reverse engineering methodology for generating one UML class diagram of both apps and then start the detection of the anti-patterns process for the integrated app (Fig. 2). The second way to integrate mobile apps is through merging the OWL ontologies of both apps, generating one OWL ontology for the two apps (Fig. 3).

The Implementation

In this section, we propose the implementation of the proposed detection method and determine which packages are suitable for each phase. The Android Application Package (APK) file is required for starting the reverse process.

• **The First Phase:** APK files are zip files used for the installation of mobile apps. We used the unzip utility for extracting the files stored inside the APK. It contained the AndroidManifest.xml, classes.dex containing the java classes we used in the reverse process, and resources.arsc containing the meta-information. We de-compiled the APK files using apktool or Android de-compiler. Android de-compiler is a script that combines different tools to successfully de-compile any (APK) to its java source code and resources. Finally, we used a java de-compiler tool such as JD-GUI to de-compile the java classes. JD-GUI is a standalone graphical utility that displays the Java source codes of ".class" files. So, the input of the first phase was the APK file of the mobile app and the output was the java classes of the APK app.

• **The Second Phase**: We used the RE approach for generating the UML class diagram model of the mobile app. Using different modeling tools of the UML, we found Modelio 3.6 to be a suitable one for our proposed method. The UML class diagram was generated by reversing the java binaries of the mobile app. Detecting the anti-patterns in the UML model is the first step in the detection process. So, the input of the second phase was classes.java of the app and the output was the UML class diagram model of the app with a list of the detected anti-patterns.

• **The Third Phase**: By converting the model to XML format, we could generate it as an OntoUML model in OLED, which is the editor of OntoUML for detecting semantic anti-patterns. OntoUML is a pattern-based and ontologically well-founded version of UML. Its meta-model has been designed in compliance with the ontological distinctions of a well-grounded theory named the Unified Foundational Ontology (UFO). OLED editor also supports the transformation of the OLED file to the OWL ontology of the mobile app, allowing the detection of inconsistency and semantic anti-patterns using the 'reasoner' ontology in Protégé. Protégé is the broad ontology editor commonly used by many users.

Empirical Validations

We assessed our approach by reporting on the results we obtained for the detection of 15 anti-patterns on 29 popular Android apps downloaded from the APK Mirror.

Applications under Analysis

We downloaded the mobile apps in Table 1 from APK Mirror. We selected some popular apps such as YouTube, WhatsApp, Play Store and Twitter. The size of the apps included the resources of the application, as well as images and data files (Table 1). The research study included the identification and repeating of anti-patterns across different domains and different sizes.

Case Study

To explain the proposed method, we presented a snapshot of it in a different case study "Avast Android Mobile Security". Using the reverse technique, we generated the UML class diagram model of the java classes in Modelio, and included its classes, subclasses, class attributes, operations, and the associations between them (Fig. 4).

After generating the UML class diagram of the app in Modelio, we detected 229 anti-patterns using the 'Avast Android Mobile Security'. The anti-patterns are shown in Fig. 5. The number and the location of the anti-patterns were determined.

There were 10 detected anti-patterns (without repeat): "NameSpaces have the same name (NHSN)", "NameSpace is Leaf and is derived (NLAD)", "NameSpace is Leaf and is abstract (NLAA)", "Generalization between two un-compatible elements (GBUE)", "A public association between two Classifiers one of them is public and the other is privet (PACPP)", "Classifier has several operations with the same signature (CHSO)", "Classifier has attributes with the same name (CHSA)", "The status of an Attribute is abstract and class(SAAC)", "A destructor has two parameters (ADHPS)" and finally "MultiplicityMin must be inferior to MultiplicityMax (MMITMM)". Figure 6 shows a sample of them.

For correcting the detected anti-patterns, we introduce the anti-pattern and the correction method in Table 2.

To convert the UML model to XML format, we converted it into an enterprise architecture file (EA) then converted it to an OLED file. In the "Avast Android Mobile Security' OLED file, we validated the model for detecting the anti-patterns. The detected anti-patterns in the different apps were: Association Cycle anti-patterns (AC), Binary relations with Overlapping Ends anti-patterns (BinOver), Imprecise Abstraction (ImpAbs) anti-patterns and Relation Composition anti-patterns (RC)). For correcting the detected anti-patterns via OntoUML and the correct method for each one, we introduce 'Table 3'.

After anti-pattern detection using OntoUML editor, OLED supports the transformation of OLED file to the OWL Ontology. We checked the inconsistency anti-patterns using the reasoner of Ontology editor (Protégé). The reasoner detected the inconsistency anti-patterns.

We detected the anti-patterns NameSpaces have the same name, Classifier has several operations with the same signature, Classifier has attributes with the same name, and MultiplicityMin must be inferior to MultiplicityMax, which we detected after generating the class diagram in Modelio, and detected the anti-pattern (Association Cyclic) which was detected via OntoUML.

Integration of Mobile Applications

In this section, we apply two different methods for integration between mobile apps to compare which method reduces the outcome of anti-patterns in the mobile app results.

In order to explain that, we took the integration of "Viber” and “Whatsapp" apps as a case study for the integration of social mobile apps. In the first integration method, we reformatted the APK of the apps to Java format, using RE methodology to generate one UML class diagram for both apps. After that, we did all the steps of the proposed detection method to generate a pattern of the new app.

