Impact of code smells on open source android application's rating in the Google playstore

Ivan Tactuk Mercado, Omeya Sandeep Jadhav and Dipti Menda

Abstract— **Recently the popularity of android applications has increased tremendously. Android applications are evolving with addition of many features to keep their users engaged and involved. Google playstore is used as the official and the primary medium to publish applications. With million of applications into the playstore the ratings of an android applications is being used as an quality evaluation criteria by the users before downloading any app. Thus, to keep the android rating high and stable has been the major challenge and concern for the developers and app owners. As the app evolves over a period of time there might be issues with respect to maintainability, as code smells gets ingested into the application. These code smells may lead to future bugs and degrade the popularity of an app, which may lower the playstore rating due to user dissatisfaction. This paper provides a research whether does the presence of code smells really affects the ratings of an android application on the playstore when it evolves over a period of time? We have analyzed and answered several research questions. Our research paper reveals some interesting facts related to the code smells, bugs and evolution of android apps.**

I. INTRODUCTION

Android mobile application development is done on open source platform using android SDK. There are more than one million applications with more than 25 billion downloads from Google play store [3]. From small individual developers to large companies are into development of the apps and games. The rating and comments of the apps are the two important criteria for the popularity and success of an app. Android applications are written in Java programming language and use Android Software Development Kit (SDK). Thus, the quality of an android application with respect to the coding standard is crucial for performance of the app on the Google play store. We are analyzing how improper implementation of coding standard and presence of code smells can impact the life of an app on the play store. Code smells are the surface indication that usually corresponds to a deeper problem in a software system [8]. Rating of an application is a nice measurable criteria for the quality and popularity of an android application. As application evolves the complexity of the app increases which can result in code smell if proper coding guidelines are not followed while designing it. While there are many factors which can affect the ratings of android applications such as features, user interface, ease of use, etc., code smells affects the maintainability, testing, design and evolution of an software system which can result in future bugs affecting the quality of application and lead to the decline in popularity of the app. The decline in popularity of an app can lead to poor ratings. In this paper we present the result of our research on whether the presence of code smells can affect the ratings of android apps. For the experiment we have used 50 open source android applications from the Google Play Store. We tested all the versions of the 50 android applications using code smell detection tool "inFusion"[1]. Infusion code smell detection provided us the Lines of codes(LOC) and code smells in each version of the application. We have used F-Droid repository to get the source code of the android apps. In our experiment we have measured the correlation between code smells and bugs. All the applications which we have used in this experiments have more than 1000+ ratings. The code smells which we have targeted using the infusion code smell detector are as follows:

- Brain Class: This class accumulates a lot of intelligence and a lot of methods can be affected by making changes in the brain class. Brain classes are therefore difficult to understand and maintain.
- Brain Method: This method centralizes a lot of functionality of a class or the complete software system. Any modification done in this method can affect the entire class or system.
- Data Class: These classes are just the dumb data holders that contains nothing other than the fields and getter, setter methods. In spite of this fact other classes try to alter the data classes in far too detailed manner.
- Data Clumps: Data items tend to appear together in many places like within the fields in many classes or within the parameters in many method signatures. These data items are nothing but data clumps.
- Duplicate Code: It is the repetition of same or similar code structures in multiple places within a program.
- Feature Envy: Condition of Feature envy occurs when a method is in a class other than the class where it actually should have been, i.e the class which is excessively using that method.
- God Class/Large Class(DECOR): God class is a class that has too many responsibilities incorporated in it compared to the other classes that it is linked with.
- Intense Coupling: This situation arises when methods call many other methods that are dispersed only into few classes.
- Message Chain: In message chain, a client asks the first object for the second object, the second object asks for the third and so on in chain. If any changes will take place at the intermediate level, the client might have to undergo changes.
- Shotgun Surgery: If some changes are made in a certain

[1]"https://www.intooitus.com/products/infusion"

class, it necessitates though minute but several changes in many other classes.
- Tradition Breaker: This code smells occurs when a class increases in size and provides a lot of functionality different and unrelated to the one the base class provided.

These are the most common code smells present in the apps which can affect the maintainability, performance and quality of apps. The result of our experiment showed that maintainability could be an issue with the evolution of android application but presence of code smell does not show any impact on the ratings of android applications.

II. Motivation

We understand that maintainability of the apps is important and care should be taken during the process of evolution of the apps. The maintainability could be increased by understanding the effects of code smells and how it can affect the developer as well as the end user. For big projects with hundreds of developers contributing it can happen that code smells can make maintainability difficult [2]. In this paper we try to see if in the case of android apps does the code smells affects the quality of the apps. Thus, considering the importance of following proper coding standards to overcome the issues we decided to use ratings as an important factor to determine the success of android apps. The degradation in quality of an app by intrusion of code smells, the app can began susceptible to bugs. These bugs can lead to damage the quality of the apps which can lead to bad ratings. We analyzed the problem statements using three research questions as follows:

- RQ1: Does the presence of code smells affect the success of an android app?
- RQ2: Is it necessary for the developers to be aware of code smells by following a proper coding standard and refactor their code wherever required.
- RQ3: Does the use of bad programming practices has an impact on the bugs reported in an android application?

