UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

TECHNICAL DEBT IN MOBILE APPLICATIONS

Agha Mohib

Submitted by: Agha Mohib - 0522500895

Table of Contents

1. Abstract:

In the recent years, the number of mobile applications have been growing which increasing the complexity in the development of high quality mobile apps. The continuous changing in a software with the new requirements or modifications and poor implementation in the code design may lead to technical debt. In this project, I am going to analyse technical debt in around 31 android apps by evaluating code-metrics during the development history, code-smells and architectural smells.

2. Introduction:

The increased complexity of modern android applications increases the difficulty of making the code reliable and maintainable. Technical debt (also known as code debt) describes when development teams take actions to accelerate the delivery of a project which later needs to be refactored. In other words, it is the result of prioritizing speedy delivery over perfect code. Code metrics is a set of software measures that provide developers better insight into the code they are developing. While, during the development, any symptom in the source code of a program that possibly indicates a deeper problem refers as code-smell which hindering software maintenance and evolution. Code smells are not bugs or errors but these are absolute violations of the fundamentals of developing software that decrease the quality of code. Smelly code contributes to poor code quality and hence increasing the technical debt. On the other hand, architectural smells may be caused by applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity.

3. Methodology of The Project:

In this project, my goal is to analyze the technical debt in our applications by finding code metrics, code smells and architecture smells. For code metrics and code smells, I used **Swanlyticcs** (a software analytics Java application) with RepoDriller (for mining android repositories) and for architectural smells, I used **DesigniteJava** (an open-source code quality assessment tool for Java-based applications).
For the analysis, I aggregated the data by calculating mean using MS Excel, as the tools give the output in csv file format.

3.1 Code Metrics:

In the code metrics, I evaluated three types of metrics according to CK as follows;

LOC: Size is the most obvious metric for source code. The number of lines of code (LOC) is the simplest way of measuring size. It is also used to estimate programming productivity or maintainability once the software is produced.

LCOM: LCOM (Lack of Cohesion in Methods) shows the degree of lack of cohesion among methods of a class. Cohesion means how well the methods of a class are related to each other. LCOM is an important measure because a high LCOM value means low cohesion.

CBO: Coupling between objects (CBO) is a count of the number of classes that are coupled to a particular class i.e. where the methods of one class call the methods or access the variables of the other. Increased coupling increases interclass dependencies, making the code less modular and less suitable for reuse.

3.2 Code Smells:

This project detects five types of code smells by using Swanlytics as following;

God Class: It is a kind of object that knows too much or does too much. In other words, a class contains huge number of lines of code (LOC). It is very famous in term of code smells because it creates tight coupling and increases the challenges in the code maintainability. If a class contains too many functions or properties, then it increases the lines of codes in that class and this class will be considered as God class.

Functional Decomposition: It refers as the output of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language. The resulting code resembles a structural language in class structure. In simple words, an incredibly degenerate architecture that completely misses the point of object-oriented architecture.

Spaghetti Class: It is a classic code smell that occurs when the code does not use appropriate structuring mechanisms. Spaghetti Code prevents the use of object-oriented mechanisms, namely polymorphism and inheritance. Classes affected by this smell, are characterized by complex methods, with no parameters and interaction between them using instance variables.

Complex Class: Classes having high complexity that means a class which contains complex methods and very large in terms of LOC. In simple words, more parameters/attributes in a method has, the more complex it is. Complex class is different from a Blob/God class because it exposes a high complexity, whereas the Blob is a singleton monopolizing all processing and data of a system

Class Data Should Be Private: This smell arises when a class exposes its attributes that means a class which attributes can be accessible from everywhere.

3.3 Architectural Smells:

For architectural smells detection, I used an open-source tool called DesigniteJava which detects architectural smells. This poject is detecting 5 types of architectural smells as follows;

Cyclic Dependency: This smell arises when two or more architecture components depend on each other directly or indirectly. In other words, if a component involved in a dependency cycle then it can be hard to release, maintain or reuse the software.

Unstable Dependency: This smell describes as when a component depends on other components that are less stable than itself.

God Component: This smell occurs when a component is excessively large either in the terms of LOC or number of classes.