For the second method of integration, we already had the OWL ontologies for all apps from the first part of the proposed method. We merged the OWL ontologies of both apps using a Prompt Protégé plugin to generate one OWL ontology pattern. Figure 7 and Figure 8 show the integration input and output. Finally, we used "Reasoner in Protégé" to check the consistency after integration.

Results and Discussion

Anti-pattern Detection

We applied our proposed method on a sample of 29 Android applications, which we downloaded from the APK Mirror. The proposed method detected 15 anti-patterns. The total number of anti-patterns that appeared in the 29 apps was 1262 anti-patterns. We classified the anti-patterns according to their existence in the UML class diagram components. The occurrences of the anti-patterns are given in Table 4.

Table 5 shows the detected anti-patterns in each app using the proposed methodology and the total number of anti-patterns in the 39 mobile apps.

We found that the "Anti-patterns in the class" group is the most commonly detected anti-pattern in Android apps. The "Anti-patterns in Operation" is the least commonly appeared anti-pattern (Fig. 9).

We used SPSS (Statistical Package for the Social Sciences) to analyze the correlation between the five groups of anti-patterns, and the correlation between the detection tools of the proposed method. A negative correlation means a reverse correlation between them, and a positive correlation means there is a direct correlation between them (Table 6 and Table 7). The greatest correlations were between attributes and operations groups, and between Modelio and Protégé.

We used a one-way ANOVA test among the three tools and the anti-patterns of the five groups to find the relationship between the used tools and the detected anti-patterns. From the ANOVA test, we found a significant difference of 0.578 in Protégé detection, while in Modlio detection it was 0.464, and finally in OLED, it was 0.926. This implies they are all necessary and we cannot ignore any one of them in our proposed method to get a high-quality mobile app pattern (Fig. 10).

Comparing the Mobile App integration methods

Although there are similarities between Viber and Whatsapp apps, when we applied the two integration methods in section 6.3, we found that the number of detected anti-patterns in the new app was not the same. The detected anti-patterns using the second method (Ontology Integration) was less than the number detected by using the first method (UML). This indicates that semantic integration decreases the accuracy of anti-patterns in apps. Table 8 shows the number of anti-patterns in each app and the number of them in the mobile app pattern after merging. The

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

enhancement using ontology is approximately 11.3% in addition to a consistency check. Additionally, using ontology to separately refine Viber or Whatsapp as a pattern enhanced them approximately 4.04 % and 89%, respectively, in addition to a consistency check.

Conclusions

In this paper, we focused on improving mobile applications' quality. Our method is distinct from other methods. We introduced a general method to automatically detect anti-patterns not by using specific queries, but by using Modelio, OntoUML, and Protégé in a specific order to get positive results. Also, concerning the related work section, our proposed method is more general than other methods as the proposed method supports semantic and structural anti-pattern detection at the levels of both design and code.
For evaluation of the proposed method, we applied it on a sample of 29 mobile applications, and it detected 15 semantic and structural anti-patterns. According to the proposed classification of anti-patterns, "the anti-patterns in the class group" was the most frequent anti-pattern, and "the anti-patterns in the attribute group" was the least frequent. The results also showed that there is a correlation between the Modelio and Protégé platforms. Additionally, there is a correlation between OLED and Protégé while there is no correlation between Modelio and OLED.
For evaluating and analyzing which integration method was better, we applied the two methods on similar mobile apps. We found that using ontology increases the detection percentage approximately 11.3%, and guarantees consistency. In addition to that, it decreased the accuracy of the anti-patterns in the new ontology. Accordingly, semantic ontology integration has a positive effect on the quality of the new app. It helps to develop a correct, consistent and coherent integrated model that has few anti-patterns.

In the future, we will analyze the relationship between design and code anti-patterns before and after the integration of mobile apps.

References

Alharbi, K., Blackshear, S., Kowalczyk, E., Memon, A. M., Chang, B. Y. E., & Yeh, T. (2014, April). Android apps consistency scrutinized. In CHI'14 Extended Abstracts on Human Factors in Computing Systems, 26 April – 01 May 2014; Toronto, Ontario, Canada. New York, NY, USA: ACM. pp. 2347-2352. ACM.
Aljawarneh, S. A., Alawneh, A., & Jaradat, R. (2017). Cloud security engineering. Future Generation Computer Systems, 74(C), 385-392.
Arnatovich, Y. L., Wang, L., Ngo, N. M., & Soh, C. (2018). A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation. IEEE Access, 6, 12382-12394.
Bartussek, W., Weiland, T., Meese, S., Schurr, M. O., Leenen, M., Uciteli, A., ... & Lauer, W. (2018, April). Ontology-based search for risk-relevant PMS data. In Biomedical Engineering Conference (SAIBMEC), 1-4. IEEE.

Chatzigeorgiou, A., & Manakos, A. (2010, September). Investigating the evolution of bad smells in object-oriented code. In Seventh International Conference on the Quality of Information and Communications Technology (QUATIC), 10, 106-115. IEEE.

Calero, C., Ruiz, F., & Piattini, M. (Eds.). (2006). Ontologies for software engineering and software technology. Springer Science & Business Media.