III. HYPOTHESIS

- Successful apps have less code smell and applications with more code smells have less rating
- If an application reduces its amount of code smells, an improvement in its rating is expected.
- The less code smells identified in an app, the less bugs reported in the app.

IV. EXPERIMENT

We analyzed the source code of 464 versions of 50 open source android apps. The source codes of all the apps were downloaded from the repository provided by F-droid[2]. We acquired the issue tracker from the F-Droid repository. We obtained the track of ratings of some released versions of android apps from [9]. They started crawling Google Play Store website since 2011. They did it once a day during the first half of 2011 and twice a day during the second half. The ratings of versions of app was maintained. We used the code smell detection tool inFusion to find the code smells from the downloaded source code. After all the data was collected, we made a linear regression model of code smells and ratings and another one of code smells and bugs. We plotted a graph for each code smells, which contains the correlation of the density of a code smells. A more detailed explanation of these procedures will be presented in the following subsections.

[2]F-droid is an online catalog of free and open source android apps. Link: https://f-droid.org/

A. Collecting Data

At the beginning, we had a list of apps whose ratings were collected in [9]. F-droid did not have the source code of all those applications. For this reason, a web scraping technique was used to match those applications of which we had both the source code and the ratings. This resulted in a list of 99 applications. We have analyzed 50 applications due to time constraint. Some of these applications had too many versions that were created and were in very narrow time frame. Some applications have more than one version per month. The amount of changes that can be performed to an application in such a small period time is very likely that it does not affect the number of code smells presented in its source code. Hence, to avoid testing unnecessary versions, only one version per month was analyzed for those applications that have multiple versions in a month. Then, we proceeded to download all the source code of the versions of these applications from the repository supplied by f-droid.

At first, we thought of using multiple code smell detection tools for analyzing all the applications. However, according to [1], the detection tools, that we thought we could use, showed a very high level of agreement (between 85% and 100%) in most of the code smells detected, except for long method (84.28% for the worst case tested). Among all the available code smells detection tools, inFusion was the one that detects more code smells. For this reason, we used inFusion to identify the code smells in each version of each application. The number of lines of code was also collected from inFusion. The creation date of a specific version was taken from its repository.

B. Correlation analysis

After all the data was collected, the next step was to map each of the code smell's samples to a version rating sample. In other words, we had two data sets: one with the ratings (Data set A) and another one with the code smells and LOC (Data set B). Both data sets have a date field. Data set A contained the date in which the version was uploaded to the Google Play Store. Data set B had the date in which the version was created. Mapping these two data sets where both date fields were equals was not the best choice, because the date, in which an application was uploaded to the Google Play Store, does not necessarily match with the date it was created in the repository. So we decided to map each rating sample(Data set A) to the closest previous code smells sample (Data set B). Another thing that made this way of mapping a better choice is that code smells influences maintainability, their impact in the rating of an application will not materialize immediately. So assuming that a rating corresponds to the closest previous version in the repository was an acceptable solution.

In the rating data set, the aggregated rating was available, but not the version rating. The aggregated rating is the cumulative rating that the application have accrued since the beginning. The aggregated rating includes the rating of the previous versions. Meanwhile, the version rating is the rating of only one version. To calculate the version rating, the following formula was used:

$$VR_i = \frac{AR_i \times RC_i - AR_{i-1} \times RC_{i-1}}{RC_i - RC_{i-1}} \quad (1)$$

In equation 1, VR_i is the version rating for the current version. AR_i is the aggregated rating of the current version. RC_i is the rating count or the number of votes in total of the current version. AR_{i-1} is the aggregated rating of the previous version. RC_{i-1} is the rating count of the previous version. For the first version we assumed that the aggregated rating is equals to the version rating, because it was not possible to obtain a previous aggregated rating for this version.

Since it is more likely that a longer code has more code smells, density of code smell count was decided to use instead of using the count of each code smells. For example, if a project only has one class, only one class can be a brain class. However, if an application has 100 class, it could have as much as 100 brain class. Using code smells density, will help to do a more fair comparison between applications, because it measure how pervasive was a specific code smell. For some code smells, using number of class instead of LOC could be a more accurate measurement. Nevertheless, previous experiments have demonstrated that lines of codes are strongly correlated to the number of classes. We decided to use LOC to calculate the code smells density of all the code smells. To avoid obtaining long decimals in our results, a thousand lines of codes (KLOC) to measure the number of lines of code. The code smells density was calculated as follows:

$$CD = \frac{CC}{KLOC} \quad (2)$$

In equation 2, CD is the Code Smell density. CC is the count of code smells. KLOC is the number of thousand lines of code. CD was calculated for each type of code smell.

