International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:7, 2014

On Measuring the Reusability Proneness of Mobile Applications

Fathi Taibi

Abstract**—**The abnormal increase in the number of applications available for download in Android markets is a good indication that they are being reused. However, little is known about their real reusability potential. A considerable amount of these applications is reported as having a poor quality or being malicious. Hence, in this paper, an approach to measure the reusability potential of classes in Android applications is proposed. The approach is not meant specifically for this particular type of applications. Rather, it is intended for Object-Oriented (OO) software systems in general and aims also to provide means to discard the classes of low quality and defect prone applications from being reused directly through inheritance and instantiation. An empirical investigation is conducted to measure and rank the reusability potential of the classes of randomly selected Android applications. The results obtained are thoroughly analyzed in order to understand the extent of this potential and the factors influencing it.

Keywords**—**Reusability, Software Quality Factors, Software Metrics, Empirical Investigation, Object-Oriented Software, Android Applications.

I. INTRODUCTION

MOBILE applications, which are called mobile apps or just apps, are application software designed to run on smartphones, tablet computers and other mobile devices [27]. These applications have become very popular today and the demand for them keeps increasing. Initially, these 'apps' provided solutions to basic mobile usages such as email and calendar. However, the scope they cover today is considerably larger and includes areas such as location-based services, banking, mobile-commerce, games and medical. The amount of mobile Android applications [4] available today is astonishing. There are more than 1 million Android applications available for download in Google play [4]. According to [6], 21% of these applications are of a low quality. Google removes applications from the market regularly if found to be of a poor quality. However, while these low quality applications are in the market, users are able to download and use them. Some developers will reuse them to build new applications, which make the impact even greater. Moreover, these applications are intended for various mobile platforms, more than 82% of them are free and more than 51% of the later are Android applications. Furthermore, Android applications represent more than 51% of the downloaded mobile applications [6].

Dr. Taibi is with UNITAR International University, Petaling Jaya, 47301 Selangor, Malaysia (phone: +603-76277200; fax: +603-76277447; e-mail: taibi@unitar.my).

The availability of this huge amount of Android applications is harmful from two different perspectives. Firstly, people are using low quality applications, which are just buggy in the normal cases and malicious in the worst cases. Secondly, developers are reusing these low quality applications to develop new buggy and malicious applications. This process can be quite damaging since it repeats itself. Moreover, the increasing number of these applications in the very short period of time span they have been around indicates some troubling facts. Firstly, most of these applications are probably the result of excessive reuse. Secondly, this excessive reuse is mostly cloning of popular paid applications [12]. Finally, a considerable portion of these applications can be easily categorized as malware or containing malicious code [39].

Mobile applications are often associated with malicious code. Malicious authors can attach malicious code to legitimate applications, which leads to the creation of applications labelled as "piggybacked" [39]. These applications are then advertised in the available application markets in order to infect unsuspecting users. One example of the malicious actions performed as a result of that is converting the infected phones into bots [15]. Moreover, mobile applications are inherently complex since they rely on third party libraries or Application Program Interfaces (APIs) [30]. These APIs change very frequently. Hence, a considerable percentage of API references in mobile applications are outdated.

Several qualities are desired in software systems in general and the OO ones in particular. Functionality, efficiency, maintainability, reliability and reusability are examples of some of these desired quality factors. They are measured using software metrics that are applicable at various levels of development. Some metrics are applicable only when the project is completed (i.e. to the source code) whereas some other metrics are applicable at earlier stages such as at the design [9]. In addition to a sound theoretical foundation, the metrics used in assessing software quality factors must be empirically validated by showing a clear and strong correlation between them and the qualities measured. Several approaches and models were proposed to measure individual software quality factors such as maintainability in [23], reusability of components in [38], usability of components in [7], reliability of component-based software architectures in [33], and the stability of Java classes in [17].

A class is a fundamental concept in the development of OO software systems. It would be very useful to measure quality factors at its level. This should be beneficial both during

software development and after its release. During development, getting measurements representing the degree of compliance of a class in regards to certain desired qualities can help in deciding the refactorings that should be applied in order to improve this compliance. Additionally, this can help in planning and executing the necessary modifications when dealing with various types of maintenance requests. Furthermore, this can be extremely useful for deciding the classes to be reused (or modified to be reused) in the developed of new software systems. However, measuring quality factors at the class level has not received a lot of attention from researchers, which is shown by a lack in published work addressing this problem. This is especially true for assessing the reusability proneness of classes in OO software systems.

Software reuse has been practiced since the early days of programming. It saves cost, increases the speed of development and improves reliability [20]. Reused components are often more stable (i.e. they are modified less than newly developed ones) and have lower defect density [31]. Several quality factors have been associated with the reusability proneness of software modules. Modularity, low complexity, high cohesion and low coupling are examples of highly reliable factors of a module's reusability potential [37]. Several studies have established a clear relationship between these factors and software defects such as [35] and [40]. Android applications are developed in the Java programming language using the Android Software Development Kit (SDK). When studying these applications, it is important to understand that a software module in this context is a file containing one or several Java classes. Hence, the reusability of an application dependents on the reusability potential of its individual classes.