Feature Concentration: This smell occurs when a component realizes more than one architectural concern/feature.

Dense Structure: This smell arises when components have excessive and dense dependencies without any particular structure.

4. Results:

In this section, I am going to show the project findings and analysis on the android applications with respect to the code metrics, code smells and architectural smells as following;

4.1 Code Metrics:

The Figure 1.1 refers to LOC the number of lines of code with the history of commits.

We could say that the with the increase of the commits the LOC slightly increases. In the most cases, the max number of LOC reach in the middle phase during maintaining of the project.

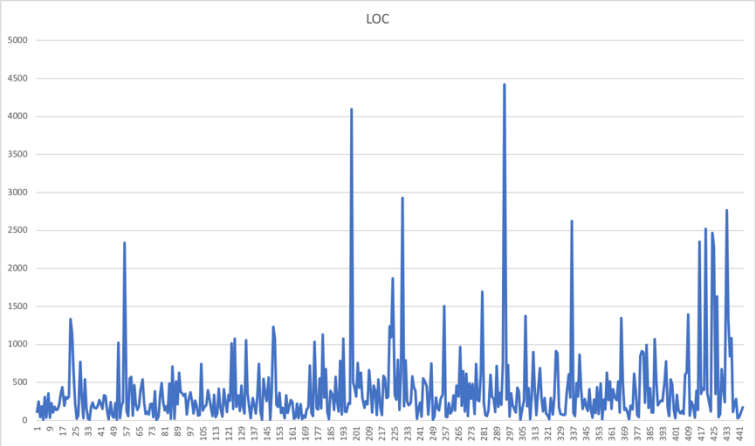

Figure 1.1

The Figure 1.2 shows the line chart of the Lack of Cohesion in Methods (LCOM) with the commits history.

It is observed ups and downs in the LCOM with the commits increases but has high values during the period of middle and in the last of maintenance.

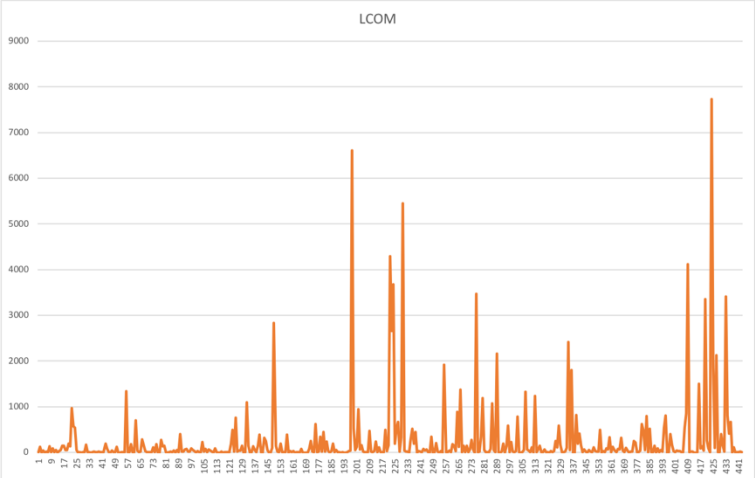

Figure 1.2

The Figure 1.3 displays the data of Coupling between objects (CBO) according to the history of commits.

It is observed that our apps found CBO in the classes and the development history shows with the increase of commits CBO also increases and we may say that the value of CBO remain constant during the maintaing android apps.