After the version rating and each code smell density was calculated and each rating was map to a source code, we proceeded to compute the correlation coefficient between the density of each code smell in a version's source code and the rating of the corresponding version. The correlation coefficient was calculated for each version of each application. Each application had a correlation coefficient for each code smells. Some applications did not present some code smells or the code smells stayed the same over the application evolution, so no correlation coefficient was obtained from those applications. If a certain code smells have a constant value over all the versions of an application, the correlation coefficient will be undefined.

After all defined correlation coefficients were calculated for each code smell and for each application, seven graphs were plotted. One graph for each code smell with more than three apps with a defined correlation coefficient. Some code smells were not very pervasive in all the applications that were tested. A graph was plotted for the following code smells: Data Class, Data Clumps, Duplicated Code, Feature Envy, God Class, Intensive Coupling and Tradition Breaker. On the other hand, for Shotgun Survey, Refused Parent Bequest and Message Chains no graph was plotted, because there are less than three applications with a correlation coefficient that was defined.

Calculating the correlation coefficient between the density of code smells and the version rating will help to address RQ1 and RQ2. A positive relation between these two variables will imply that as the density of code smells of a application increases, its rating also increases. Negative correlation coefficient will imply the opposite.

C. Linear Regression Model analysis

Six linear regression models were created. Three of those models has Ratings as its dependent variable. The other three have the number of bugs as its dependent variable. In all the cases, the independent variable was some sort of variation of the number of code smells in the evolution of an application. For each code smells and for each application, the average (3), standard deviation (4) and the growth of code smell (5) was calculated of all the version.

$$\overline{CC} = \sum_{i=0}^{N} \frac{CC_i}{N} \quad (3)$$

In equation 3, $\overline{CC}$ represents the average of a specific code smell of all the version of each application. N is the total number of versions of a specific application. CC_i is the number of code smells detected by inFusion in the source code of a version of an application.

$$s = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N}(CC_i - \overline{CC})^2} \quad (4)$$

In equation 4, s is the standard deviation of the number of a code smell in the source code of all the versions of a application. N is the total number of versions of a specific application. $\overline{N}$ represents the average of code smells in the versions in a app. CC_i is the same as in equation 3.

$$CG = CC_N - CC_0 \quad (5)$$

In equation 5, CG is the growth of the number of a specific code smell since the earliest available version to the latest available version. CC_N represents the number of code smells detected in the latest version of an application. CC_0 is the number of code smells detected in the earliest version of an application.

In this analysis, the rating variable was the aggregated rating of the latest version of each application. The aggregated rating accounts for the entire rating history of a given application summarizes all the votes that a given application has received from the beginning of the data sampling.

The following linear regression models were created:

- RATINGS $\sim$ AVERAGE CODE SMELLS: $\overline{CC}$ for each code smell as the independent variables and the latest aggregated rating as the dependent variable.
- RATINGS $\sim$ STD. DEVIATION CODE SMELLS: s for each code smell as the independent variables and the latest aggregated rating as the dependent variable.
- RATINGS $\sim$ GROWTH CODE SMELLS: GC for each code smell as the independent variables and the latest aggregated rating as the dependent variable.
- BUGS $\sim$ AVERAGE CODE SMELLS: $\overline{CC}$ for each code smell as the independent variables and the number of reported bugs as the dependent variable.
- BUGS $\sim$ STD. DEVIATION CODE SMELLS: s for each code smell as the independent variables and the number of reported bugs as the dependent variable.
- BUGS $\sim$ GROWTH CODE SMELLS: GC for each code smell as the independent variables and the number of reported bugs as the dependent variable.

V. RESULTS

Code smells are one of the factors that affects the maintainability of android apps. An app that is hard to maintain will be harder to get new features implemented and bugs will be created more easily causing user dissatisfaction and a lower rating for apps with many code smells. Our assumption is that as the number of code smell increases then the ratings of android apps decreases. Thus, the code smell affects the success of an android application and developers need to be aware of the consequences. To improve the quality of an app, developers should find the code smells using code detection tools and refactor the code to improve the maintainability.

In Figure 1, it is shown how many versions of an application have a specific code smell. This graph does not show the density of code smells, but how common a code smell is among the applications that were analyzed. The most pervasive code smells are Data Class, God Class and Feature Envy. These code smells appeared in 34.42%, 33.12%, and 28.79%, respectively, of the 462 versions analyzed. The least common code smells are Shotgun Surgery (1.30%), Refused Parent Bequest (2.38%) and Message Chains (1.30%). These three code smells are present in only a few number of apps (less than 3 apps). For this reason, a graph for the correlation coefficient of those three code smells and their ratings were not included in this paper. The graphs of those code smells only display like one or two applications.