An approach is proposed in this paper to measure the reusability potential of the classes of Android applications. This potential represents the probability that a class will be reused successfully through inheritance and instantiation. The approach is an extension of the one proposed in [37] in order to improve the measurement of the understandability and low complexity factors. The proposed approach allows eliminating applications with poor quality from being reused through inheritance and instantiation since the factors used in measuring the reusability potential are also associated with several other qualities that are desired in Android applications and OO software systems in general. Moreover, the proposed reusability metric should allow ranking the classes of a particular software according to their reusability proneness as well as ranking several software systems according to the aggregated reusability proneness of their classes. This should provide a good support for reuse in software development. Furthermore, an empirical investigation is conducted to measure the reusability of randomly selected Android applications. The results are thoroughly analyzed in order to discover the real reusability potential of Android applications and the factors influencing it.

II. Related Work

There is no doubt that Android applications are being reused. This is due to the abnormal increase in the number of applications available in the markets in a very short period of time span they have been around. These applications are largely un-reviewed because of too many submissions. However, Google removes low quality applications on a regular basis [6]. This is insufficient because while these poor quality applications are available for download, they are used and reused to develop new applications. Full reuse of their classes through inheritance and instantiation is more harmful in comparison with partial reuse such as when calling their static methods. Software reuse in Android applications was analyzed in [34] from two perspectives: reuse by inheritance and by reusing the classes (i.e. instantiation). Thousands of mobile applications were analyzed. The results showed that 23% of the studied classes are derived from a base class in the Android API. Moreover, 27% of the studied classes were found to be derived from a domain base class and 61% of the classes occurred in two or more categories.

An empirical study about software reuse in Java open-source projects was conducted in [22]. The study aims to find out whether open source projects use third party code and to study the extent of code reuse occurrence. Black-box software reuse was found to be the predominant form of software reuse. Additionally, all the 20 studied projects had more than 40% of software reuse and in 19 of these projects the amount of reused code exceeded the amount of the original one. Moreover, a quality model targeting the maintainability and reusability of software was presented in [28]. The model is tool supported and depends on user intuition in selecting a metric set for their projects. The reusability quality factor was measured based on modularity and complexity. The former is measured based on the cohesion and coupling of classes while the latter is measured based on the internal and external complexity of classes.

The ability of 29 internal class measures to estimate reuse proneness from the perspectives of inheritance and instantiation was studied empirically in [2]. These measures represent class attributes such as cohesion, coupling and size. Two interesting findings were derived from this study. Firstly, size and coupling attributes are correlated to its reuse proneness via inheritance and instantiation. Secondly, the cohesion attribute has a positive impact on its reuse proneness via instantiation only. However, the model lacks effectiveness due to the large number of attributes used and the overlapping in the qualities they measure.

A metric suite to measure the reusability of components in component-based software development was proposed in [38]. Five metrics were defined and used to measure understandability, adaptability and portability factors of a given component. A confidence interval for each metric was set trough statistical analysis of a number of JavaBeans components. Understandability was measured based on the existence of meta-information and the observability of a component. Adaptability and portability were measured based on metrics measuring customizability and external

dependency, respectively. Moreover, new coupling and cohesion metrics to rank the reusability of Java components was proposed in [18]. Cohesion was measured as the degree of cohesion between the methods of a class including transitive cohesion. A similar intuition was used for the proposed coupling metric. The experiments conducted revealed that the proposed metrics were better predictors of the number of lines of code that were added, modified or deleted in order to extend the functionality of the studied components in comparison to some of the existing cohesion and coupling metrics. However, these two metrics alone cannot form a good reusability predictor since they don't measure other important factors such as complexity, understandability and customizability.

265359 mobile applications were analyzed in [11] and 4295 of them were discovered to be victims of cloning. Each one of these applications was probably cloned several times. Additionally, 36106 applications were rebranded including 88 malware and 169 malicious applications. Moreover, a scalable infrastructure for code similarity analysis in Android applications was proposed in [21]. The developed system was evaluated using 58000 applications. 463 applications were found to contain buggy code reused from sample code provided by Google. 34 applications were found to be instances of malware and their variants while 3 applications were found to be copies of a popular paid game. Furthermore, an approach to detect 'piggybacked' applications in Android markets was presented in [39]. The approach is based on the idea that the attached malicious code in not an essential part of a given application primary functionality. A prototype was developed as an implementation of the proposed approach and was used to analyze 84767 applications. The results obtained showed that the rate of these malicious applications ranges from 0.97% to 2.7% while it is around 1% in the official Android market.

An exploratory study on micro-applications for Android and BlackBerry platforms was conducted in [36] in order to understand their development and maintenance processes. This study led to two major discoveries. Firstly, Android micro applications rely primarily on android APIs whereas BlackBerry micro applications rely on Java libraries. Secondly, source files in Android change more frequently; however, they are subject to smaller changes in comparison to BlackBerry source files. Moreover, [26] analyzed the relationship between the fault and change proneness of APIs used by mobile applications and their lack of success, which was estimated based on their user ratings. The applications having higher user ratings were found to exhibit a lower number of bug fixes in the used APIs. Additionally, applications with higher user ratings use more stable APIs compared to those with lower user ratings. Furthermore, a case study of the co-evolution behavior of the Android API and its dependent applications was presented in [29]. The results obtained showed that 28% of API references in the applications are outdated and 22% of these outdated API usages get upgraded eventually to newer API versions. However, this happens at an interval that is much slower than the average API release interval, which is about 3 months.