De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., & Rosati, R. (2018). Using ontologies for semantic data integration. In A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years, 187-202. Springer, Cham.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering, 27(1), pp. 1-12.

Elsayed, E., El-Dahshan, K., El-Sharawy, E., & Ghannam, N. (2016). Semantic Anti-patterns Detection in UML Models based on Ontology Catalogue. Artificial Intelligence and Machine Learning Journal, 16, 1687-4846.

Happel, H. J., & Seedorf, S. (2006, November). Applications of ontologies in software engineering. In Proc. of Workshop on Semantic Web Enabled Software Engineering"(SWESE) on the ISWC, 5-9.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., & Duchien, L. (2015, November). Tracking the software quality of Android applications along their evolution (t). In 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 9-13 Nov. 2015; Lincoln, NE, USA. Washington, DC, USA: IEEE, pp. 236-247.

Hecht, G., Moha, N., & Rouvoy, R. (2016, May). An empirical study of the performance impacts of android code smells. In Proceedings of the International Conference on Mobile Software Engineering and Systems, 14-22 May; Austin, Texas. New York, NY, USA: ACM. pp. 59-69. ACM.

Hecht, G., Rouvoy, R., Moha, N., & Duchien, L. (2015, May). Detecting antipatterns in android apps. In 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 16 – 17 May 2015; Florence, Italy. Piscataway, NJ, USA: IEEE. pp. 148-149.

IEEE Standard Glossary of Software Engineering Terminology-Description, (1990). http://ieeexplore.ieee.org/servlet/opac?punumber=2238. Accessed in January 2019.

Joorabchi, M. E., Ali, M., & Mesbah, A. (2015, November). Detecting inconsistencies in multi-platform mobile apps. In IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), 02 - 05 Nov. 2015; Gaithersbury, MD, USA. IEEE. pp. 450-460.

Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W.,& Wimmer, M. (2006). Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In Proceeding of International Conference on Model Driven Engineering Languages and Systems (MODELS), .528-542. Springer.

Khomh, F., Di Penta, M., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering, 17(3), 243-275.

Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D., & Guéhéneuc, Y. G. (2014, June). Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In Proceedings of the 22nd International Conference on Program Comprehension02 – 03 June 2014, Hyderabad, India. New York, NY, USA: ACM. pp. 232-243.

Maurice, P., Dhombres, F., Blondiaux, E., Friszer, S., Guilbaud, L., Lelong, N., ... & Jurkovic, D. (2017). Towards ontology-based decision support systems for complex ultrasound diagnosis in obstetrics and gynecology. Journal of gynecology obstetrics and human reproduction, 46(5), 423-429.

Morales, R., Saborido, R., Khomh, F., Chicano, F., & Antoniol, G. (2016). Anti-patterns and the energy efficiency of Android applications. arXiv preprint arXiv:1610.05711.

Obrst, L., Grüninger, M., Baclawski, K., Bennett, M., Brickley, D., Berg-Cross, G., & Lange, C. (2014). Semantic web and big data meet applied ontology. In Applied Ontology, 9(2), pp. 155-170.

Parnas, D. L. (1994, May). Software aging. In Proceedings. Of 16th International Conference on Software Engineering ICSE-16., 16-21 May; Sorrento, Italy. Los Alamitos, CA, USA: IEEE. pp. 279-287.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., & De Lucia, A. (2017, February). Lightweight detection of Android-specific code smells: The a Doctor project. In IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 20-24 Feb. 2017, Klagenfurt, Austria. IEEE. pp. 487-491.

Raja, V. (2008). Introduction to reverse engineering. In Reverse Engineering, 1-9. Springer, London.

Romano, D., Raila, P., Pinzger, M., & Khomh, F. (2012, October). Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In 19th Working Conference on Reverse Engineering (WCRE), 15-18 Oct.; Kingston, ON, Canada. Washington, DC, USA: IEEE. pp. 437-446.

Song, L., Tang, Z., Li, Z., Gong, X., Chen, X., Fang, D., & Wang, Z. (2017, December). AppIS: Protect Android Apps Against Runtime Repackaging Attacks. In 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 25-32. IEEE.

Yus, R., & Pappachan, P. (2015, October). Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications. In Mobile Deployment of Semantic International Workshop MoDeST@ ISWC, Bethlehem, PA, USA. pp. 2-13.

Zhou, X., Wu, K., Cai, H., Lou, S., Zhang, Y., & Huang, G. (2018). LogPruner: detect, analyze and prune logging calls in Android apps. Science China Information Sciences, 61, pp. 1-3.