The results obtained from this study does not imply causation of code smells in ratings, but it shows if a correlation exists between the behavior of both variables. A high rating of an application depends on several factors and code smells are just a phenomenon that could impact quality and thus impact how users will rate applications.

In figures 2, 3, 4, 5, 6, 7, 8, each of the red dots represents the correlation coefficient of one application. This graphs shows how correlated were the density of a given code smell and the rating during an application evolution. Some of these graph have less dots as it was expected, because if the number of code smells stayed constant through out the evolution of an application, the correlation coefficient of this application would be undefined, and thus no dot for this application would be plotted.

The correlation coefficient ranges from -1 to 1. The closer to zero this value gets, the less correlation between the presence of a given code smell and the rating in an application. If the correlation coefficient approaches to +1, the rating and the presence of code smells changes in the same direction, i. e., if the number of code smells increases, rating also increases. When the coefficient gets closer to -1, there is a inverse relationship between this two variables.

All the points are spread from -1 to +1 in figures 2, 3, 4, 5, 6, 7, 8. There is no strong tendency in the behavior of the applications. However, some applications have a high positive or negative value for the calculated correlation coefficient. A categorized analysis by type of application could lead to some other conclusion. Some code smells could be more harmful for some types of applications. It is also important to notice that the correlation coefficient was calculated between version rating and code smell count divided by LOC.

Table I, II, III, IV, V and VI reflect the results of the linear regression model analysis. This analysis shows how the presence and variation of a code smells impacts in the ratings of an Android application. The asterisk marks values that are statistically significant. In tables III and VI, there were no conclusive information.

According to table II, applications with a high density of feature envy got a higher rating. When we look at table I, applications with an unstable feature envy density obtained a higher rating. This is against our hypotheses, because there is a direct relationship between feature envy density and the rating received by an app. Feature envy is a code smells that occurs when a class is using very often a method from another class. A possible explanation for this phenomenon could be that successful applications make use of android framework classes very often and may be interpreted as feature envy classes. However, to be more accurate about the causes of these results a closer look to the source code of the application should be taken. According to tables IV and V, an unstable and high average density of Feature Envy implies a higher number of reported bugs. Thus, Feature Envy might cause bugs in Android applications, but these bugs could not impact much user's decision to give a low rating to an app.

An application with a high average and unstable density of Intensive Coupling received a lower rating. The exact reason for this behavior should be analyzed in more detail. It could be possible that this could smell makes maintainability harder. However, the information from the bugs analysis where not conclusive for this code smells.