A tool-supported approach to comprehend mobile applications was presented in [30]. This led to several interesting discoveries. Firstly, the use of inheritance is almost absent in the analyzed applications. The average number of derived classes metric in the studied projects was 0.19. This shows that many applications are not developed in a systematic way. Secondly, some applications contained the entire source code of third party libraries. Instead, Java Archive files (JAR) files should be used and imported into these applications. Finally, development guidelines are often ignored. For example, having too many main activities. The latter leads to diverse entry points to the applications, which makes their comprehension and maintenance difficult.

Software reusability proneness is associated with several factors. One of these factors is readability or understandability of the source code. Using naming conventions and writing useful comments are examples of techniques that can improve understandability. The usage of naming conventions has been found to be reliable if the names used are related to the concepts implemented [5]. Lexicon bad smells such as inconsistent term usage and odd grammatical structures can make carrying maintenance tasks difficult [1]. Moreover, low complexity is desired. Highly complex programs are less reusable, hard to test and maintain. Furthermore, structuring program code using modules that are highly cohesive [3] and highly independent [13] is a crucial factor for reusability. Excessive coupling between classes was found to be a very reliable predictor of faults in OO systems as indicated in [19] where it was found that Coupling Between Objects (CBO) is more reliable than Lack of Cohesion of Methods (LCOM) and several other OO design metrics in predicting faults. Hence, this metric together with similar other metrics can form the basis of a reusability assessment approach since they allow measuring the factors related to the reusability proneness of program code while at the same time discarding defect-prone code from being reused. Finally, classes participating in anti-patterns (i.e. bad smells, which are poorly designed classes) have been found to be more change and fault prone than those that do not [25].

III. The Proposed Approach

The proposed approach is intended for OO software systems in general and not just for Android applications. It uses a set of well-established software metrics to measure the different factors of reusability assessment. These factors are qualities with proven positive effect on the reusability proneness of a software module. Given a completed OO software project P comprising n classes (C_i. i=1 to n) and m source code files (S_j. j=1 to m), reusability assessment consists of measuring the proneness of each one these classes to be reused successfully. The proposed metric is based on measuring three distinct factors using values between 0 and 1 that indicate the degree of compliance of a class in their regard. Firstly, Understandability (U) is measured using the relevance of names used for the classes, fields and methods

(Relevance of Identifiers - ROI) together with their correlation with code comments (Correlation Identifiers Comments - CIC). This factor has been extended to include the Rate of Code Comments (RCC) as well, which is a value between 0 and 1 representing the ratio of the number of comment lines by the total number of lines of code excluding blank lines. Secondly, Modularity (M) is measured based one the values of LCOM and CBO. Low coupling and high cohesion are used a basis to measure M where LCOM allows measuring structural cohesion and CBO measures coupling. CBO is given more weight than LCOM in the calculation due to its significance as a defect predictor. The usage of CIC in the calculation of the metric U allows incorporating conceptual cohesion in reusability assessment. However, CIC was not used in the calculation of the metric M in order to avoid penalizing highly cohesive and lowly coupled classes that are poorly commented.

The Low Complexity (LC) factor has been extended to include Response For a Class (RFC) and improve the way Cyclomatic Complexity (CC) is used in the calculation. This involves calculating the sum of the weights of the individual methods of the class in regards to their CC and dividing it by the number of methods (Weighted Cyclomatic Complexity - WCC). This way the result is more precise than calculating the average CC of the methods of the class since this average could be acceptable while the class has a considerable number of complex methods. Using RFC allows measuring the complexity of the class in terms of method calls, which could affect its testability and maintainability as well. Two other metrics are used to measure LC. These are the Number of Methods (NM) in the class and its Depth of Inheritance Tree (DIT). This allows measuring the internal complexity (WCC), the structural complexity (DIT) and the amount of responsibility in the class (NM). WCC, DIT and RFC are given more weight than NM due to their significance in measuring complexity. Finally, it is important to note that being defect-free is also an important factor to consider in assessing the reusability proneness of a class. This factor is already included in the factors LC and M since most of the metrics that are used to measure complexity, coupling and cohesion are proven defect predictors in OO software systems. Similarly, the size factor is already included in the factor LC since the metrics used to assess low complexity are highly correlated to the size of the class.