Table 1(on next page)

The description of the mobile apps under analysis

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Mobile App Name	Size(MB)	Downloads
Test DPC 4.0.5	3.14 MB	1.076.791
Avast 6.5.3 Security	20.71 MB	1.364
Free-Calls-Messages	31.59 MB	1.537
Beautiful Gallery 2.3	11.31 MB	497
Play Store 9.3.4	14.17 MB	6.950
Wall Paper 1.2.166	2.29 MB	9.730
Oasis-Feng/Island 2.5	2.34 MB	822
Netflix-5-4-0-Build	18.81 MB	22.043
Remainder 1.4.02	9.36 MB	3.612
Sound-Picker 8.0.0	3.9 MB	2.142
Air-Command 2.5.15	0.82 MB	1.747
Lifesum-Healthy-Lifestyle	31.4 MB	3.594
Background-Defocus 2.2.9	3.45 MB	2.960
Gasbuddy-Find-Cheap-Gas	29.64 MB	334
Soundcloud -Music-Audio.03.03	33.2 MB	2,066
Network-Monitor-Mini 1.0.197	2.88 MB	307
Casper Android 1.5.6.6	18.77 MB	383.765
Line 8.4.0	70.25 MB	260
Diagnosises	6.96 MB	36
Viber 7.7.0.21	38.4 MB	1.628
Whats App 2.17.235	35.81 MB	28.978
Firefox 56.0	40.62 MB	20.423
Blue- Email And Calendar 1.9.3.21	43.2.4 MB	203
Google Camera 5.1.011.17	36.48 MB	211.822
You Tube 13.07	24.13 MB	23.667
True Caller 8.84.12	23.09 MB	609
Samsung Gallery 5.4.01	17.61 MB	10.712
Twitter 7.48.0	35.82 MB	694
Chrome Browser 66.0.3359	41.51 MB	29.129

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Table 2 (on next page)

Ten Modelio anti-patterns and their correction way

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

The anti-pattern	Correction way
NameSpaces have the same name.	Change the name of the conflicting *NameSpaces*
NameSpace is Leaf and is derived.	Make the *NameSpace* non-final.
NameSpace is Leaf and is abstract.	Make the NameSpace non-final.
Generalization between two un-compatible elements.	Change the source or the target in order to link two compatible elements.
A public association between two Classifiers one of them is public and the other has different visibility.	Change the visibility of the target class to public.
Classifier has several operations with the same signature.	Rename one of the *Operations* or change their parameters.
Classifier has attributes with the same name.	Rename the *Classifier*s *Attributes*.
MultiplicityMin must be inferior to MultiplicityMax.	Change the value of the minimum multiplicity to be less than the maximum multiplicity.
The status of an Attribute is abstract and class at the same time.	Set only one of the statuses to true.
A destructor has parameters.	Remove these parameters, or remove the destructor stereotype from the method.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Table 3(on next page)

OntoUML anti-patterns and the correction way

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

The Anti-pattern	The Correction Way
Association Cycle.	Chang the cycle to be closed or open cycle.
Binary relation with Overlapping Ends.	Declare the relation as anti-reflexive, asymmetric and anti-transitive.
Imprecise Abstraction.	Add domain-specific constraints to refer to which subtypes of the association end to be an instance of the other end may be related.
Relation Composition.	Add OCL constraints which guarantee that if there is a relation between two types and one of them has subtypes, there must be constraints says that the subtypes are also in a relation with the other type.
Relation Specialization	Add constraints on the relation between the type and the super-type, declaring that the type is to be either a specialization, a subset, a redefinition or disjoint with relation SR

1

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Table 4 (on next page)

Occurrences of the anti-patterns in the mobile apps

The Group	%of occurrences across models	Total # of occurrences
Anti-patterns in Attributes	0.7%	9
Anti-patterns in Namespaces	7.21%	91
Anti-patterns in Operations	%0.3	5
Anti-patterns in Associations	%44.89	554
Anti-patterns in the Class	%47.7	603
Total		1262

1

Table 5(on next page)

The anti-patterns in each app

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

	Mobile App	CHSO	NHSN	NLAD	NLAA	GBUE	CHSA	MMITMM	PACPP	SAAC	TDHPS	BinOver	AC	RS	RelComp	ImpAbs	Total
1	Test DPC 4.0.5	7	2	1			2		1			10	6	1			**30**
2	Avast Android Mobile Security	149	15		2		58	3		2		6			2	3	**240**
3	Free-Calls-Messages	4	1	2		2	1	1				3	2				**16**
4	Beautiful Gallery 2.1	5	2						1								**8**
5	Play Store	8	1	1		1	1	2	3	1		6	16		41	2	**82**
6	Wall Paper	1	1					1							4		**7**
7	Oasis-Feng/Island	17	2				4									3	**26**
8	Netflix-5-4-0-Build	60	7		2						5	5					**79**
9	Remainder	11	4				1	2				4	7	5			**34**
10	Sound-picker	9	1													2	**12**
11	Air-Command	8	1			1											**10**
12	Lifesum-Healthy-Lifestyle	5	1					1		4			5	1	2	2	**21**
13	Background-Defocus	10	4				4	1				10			6		**35**
14	Gasbuddy-Find-Cheap-Gas	11	4		1		2			1			7	2		3	**31**
15	Soundcloud -Music-Audio	6	4						2					8	1	2	**23**
16	Network-Monitor-Mini	7	2				1	2					3				**15**
17	Casper Android	6	4						3			20		6			**39**
18	Line	15	1				1	1					6		2	1	**27**
19	Diagnoses	1											2	1			**4**
20	Viber	42	4			1	1			1		9		7	5		**69**
21	Whats App	5	1					2				30		2		2	**42**
22	Firefox	40	4				1	1	4					8		1	**59**
23	Email And Calendar	15	2				1					108	2				**128**