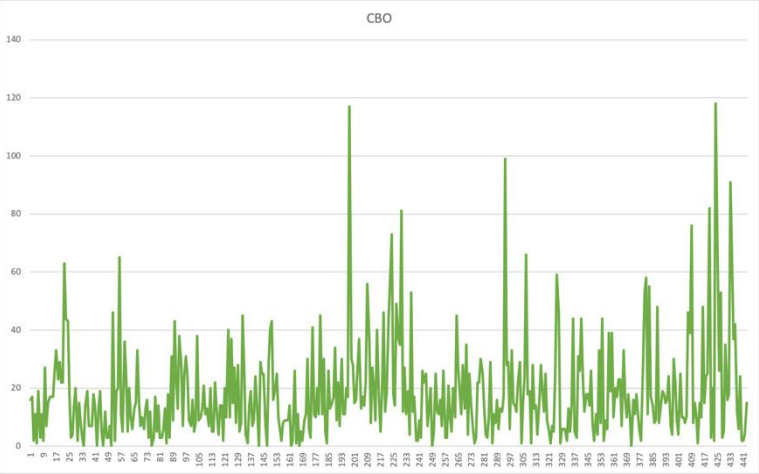

Figure 1.3

4.2 Code Smells:

The Figure 1.4 is the horizontal bar chart of code smells which shows that the apps, which were in the detection, have no Functional decomposition smell. On the other hand, 2.09%, 0.5%, 2.59% and 1.72% for Class Data Should be Private, Complex Class, God Class and Spaghetti Code respectively.

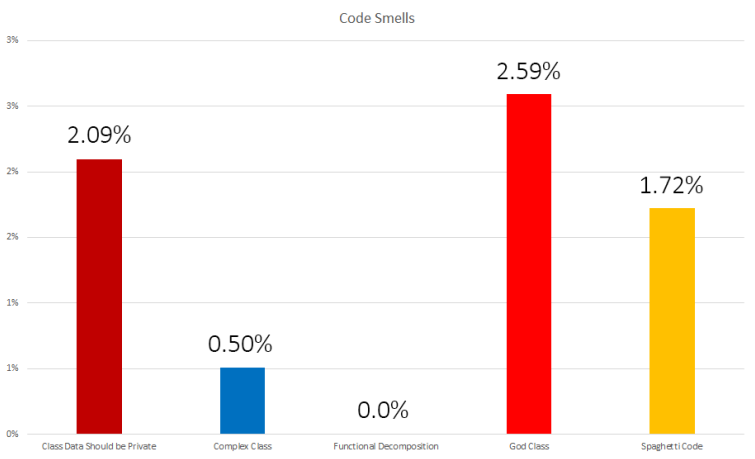

Figure 1.4

4.3 Architectural Smells:

The Figure 1.5 is the horizontal bar chart of architectural smells which shows that the apps detected architecture smells which are 4% of God Component, 60% of Cyclic Dependency, 15% of Unstable Dependency, 19% of Feature Concentration and 1% of Dense Structure.

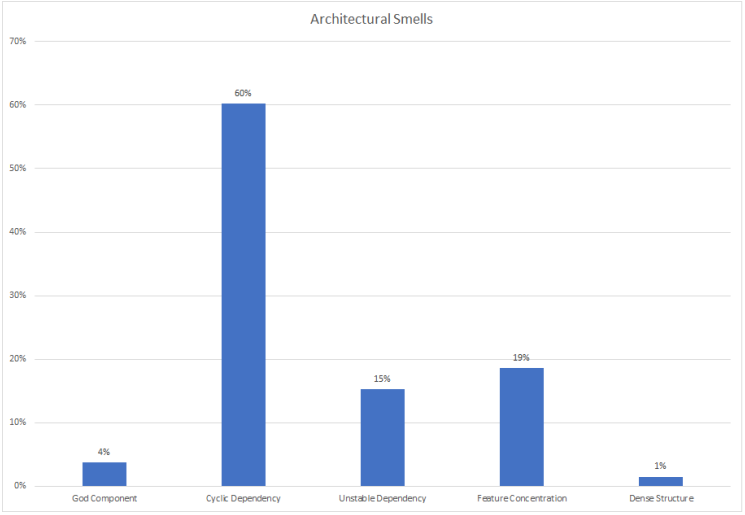

Figure 1.5

5. Conclusion:

In this project, there are 31 android applications for the analysis of code metrics, according to the development history, detection of code smells and architectural smells by using Swanlytics and DesigniteJava. By aggregation of all the commits history of the android applications, the classes found lack of cohesion methods(LCOM) and coupling based methods(CBO), but the history says that LCOM and CBO may decrease with the time. Code smells and architecture smells also detected in our app's classes in which God class (code-smell) and Cyclic dependency (architecture-smell) has highest percentages i.e. 2.59% and 60% respectively. The detection of smells is important to improve the quality of mobile applications to reduce the overall cost of their development and maintenance, to avoid or overcome the technical debt.