Several other factors associated with reusability were considered. The most relevant among them was the customizability of a class. It indicates the degree of which a class' interface can be customized and its fields configured. One way to measure customizability is to compute the ratio between the numbers of fields with associated 'setter' methods by the total number of fields. There are two main reasons why currently this factor was not used in reusability assessment. Firstly, the modularity factor includes LCOM which is in a way related to customizability. This is because when a class is cohesive, its methods overlap in the way they access its fields, which could very likely indicate the presence of 'setter' methods. Secondly, data classes (i.e. classes with fields, 'setter' and 'getter' methods and nothing else) have a low cohesion (i.e. poor LCOM) and are a bad smell. However, this type of classes has a high customizability. Furthermore, the stability of a class is an important factor to consider in assessing its reusability proneness. Stability in this context refers to the ease of which a class can evolve while preserving its design. Such evolution is usually driven by error detection, changes in the environment or to the requirements. This factor was discarded due to the absence of reliable metrics which can measure it using information from the source code only.

Table I gives a summary of the factors used or considered in assessing the reusability proneness of a class together with how they are measured and the rational for including or discarding them. Reusability assessment is performed according to the diagram shown in Fig. 1.

TABLE I
REUSABILITY ASSESSMENT FACTORS CONSIDERED AND THEIR ASSOCIATED METRICS

Factor	Reason for inclusion or exclusion	Metrics used	How is it measured?
Understandability (U)	Without it, a class cannot be modified, extended or even used through instantiation successfully.	ROI, CIC and RCC	Weighted average of values derived from the metrics used.
Modularity (M)	To be reused successfully, a class must be highly cohesive and highly independent in order to avoid any undesirable effect.	CBO and LCOM	Weighted average of values derived from the metrics used.
Low Complexity (LC)	Without it, a class cannot be modified, extended, tested or maintained successfully.	WCC, DIT, RFC and NM	Weighted average of values derived from the metrics used.
Defect-free (DF)	Without it, a reused class introduces new defects which are hard to detect and resolve. The metrics used to assess the factors M and LC measure this factor as well since they are proven defect-predictors.	Same as M and LC	Covered by the measured M and LC
Size (SZ)	A class cannot be modified, extended or used successfully if its size is not manageable. The metrics used to assess the factor LC measure this factor as well since they are highly correlated to the size of a class.	Same as LC	Covered by the measured LC
Customizability (CUS)	It was discarded because some bad smells have a high customizability. Also, it overlaps slightly with modularity.	-	-
Stability (STA)	It was discarded due to the absence of reliable metrics which can measure it using information from the source code only.	-	-

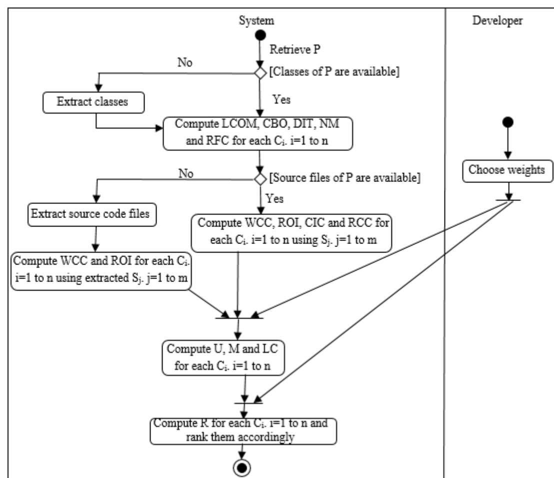

Fig. 1 Reusability Assessment Approach

The metric R is calculated as a weighted average of the factors U, M and LC. A heuristic method was used to find their weights using a set of classes with known reuse potential. These weights could also be chosen according to the qualities required by a developer in search of reusable modules. LC and M are given more weights than U since the latter factor was found to be slightly less significant than the former two factors in measuring the reusability proneness of a class. Currently, the weight 0.35 is used for M and LC and 0.3 for U. U is calculated as the weighted average of ROI, CIC and RCC (when applicable). CIC is calculated using a similarity metric based on N-Grams [32]. ROI is given more weight than CIC and RCC (1.5 versus 1). This is justified by observations made on a large number of classes that are highly reusable where the names chosen for their attributes, methods and classes are very expressive. However, their rate of code comments is very low, which translates into poor values for both RCC and CIC. The extraction of the names used for the classes, methods and attributes as well as code comment is automatic. However, assessing their relevance is currently done manually. M and LC are calculated as weighted average of the metrics used. CBO, WCC, DIT and RFC were given more weight than LCOM and NM (1.5 versus 1) because of their higher significance in measuring M and LC respectively. The values of the metrics used in the calculation are shown below.

TABLE II
VALUES OF THE METRICS USED TO CALCULATE THE FACTORS U, M AND LC

	1	**0.75**	**0.5**	**0.25**	**0**
Condition	0.2≤RCC	0.15≤RCC<0.2	0.1≤RCC<0.15	0.05≤RCC<0.1	RCC<0.05
	LCOM=0	0<LCOM<3	3≤LCOM<5	5≤LCOM≤10	LCOM>10
	CBO≤5	5<CBO≤7	7<CBO≤9	9<CBO≤10	CBO>10
	CC≤10	10<CC≤20	20<CC≤35	35<CC≤50	CC>50
	NM≤7	7<NM≤10	10<NM≤13	13<NM≤16	NM>16
	DIT≤5	5<DIT≤7	7<DIT≤9	9<DIT≤10	DIT>10
	RFC<20	20≤RFC<30	30≤RFC<40	40≤RFC≤50	RFC>50

The values chosen are based on ranges defined according to know 'safe' values. For example, many tools used 5 as a threshold for CBO. Hence, this value was used to create a range that allows attributing values between 0 and 1 to classes in regards to low coupling. Similarly, 10 is a known good threshold for the CC of a method and was used to attribute weights between 0 and 1 to the methods of a class in order to compute WCC. The same process was repeated for all the other metrics.