According to table V, Data Class was the code smells

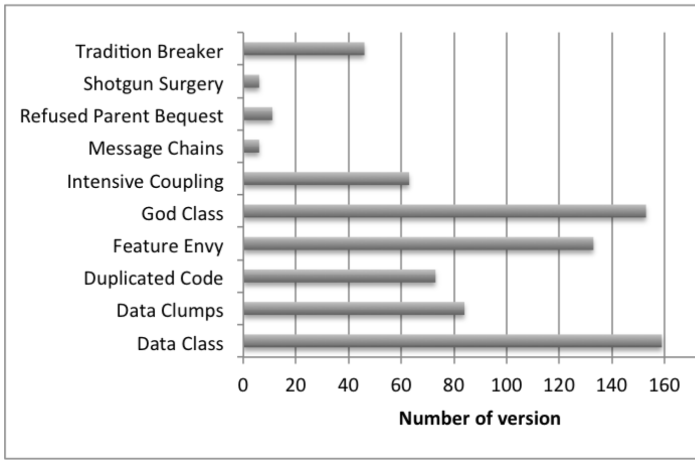

Fig. 1. Code smells vs. ratings

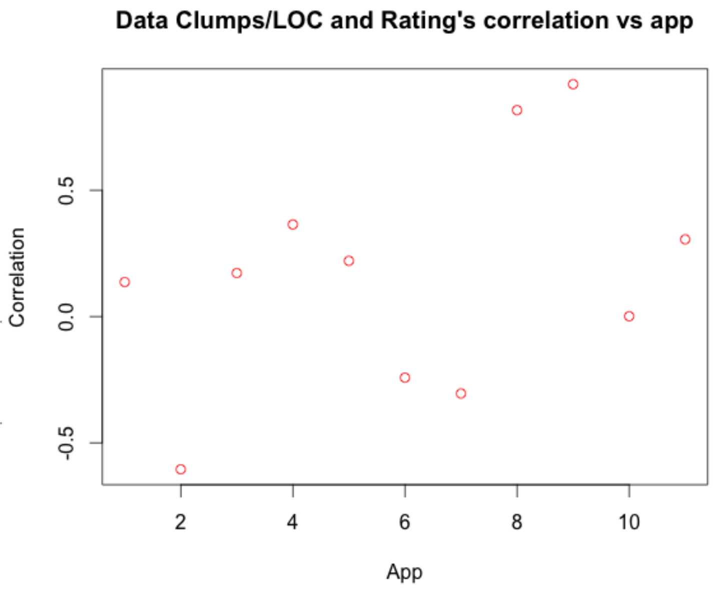

Fig. 3. Data Clumps density vs. ratings

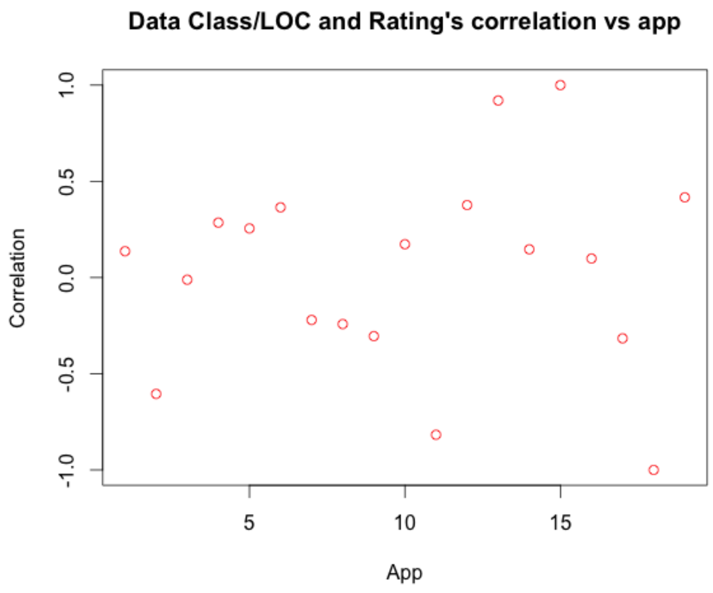

Fig. 2. Data Class Density vs. ratings

TABLE I
RATINGS ~ STD. DEVIATION CODE SMELLS

Code Smell	Estimate	p value
Data Class	0.028199	0.1565
Data Clumps	0.013661	0.1058
Duplicated Code	-0.006423	0.4892
Feature Envy	0.032785	0.0220 *
God Class	-0.077152	0.1024
Intensive Coupling	-0.0904	0.0474 *
Message Chains	-0.208975	0.6232
Refused Parent Bequest	0.112457	0.6081
Shotgun Surgery	0.300061	0.7237
Tradition Breaker	0.037053	0.2594

that had more impact in the reported number of Bugs. If the density of Data Classes in an application is high, it is likely that this Android application will have a high number of reported bugs. Feature Envy also have a similar impact, but in a lower scale. Meanwhile, Message Chains have an opposite effect. Applications with a high number of Message Chains presented a low number of reported bugs. This could imply that Message Chains are not harmful for Android applications.

Based on table IV, an unstable density of Data Class and Feature Envy is related to a high number of reported bugs. An unstable density of Duplicated Code and Message Chains have an opposite relation to the number of reported bugs. Having an unstable density of a code smell does not imply that the density of a code smell has increased or decreased.

VI. THREATS TO VALIDITY

We had certain threats to validity in our study. As a part of our experiment we decided to choose 50 open source android applications. This is extremely a small number considering all the android apps in the Google Play store. We made sure that the applications were taken from a valid source i.e the F-droid catalog. Some of the applications had about 400 versions. It was practically impossible to test all the versions because of the time limitations that we had,so we decided to consider one latest version per month. The decision was made upon the observation that the code smells remained almost the same for a month. All the application we choose were having 10000+ ratings. We decided to use Infusion as the only tool for code smell detection. Infusion was chosen because among all the other tools,it can detect maximum types of code smells and also it was compatible with both Windows as well as Mac.

VII. RELATED WORK

Daniel Verloop [3] has explained in his paper that mobile application is very different than traditional software