24	Google Camera	9	1						1			15	8		1	1	**36**
25	You Tube	21	4				3					3	3		3	2	**39**
26	True Caller	31	2					2				17	5		1		**58**
27	Samsung Gallery	12							1				9	3		1	**26**
28	Twitter	6	2					1				15	6	1	1		**32**
29	Chrome Browser	1	4	1								9	5		12		**32**
#of appearance		**522**	**81**	**5**	**5**	**5**	**81**	**20**	**16**	**9**	**5**	**270**	**92**	**45**	**81**	**25**	**1262**

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Table 6(on next page)

The correlation among anti-patterns groups

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Anti-patterns	Correlation Coefficient(r)
Attributes & Namespaces	-0.049
Attributes & Operations	0.884
Attributes & Associations	0.196
Attributes & Classes	0.342
Namespaces & Operations	-0.060
Namespaces & Associations	-0.121
Namespaces & Classes	0.010
Operations & Associations	0.345
Operations & Classes	0.267
Associations & Classes	0.070

1

Table 7(on next page)

The correlation among the three tools

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Systems	Correlation Coefficient(r)	Specification
Modelio & OntoUml	-0.032	There is a reverse correlation between Modelio and OntoUml.
Modelio & Protégé	0.966	There is a direct correlation between Modelio and Protégé.
Protégé & OntoUML	-0.060	There is a reverse correlation between Protégé and OntoUml editor.

1

Table 8(on next page)

Anti-patterns number before and after merging

Mobile Apps	Viber	Whats App	The integrated app	Total
(Merging UML designs) # of detected anti-patterns in first method using Modelio	49	8	58	115
(Merging Ontologies) # of detected anti-patterns in second method using Protégé	51	64	13	128