IV. EMPIRICAL INVESTIGATION

In order to investigate the reusability potential of Android applications, 25 applications were randomly selected from various Android markets such as [16]. They represent various types of applications such as Brain and Puzzle, Business, Communication, Education, Game, etc. They incorporated a total of 561 files comprising 1339 classes with a total of 99826 Lines Of Code (LOC). The following table shows the details of the selected applications:

TABLE III
DETAILS OF THE SELECTED APPLICATIONS

	Max	**Min**	**Median**	**Mean**	**Std**
#Files	86	2	19	22.44	17.29
#Classes	128	12	44	53.56	31.89
Size (LOC)	13676	182	2758	3993.04	3306.7
%Comments	30.53%	0.55%	3.06%	6.73%	8.2%

Some of the selected applications had Android application package files only (i.e. files with 'apk' extension). Class files (i.e. files with 'class' extension) were needed in order to calculate the metrics required to measure the individual factors. However, only a 'dex' file (Dalvik Executable) is available from an 'apk' file. A converter [14] was used in order to retrieve the individual class files. It translates a 'dex' file into a 'jar' file that contains the individual classes of an application. Additionally, a Java decompiler [24] was used to obtain the source code. Hence, for these applications, RCC and CIC were not used in the calculation of the factor U since code comments cannot be decompiled as shown in Fig. 1.

Chidamber and Kemerer Java Metrics [10] and C and C++ Code Counter [8] tools were used to calculate CC, LCOM, CBO, NM, DIT and RFC. A small prototype tool was developed to calculate RCC and CIC while ROI was assessed

manually. The results were then thoroughly analyzed. Fig. 2 shows the reusability of each class in the studied applications where the results are sorted for a better analysis.

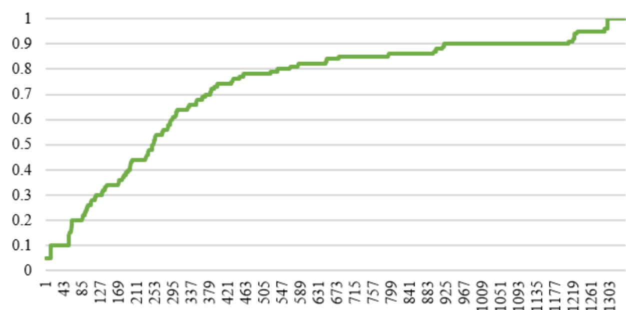

Fig. 2 Reusability of the studied classes

The reusability results obtained for the studied classes were between 0.05 and 1. The average R obtained was 0.74 and only 369 out of 1339 studied classes had scored below 0.7 (i.e. 27.55%). These figures show an acceptable performance. However, in order to have a better understanding of why the results obtained for some classes were poor, these classes were clustered into two categories. Category 1 (Cat1) included the classes with a very poor reusability (i.e. below 0.5) and Category 2 (Cat2) included those with a reusability greater or equal to 0.5 and below 0.7. There were three reasons behind this analysis. The first reason was to study the distribution of these classes across the selected applications and the impact of the application's size on the rate of classes with poor reusability. Secondly, since the calculation of R is driven by the factors LC and M, it was important to measure the impact of these two factors on the calculated R metric of the classes in these two categories. Finally, since nesting (i.e. having inner and anonymous classes) and having a large interface (i.e. having too many public methods) may affect reusability, the third reason was to measure the impact of nesting and having a large interface on the classes of these two categories. Fig. 3 shows the distribution of the classes in Cat1 and Cat2 across the selected applications together with their total number of classes (#classes).

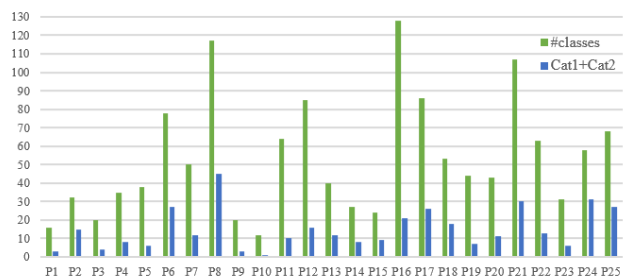

Fig. 3 Distribution of classes with poor R and the total number of classes in each application

The results obtained showed no clear correlation between the size of an application and the rate of classes with poor reusability in it. Some large applications had small number of classes in Cat1 and Cat2. For example, P16 has 128 classes and only 21 of them are in these categories. In contrast, some small applications had a large number of classes in these categories. For example, P24 has 58 classes and 31 of them are in these categories. The percentage of classes in Cat1 and Cat2 in the selected applications was between 8.33% (P10) and 53.45% (P24) with an average of 27.56%. In 9 out of the 25 selected applications, the rate of these underperforming classes was more than 30%.