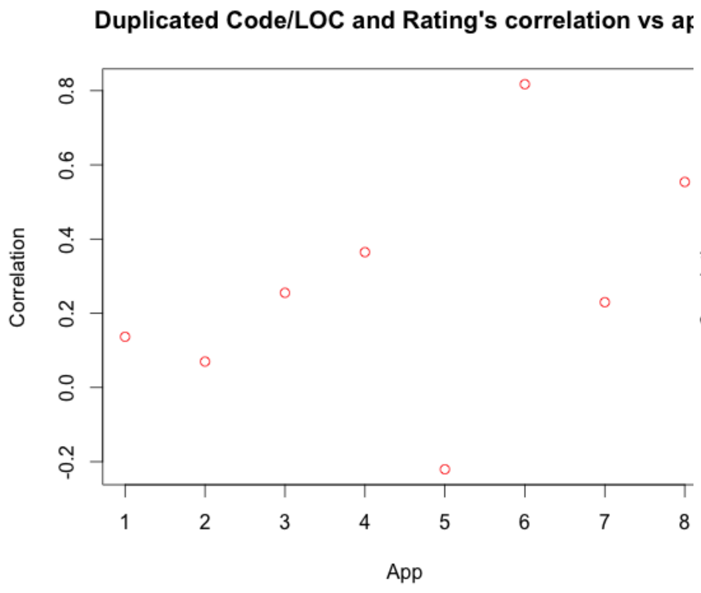

Fig. 4. Duplicated Code density vs. ratings

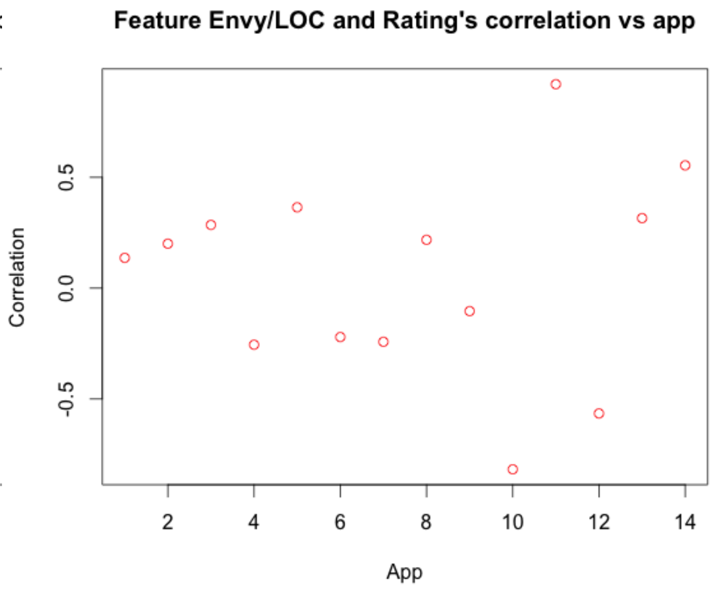

Fig. 5. Feature Envy density vs. ratings

TABLE II
RATINGS ∼ AVERAGE CODE SMELLS

Code Smell	Estimate	p value
Data Class	0.029849	0.159
Data Clumps	0.016349	0.0911 .
Duplicated Code	-0.008852	0.5611
Feature Envy	0.035539	0.0203 *
God Class	-0.084586	0.0847 .
Intensive Coupling	-0.10264	0.0451 *
Message Chains	-0.214842	0.6499
Refused Parent Bequest	0.18574	0.5506
Shotgun Surgery	0.213327	0.8437
Tradition Breaker	0.041447	0.2523

TABLE III
RATINGS ∼ GROWTH CODE SMELLS

Code Smell	Estimate	p value
Data Class	0.059972	0.566
Data Clumps	0.016701	0.582
Duplicated Code	0.003931	0.612
Feature Envy	0.008272	0.737
God Class	-0.108846	0.258
Intensive Coupling	0.031837	0.686
Message Chains	-0.159142	0.658
Refused Parent Bequest	-0.128832	0.81
Shotgun Surgery	NA	NA
Tradition Breaker	NA	NA

development in terms of less number of developers, short life span, regular updates, less processing speed and smaller size. He analyzed some of the tools which can be used for the detection of code smell in android domain. He also proved which code smells are more likely to appear in mobile applications than in desktop applications. He found that the Long Method, Large Class, Feature Envy and Type Checking code smells were more likely to be in mobile application than in non-mobile applications. The Long Parameter code smell was almost not presented in mobile applications. Dead Code was more common in non-mobile applications.

Steffen, Daniela and Dag [2] carried out an empirical study on three large scale and well known open source systems which are Log4j, Apache Lucene, and Apache Xerces. Their research was to see the impact of God Class and Brain Class on the quality of software system. They based their experiment on change frequency, change size and number of weighted defects and from the observations obtained,they came to the conclusion that as long as the two code smells are not large in size and numbers they could be beneficial for the system.

Steffen Olbrich, Daniela S. Cruzes, Victor Basili, Nico Zazworka [5] also study the impact of code smells. In this paper, they focused in two open source non-mobile applications (Apache Lucene and Apache Xerces 2 Java) and two code smells (God Class and Shotgun Surgery). To measure the impact they used change frequency and size.

Francesca Fontana, Pietro Braione, Marco [1] studied how effective automated code smell detection tools are,when there are thousands of LOC and manually finding code smells is difficult. We are using infusion amongst some of the tools suggested in their research. They studied how relevant are the smell detection tools in software development and their consistency in identifying code smells.