1

Figure 1

The proposed method phases

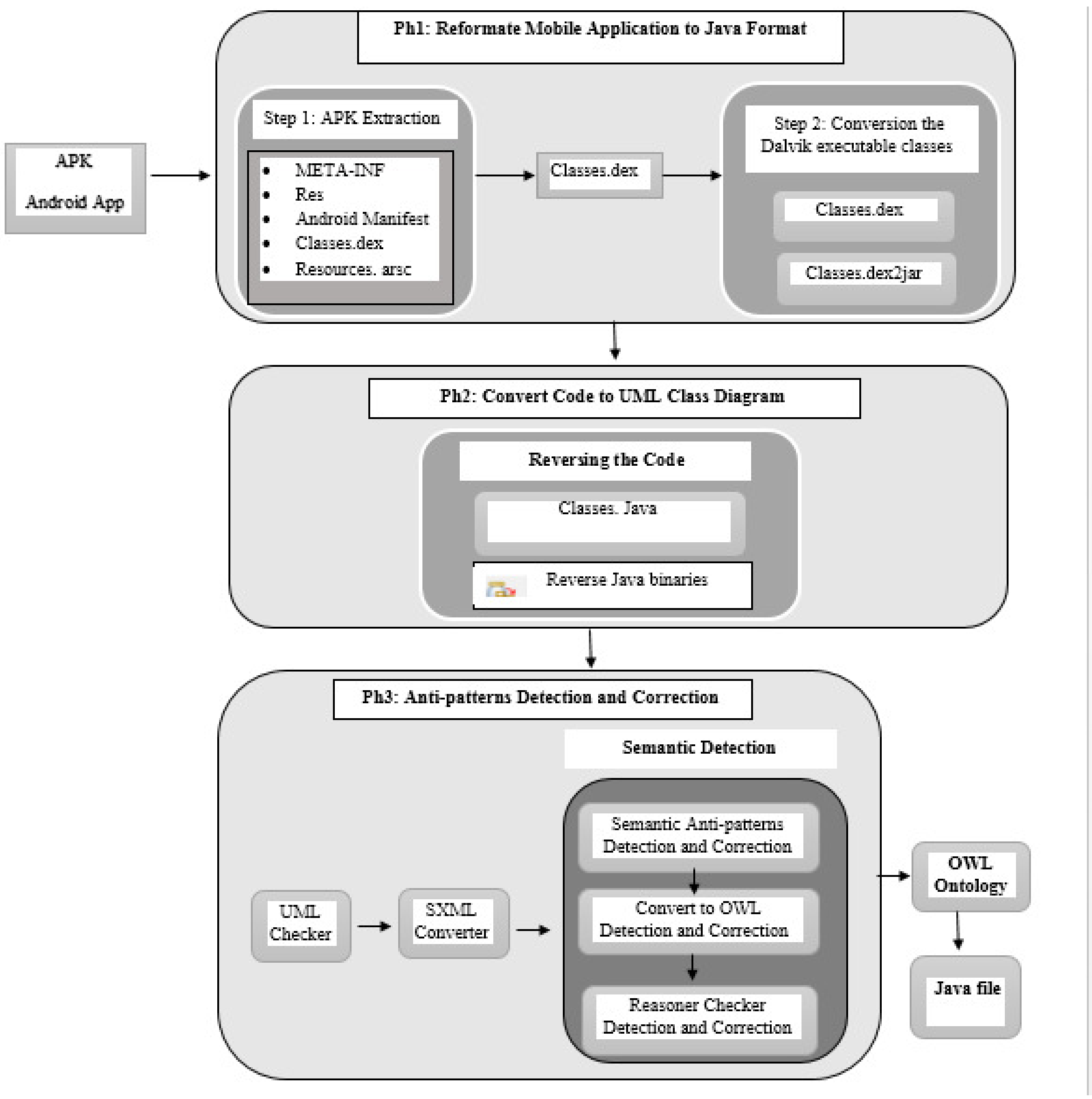


Figure 2

Merging UML class diagrams of the mobile apps

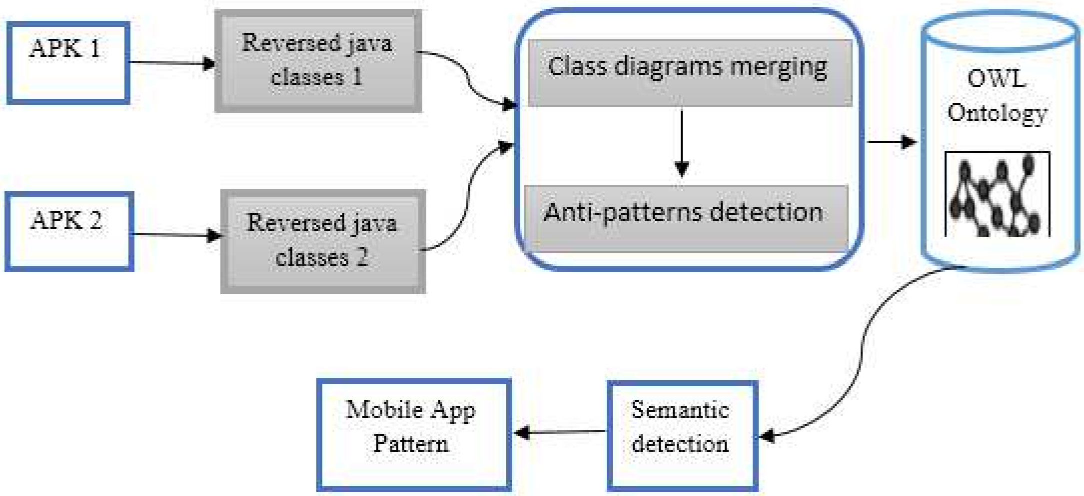


Figure 3

OWL Ontology merging

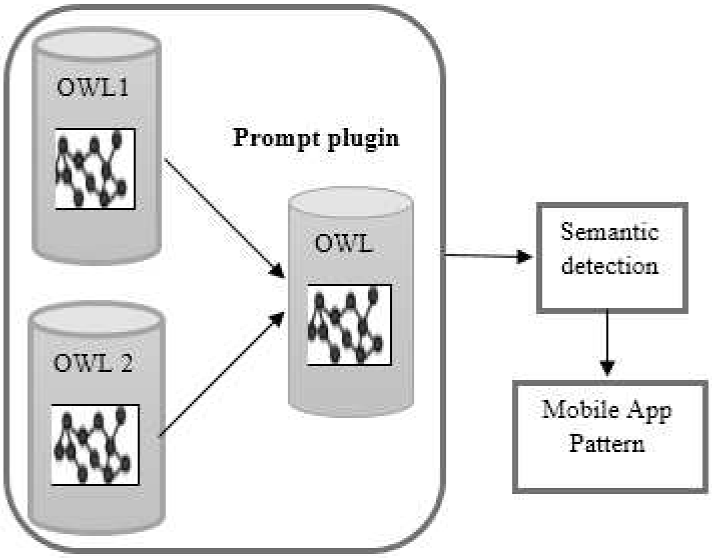
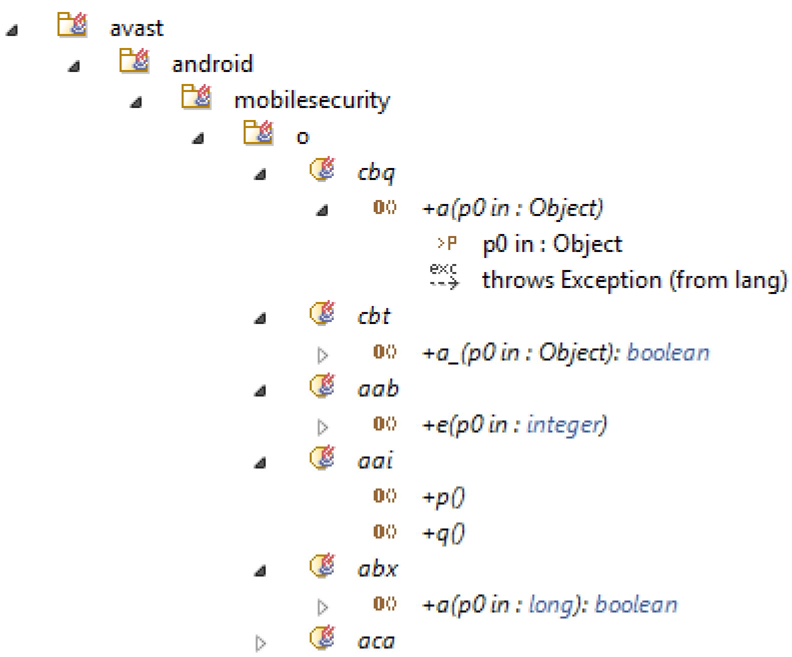