It was also important to measure the impact of M and LC on the classes in Cat1 and Cat2 as explained earlier. In order to perform this analysis, the number of classes with M and LC below 0.5 was analyzed among the classes of Cat1. Similarly, the number of classes with M and LC greater of equal to 0.5 and below 0.7 was analyzed among the classes of Cat2. Fig. 4 shows the results obtained.

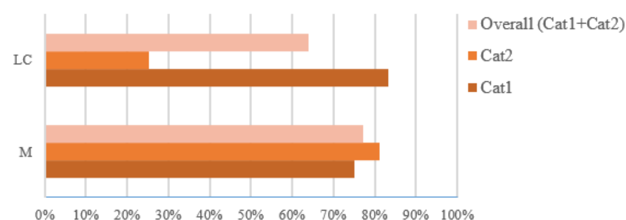

Fig. 4 Impact of M and LC on classes with poor R

The factor LC has more impact on the classes of Cat1 than the factor M (83.33% versus 75.2%) while the latter has much more impact on the classes of Cat2 (81.3% versus 25.2%). The factor M has more overall impact on the classes of Cat1 and Cat2 in comparison with LC (77.24% versus 63.96%). This means that most of these underperforming classes had primarily cohesion and coupling issues and secondly complexity issues according to the proposed R metric. Moreover, the impact of nesting on the classes in Cat1 and Cat2 was investigated together with the impact of having a large interface (i.e. classes with more than seven public methods). Even though NM is used in the calculation of the factor LC; however, as indicated earlier, it is given lower weight in comparison with WCC, DIT and RFC. Hence, it is justifiable to study the correlation between classes with poor reusability and having a large interface. Fig. 5 shows the results obtained.

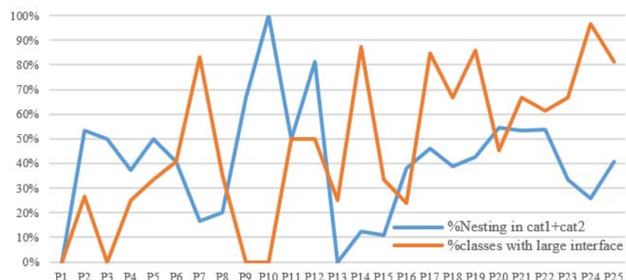

Fig. 5 Impact of nesting and large interfaces on classes with poor R

The results obtained showed that the rate of classes having nested classes among those of Cat1 and Cat2 was between 0% and 100% with an average of 38.21%. This rate was above 50% in 10 applications, which gives a significant indication that poor reusability is correlated to having nested classes. Moreover, the percentage of classes with a large interface in these categories was between 0% and 96.77% with an average of 55.56%. This rate was above 50% in 12 applications, which indicates also a more significant correlation between poor reusability proneness of a class and having a large number of public methods. This correlation could likely indicate that having a large number of public methods may be indirectly affecting the factor M, i.e. maintaining high cohesion and low coupling is difficult to achieve in a class with a large interface. The latter could be associated with poor design or improper use of inheritance because the interface of a derived class is the combination of the inherited public methods and those declared locally. However, further studies are needed to confirm this link.

In order to analyze the relationship between the average reusability of the classes of a particular application and its type (i.e. Business, Tool, etc.), the average reusability obtained for each type of applications was measured. Fig. 6 shows the results obtained.

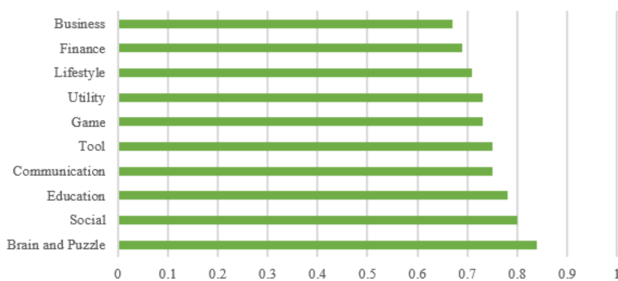

Fig. 6 Average R in regards to the type of application

Interestingly, the business and finance applications were the only type of applications with an aggregated reusability below 0.7. This may be associated with a higher complexity in their classes in comparison with other types of applications. Bain and puzzle, Social and Education were the ones with the best aggregated reusability (in this order). These results do not indicate a definite pattern or trend, rather, they provide a different view of the results obtained. A final analysis was needed to study the correlation between the rating of applications and their average reusability. A large number of applications had a small number of online votes and were not included in this analysis. The latter includes only seven applications with more than 200 online votes. In order to perform this analysis, the relative number of votes (Relative # Votes) was calculated by dividing the number of votes by the maximum number of votes for an application. Similarly, the online ratings were divided by five to make their range between zero and one (Rating). Fig. 7 shows the results obtained.