Aiko Yamashita, Leon Moonen [10] conducted a survey on 85 software developers around the world to know their knowledge on the code smells. They found that 32% of developers lacked the knowledge of code smells. The study revealed that developers need a better real time code smell detector tool which could analyze their code in the software development cycle. Such tool they feel can ease the work of refactoring.

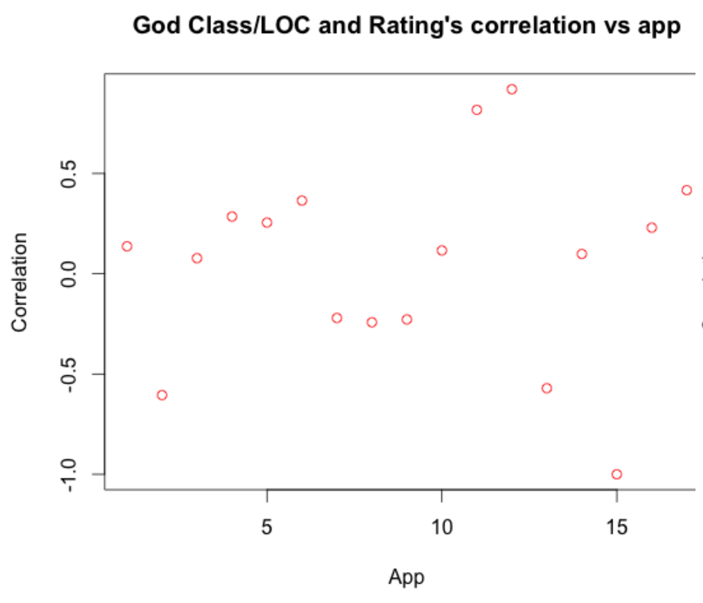

Fig. 6. God Class density vs. ratings

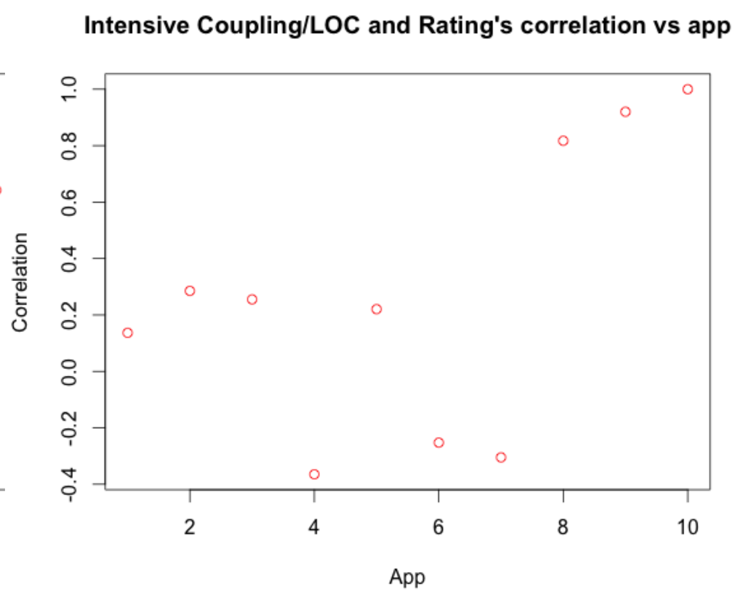

Fig. 7. Intensive Coupling density vs. ratings

TABLE IV
BUGS ∼ STD. DEVIATION CODE SMELLS

Code Smell	Estimate	p value
Data Class	50.836	0.00253 *
Data Clumps	11.516	0.18979
Duplicated Code	-9.047	0.04716 *
Feature Envy	20.966	0.00990 *
God Class	-33.992	0.23629
Intensive Coupling	-9.023	0.82881
Message Chains	-1389.418	0.01011 *
Refused Parent Bequest	-5.242	0.96043
Shotgun Surgery	NA	NA
Tradition Breaker	-41.609	0.56615

TABLE V
BUGS ∼ AVERAGE CODE SMELLS

Code Smell	Estimate	p value
Data Class	55.865	0.00497 *
Data Clumps	10.954	0.26361
Duplicated Code	-12.559	0.11784
Feature Envy	21.081	0.02536 *
God Class	-23.497	0.46969
Intensive Coupling	-28.267	0.6206
Message Chains	-1412.792	0.03267 *
Refused Parent Bequest	53.585	0.76006
Shotgun Surgery	NA	NA
Tradition Breaker	-82.123	0.45492

In [6], the authors introduced three improvements for code smells detection,the first was DECOR-a method that defines code smell detection technique,second was DETEX-for initialization of the above method,third applied Detex on 15 code smells.

In [7], an experiment was conducted to understand how the code smell detector Stench Blossom uses an interactive ambient visualization for finding out code smells.