Figure 4

The generated UML class diagram of the case study



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

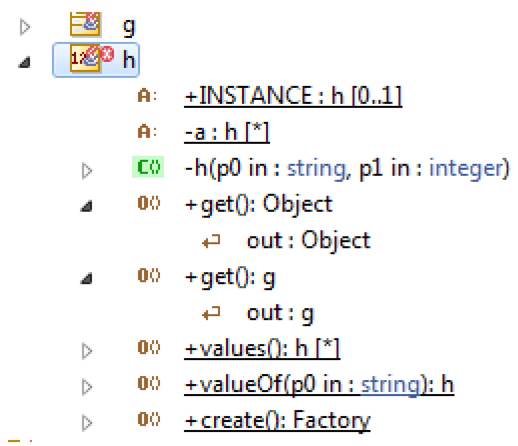
Figure 5

Modelio anti-patterns

a	R1980	The Classifier 'a' has at least two Attributes or two AssociationEnds with the
ViewDecorator_	R2260	The Classifier 'ViewDecorator_Factory' has several operations with the same
a	R2260	The Classifier 'a' has several operations with the same signature.
bxo	R2060	There are several namespaces with the same name in the namespace 'bxo'.
ra	R2260	The Classifier 'ra' has several operations with the same signature.
a	R2260	The Classifier 'a' has several operations with the same signature.
ProviderOfLazy	R2260	The Classifier 'ProviderOfLazy' has several operations with the same signatu
a	R2260	The Classifier 'a' has several operations with the same signature.
b	R2260	The Classifier 'b' has several operations with the same signature.
a	R2260	The Classifier 'a' has several operations with the same signature.
Buffer	R2260	The Classifier 'Buffer' has several operations with the same signature.
il	R2260	The Classifier 'il' has several operations with the same signature.
o	R2260	The Classifier 'o' has several operations with the same signature.
StatementExecu	R2260	The Classifier 'StatementExecutor' has several operations with the same sig
f	R2260	The Classifier '*f*' has several operations with the same signature.
c	R2260	The Classifier 'c' has several operations with the same signature.

Figure 6

The anti-pattern " Classifier has several operations with the same signature"



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Figure 7

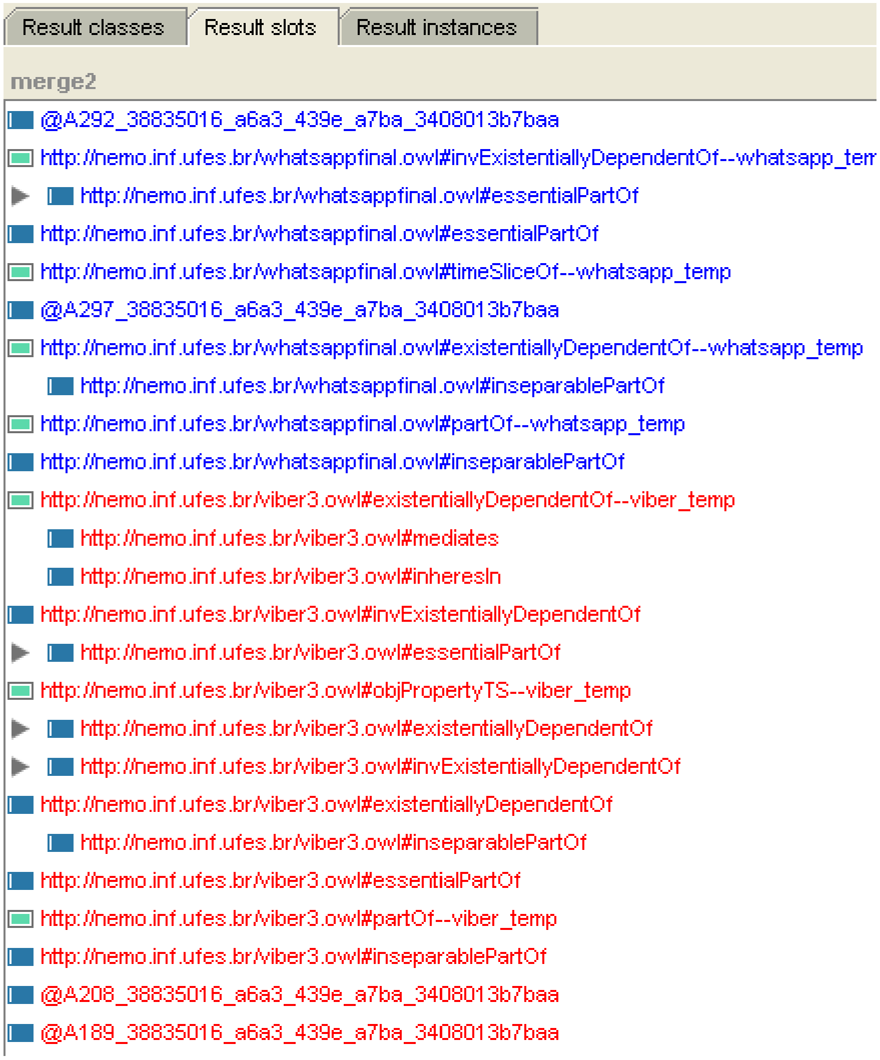
"Viber and Whatsapp" Ontologies before integration in Protégé