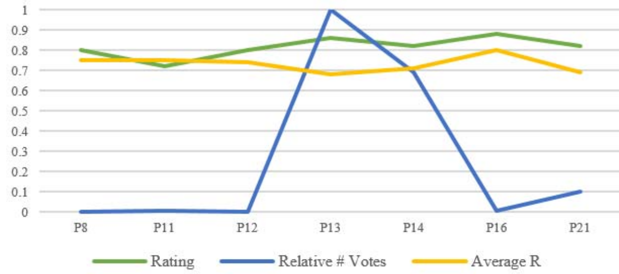

Fig. 7 Average R vs. online ratings of applications

The results obtained showed that the ratings of applications slightly exceeded their measured reusability potential except in one application (P11). The application that had the highest number of votes (P13) had the largest gap between its rating and reusability. This gap was quite small in four out of the seven applications, which could indicate that even though online rating of an application is not driven by a complete analysis but rather by assessing its usability and functionality, it still correlates quite well with its reusability. However, this does not necessarily mean that a highly rated application is highly reusable by consequence. Further studies are needed in order to analyze this correlation.

Three aspects of validity were considered. Firstly, internal validity is shown through a clear correlation between the factors used and the reusability proneness of a given class. Additionally, these factors were measured using well-established and validated metrics. Secondly, manual intervention was minimized in order to avoid errors in measurements. Also, the results obtained automatically were cross checked twice in order to find any abnormal values. This is a sign of construct validity. Finally, even though the number of studied classes in the empirical investigation was not extremely large. Various types of applications were used and were randomly selected from various Android markets. This is sign of external validity and shows that the results obtained can be replicated to a larger number of classes from other applications.

V. Conclusion and Future Work

An approach was proposed in this paper to measure the reusability proneness of the classes of an OO software system and was used to assess the reusability of randomly selected Android applications. The approach is shaped around a reusability assessment metric that measures the probability that a given class is reused successfully through inheritance and instantiation. Three factors were used in this assessment. They comprise understandability, modularity and low complexity. They were measured using some well-established OO software metrics together with newly proposed ones. These metrics allow covering two other factors namely defect-free and size. The results obtained showed that on average the classes of these applications have an acceptable reusability potential. However, more than a fourth of the studied classes had a poor reusability potential. The modularity factor had more impact on these underperforming classes in comparison

with the low complexity factor. Moreover, having a large interface was found to be highly correlated with these classes together with having nested classes. Hence, controlling the size of the interface and amount of nesting in classes should help improving their reusability proneness together with the other desired qualities represented by the factors used in reusability assessment. Furthermore, the online ratings of the studied applications were quite consistent with their average reusability obtained even if user ratings are based on their functionality and usability rather than on a thorough analysis of the qualities that were used in measuring the proposed reusability metric.

The proposed reusability approach could be extended to include more metrics and factors which are associated with the desired qualities in reusability proneness assessment. This is the case of the customizability factor. There is a need to find a way to reconcile this factor with modularity in respect to the cohesion of classes. However, special attention should be made to the overlapping that may exist between the qualities measured by the metrics used in order to achieve efficiency. Moreover, automating the calculation of the metric ROI could be made possible in the presence of a data dictionary derived from software requirements specification. Similarly, class stability metrics can be proposed in the presence of detailed requirements specifications and traceability information linking these requirements to classes. Furthermore, conducting more empirical studies involving a larger number of applications having more classes is necessary to confirm the findings made so far and potentially discover new ones. This could make it possible to analyze the real causes of poor reusability in classes and use it a platform to provide clear guidelines to improve the reusability proneness of OO software systems in general and Android applications in particular.

REFERENCES

[1] Abebe, S. L., Kessler, F. B., Haiduc, S., Tonella, P. and Marcus, A. “The Effect of Lexicon Bad Smells on Concept Location in Source Code,” *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 125 – 134, 2011.

[2] Al-Dallal, J. and Morasca, S. “Predicting object-oriented class reuse-proneness using internal quality attributes,” *Empirical Software Engineering*, vol. 19, no. 4, pp. 775-821, 2014.

[3] Al-Dallal, J. and Briand, L. C. “A Precise method-method interaction-based cohesion metric for object-oriented classes,” *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, 8:1-8:34, 2012.

[4] Android, the world's most popular mobile platform, http://developer.android.com/about/index.html, April 2014.

[5] Anquetil, N. and Lethbdige, T. “Assessing the Relevance of Identifier Names in Legacy System,” *In Proc of the Centre for Advanced Studies on Collaborative Research Conference*, 1998.

[6] Appbrain, http://www.appbrain.com/stats/number-of-android-apps, April 2014

[7] Bertoa, M. F., Troya, J. M. andVallecillo, A. “Measuring the usability of software components,” *The Journal of Systems and Software*, vol. 79, pp.427–439, 2006.

[8] CCCC, http://cccc.sourceforge.net/, April 2014.

[9] Chidamber, S. and Kemerer, C. “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[10] CKJM, http://www.spinellis.gr/sw/ckjm/, April 2014.

[11] Crussell, J., Gibler, C. and Chen, H. “AnDarwin: Scalable Detection of Semantically Similar Android Applications,” *Lecture Notes in Computer Science*, pp. 182-199, 2013.