In none of the related work, apps ratings were never use to verify if the impact, that code smells have in long term maintainability, also have an impact in the success of an open source android application in the Google Play Store.

VIII. CONCLUSION

Individual developers to giant companies are into the development of android apps. Android applications are being developed in variety of categories. In this paper our research postulates that the presence of code smells is not necessary the important factor for the bad ratings on the Google Play Store. But considering the drawbacks of code smells in programming language we conclude that as the app evolves the maintainability of app get difficult. As Google Play Store is very competitive with several apps are trying to make an impact, it is necessary to keep the users engaged by giving continuous updates. Thus, apps with code smells can have an impact if proper coding standards are not followed. We also found that many developers around the world need to understand the issues related to the ingestion of code smells and care should be taken to focus on maintainability of apps which could in future affect the apps success in evolution phases.

Data Class is the more common code smells in open source Android Applications. Applications with a high density of Data Classes are likely to have a higher number of reported bugs. For this reason, developers should avoid creating data classes.

Intensive Coupling is another code smells that developers should be concerned about since it had a strong relationship with the rating of an application.

REFERENCES

[1] Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. Journal of Object

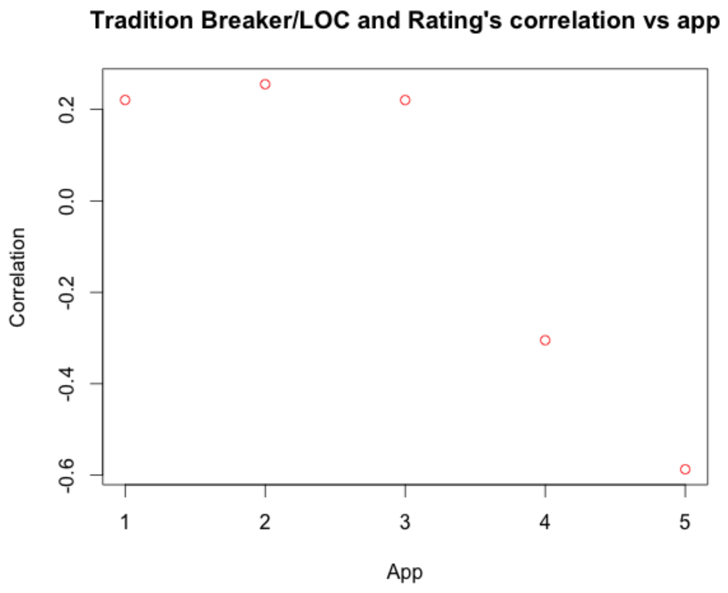

Fig. 8. Tradition Breaker density vs. ratings

TABLE VI
BUGS ~ GROWTH CODE SMELLS

Code Smell	Estimate	p value
Data Class	-42.895	0.51
Data Clumps	31.788	0.15
Duplicated Code	-7.433	0.25
Feature Envy	42.304	0.27
God Class	40.53	0.56
Intensive Coupling	-18.76	0.76
Message Chains	NA	NA
Refused Parent Bequest	-273.344	0.41
Shotgun Surgery	NA	NA
Tradition Breaker	NA	NA

Technology, 11(2), 5-1.
[2] Olbrich, S. M., Cruzes, D. S., & Sjoberg, D. I. (2010, September). Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In Software Maintenance (ICSM), 2010 IEEE International Conference on (pp. 1-10). IEEE.
[3] Verloop, D. (2013). Code Smells in the Mobile Applications Domain (Doctoral dissertation, TU Delft, Delft University of Technology).
[4] Hecht, G., Rouvoy, R., Moha, N., & Duchien, L. (2015). Detecting Antipatterns in Android Apps (Doctoral dissertation, INRIA Lille).
[5] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). The evolution and impact of code smells: A case study of two open source systems. In Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement (pp. 390-400). IEEE Computer Society.
[6] Moha, N., Gueheneuc, Y. G., Duchien, L., & Le Meur, A. (2010). DECOR: A method for the specification and detection of code and design smells. Software Engineering, IEEE Transactions on, 36(1), 20-36.
[7] Murphy-Hill, E., & Black, A. P. (2010, October). An interactive ambient visualization for code smells. In Proceedings of the 5th international symposium on Software visualization (pp. 5-14). ACM.
[8] http://en.wikipedia.org/wiki/Code_smell
[9] Mojica Ruiz, I., Nagappan, M., Adams, B., Berger, T., Dienst, S., & Hassan, A. (2014). On the Relationship between the Number of Ad Libraries in an Android App and its Rating.
[10] Yamashita, A., & Moonen, L. (2013, October). Do developers care about code smells? An exploratory survey. In Reverse Engineering (WCRE), 2013 20th Working Conference on (pp. 242-251). IEEE.