Name	Arg1	Arg2
merge	http://nemo.inf.ufes.br/viber3.owl#IndividualCc	whatsappfinal:IndividualConcept *whatsapp*
merge	viber3:TimeSlice *viber*	whatsappfinal:TimeSlice *whatsapp*
merge	viber3:TemporalExtent *viber*	whatsappfinal:TemporalExtent *whatsapp*
merge	viber3:Object *viber*	whatsappfinal:Object *whatsapp*
merge	viber3:Moment *viber*	whatsappfinal:Moment *whatsapp*
merge	viber3:Moment *viber*	whatsappfinal:Mode *whatsapp*
merge	viber3:FunctionalComplex *viber*	whatsappfinal:FunctionalComplex *whatsapp*
merge	viber3:Collective *viber*	whatsappfinal:Collective *whatsapp*
merge	viber3:Collective *viber*	whatsappfinal:CollectionsTS *whatsapp*
merge	viber3:Quantity *viber*	whatsappfinal:Quantity *whatsapp*
merge	viber3:FunctionalComplexTS *viber*	whatsappfinal:FunctionalComplexTS *whatsap*
merge	viber3:CollectiveTS *viber*	whatsappfinal:CollectiveTS *whatsapp*
merge	viber3:CollectiveTS *viber*	whatsappfinal:CollectionsTS *whatsapp*
merge	viber3:QuantityTS *viber*	whatsappfinal:QuantityTS *whatsapp*
merge	viber3:Relator *viber*	whatsappfinal:Relator *whatsapp*
merge	viber3:RelatorTS *viber*	whatsappfinal:RelatorTS *whatsapp*
merge	viber3:Mode *viber*	whatsappfinal:Moment *whatsapp*
merge	viber3:Mode *viber*	whatsappfinal:Mode *whatsapp*
merge	viber3:ModeTS *viber*	whatsappfinal:ModeTS *whatsapp*

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Figure 8

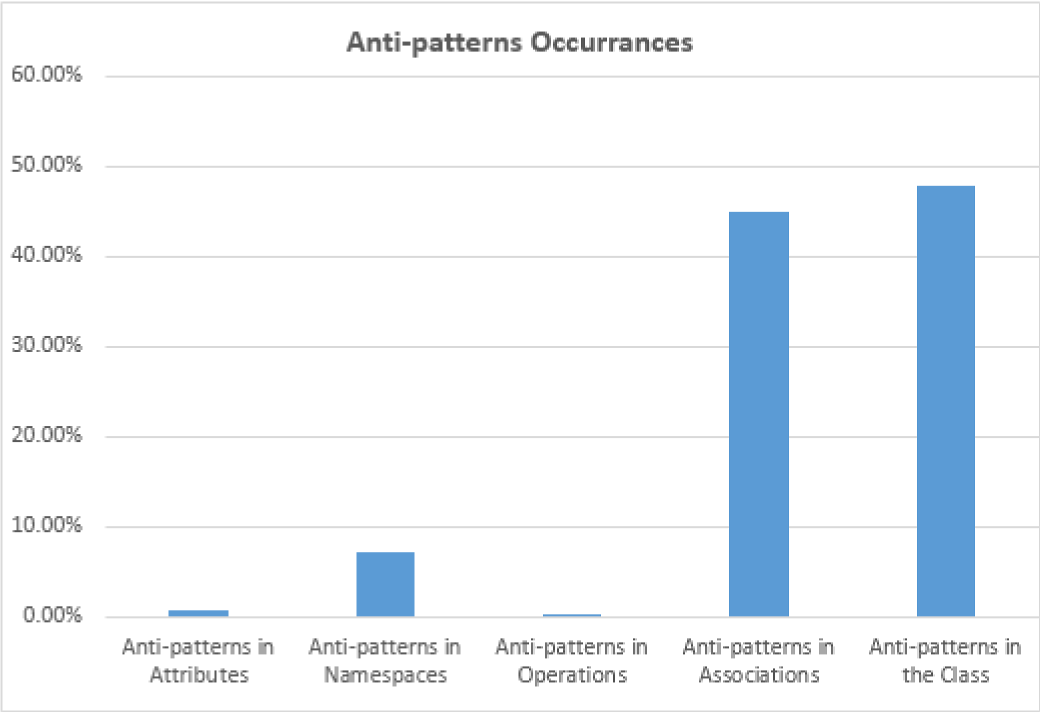
The result slots of the Ontology after integration in Protégé



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Figure 9

The occurrences of the detected anti-patterns' groups



PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27633v1 | CC BY 4.0 Open Access | rec: 4 Apr 2019, publ: 4 Apr 2019

Figure 10

The one way ANOVA between the tools and the anti-patterns

ANOVA

		Sum of Squares	df	Mean Square	F	Sig.
OLED	Between Groups	2724.910	28	97.318	.623	.926
	Within Groups	18131.200	116	156.303		
	Total	20856.110	144			
Modelio	Between Groups	9914.593	28	354.093	1.009	.464
	Within Groups	40720.000	116	351.034		
	Total	50634.593	144			
Protoge	Between Groups	9268.634	28	331.023	.925	.578
	Within Groups	41511.600	116	357.859		
	Total	50780.234	144			
anti.pattern.types	Between Groups	.000	28	.000	.000	1.000
	Within Groups	290.000	116	2.500		
	Total	290.000	144			