[12] Crussell, J., Gibler, C. and Chen, H. “Attack of the Clones: Detecting Cloned Applications on Android Markets,” *Proceedings of the 17th European Symposium on Research in Computer Security*, *Lecture Notes in Computer Science*, vol. 7459, pp. 37-54, 2012.

[13] Darcy, D. and Kemerer, C. “OO Metrics in Practice,” *IEEE Software*, vol. 22, no. 6, pp. 17-19, 2005.

[14] Dex2jar, http://code.google.com/p/dex2jar/, April 2014.

[15] Enck, W., Ongtang, M. and McDaniel,P. “On Lightweight Mobile Phone Application Certification,” *Proceeding of the 16th ACM conference on Computer and communications security*, pp. 235-245, 2009.

[16] Google Play, https://play.google.com/store/apps, April 2014.

[17] Grosser, D., Sahraoui, H. A. and Valtchev, O. “An analogy-based approach for predicting design stability of Java classes,” *Proceedings of the 9th International Software Metrics Symposium*, pp. 252-262, 2003.

[18] Gui, G. and Scott, P. D. “Coupling and cohesion measures for evaluation of component reusability,” *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 18 – 21, 2006.

[19] Gyimothy, T., Ferenc, R. and Siket, I. “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, 2005.

[20] Haefliger, S., Von-Krogh, G. and Spaeth, S. “Code Reuse in Open Source Software,” *Management Science*, vol. 54, no. 1, pp. 180-193, 2008.

[21] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C. and Song, D. “Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications Detection of Intrusions and Malware, and Vulnerability Assessment,” *Lecture Notes in Computer Science*. Vol. 7591, pp. 62-81, 2013.

[22] Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B. and Irlbeck, M. “On the extent and nature of software reuse in open source Java projects,” *Proceedings of the 12th international conference on Top productivity through software reuse, Klaus Schmid (Ed.). Springer-Verlag*, pp. 207-222, 2011.

[23] Heitlager, I., Kuipers, T. and Visser, J. “A Practical Model for Measuring Maintainability,” *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pp. 30–39, 2007.

[24] JAD, http://varaneckas.com/jad/, April 2014.

[25] Khomh, F., Di-Penta, M., Gueheneuc, Y.G. and Antoniol, G. “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243-275, 2012.

[26] Linares-Vasquez, M., Bavota, G., Bernal-Cardenas, C., Penta, M. D., Oliveto, R. and Poshyvanyk,D. “Api change and fault proneness: A threat to the success of android apps,” *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 477-487, 2013.

[27] Lee, V., Schneider, H. and Schell, R. *Mobile Applications: Architecture, Design, and Development*, 1st edition, Prentice Hall, 2004.

[28] Lee, Y. and Chang, K. H. “Reusability and maintainability metrics for object-oriented software,”*Proceedings of the ACM-SE 38th annual on Southeast regional conference*, pp.88-94, 2000.

[29] McDonnell, T., Ray, B.and Kim, M. “An Empirical Study of API Stability and Adoption in the Android Ecosystem,” *Proceeding of the 29th IEEE International Conference on Software Maintenance*, pp. 70-79, 2013.

[30] Minelli, R. and Lanza, M. “Software Analytics for Mobile Applications – Insights & Lessons Learned,” *Proceeding of the European Conference on Software Maintenance and Reengineering*, pp. 144-153, 2013.

[31] Mohagheghi, P., Conradi, R., Killi, O.M. and Schwarz, H. “An empirical study of software reuse vs. defect-density and stability,” *Proceedings of the 26th International Conference on Software Engineering*, pp. 282 - 291, 2004.

[32] Navarro, G. “A Guided Tour to Approximate String Matching,” *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[33] Reussner, R. H., Schmidt, H. W. and Poernomo, I. H. “Reliability prediction for component-based software architectures,” *The Journal of Systems and Software*, vol. 66, pp. 241–252, 2003.

[34] Ruiz, I. J. M., Nagappan, M., Adams, B. and Hassan, A. E.“Understanding Reuse in the Android Market,” *Proceedings of IEEE*

20th *International Conference on Program Comprehension*, Germany, pp.113–122, 2012.

[35] Subramanyam, R. and Krishnan, M. "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[36] Syer, D., Adams, B., Zou, Y. and Hassan, A. "Exploring the development of micro-apps: A case study on the blackberry and android platforms," *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 55-64, 2011.

[37] Taibi, F. "Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software," *International Journal of Computer, Information, System and Control Engineering*, vol. 8, no. 1, pp. 114 – 120, 2014.

[38] Washizaki, H., Yamamoto, H. and Fukazawa, Y. "A metrics suite for measuring reusability of software components," *Proceedings of the 9th Software Metrics Symposium*, pp. 211-223, 2003.

[39] Zhou, W., Zhou, Y., Grace, M., Jiang, X. and Zou, S. "Fast, scalable detection of Piggybacked mobile applications," *Proceedings of the 3rd ACM conference on Data and application security and privacy*, pp. 185-196, 2013.

[40] Zhou, Y. and Leung, H. "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771-789, 2006.