Mining Android Applications for Discovering API Call Usage Patterns and Trends

Student Name: Yash Lamba

IIIT-D-MTech-CS-2010097

Sep 21, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Ashish Sureka (Chair)
Apala Guha
Pavan Chittimalli

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science

©2014 IIIT-D-MTech-CS-2010097
All rights reserved

Keywords: Mining Software Repositories, Empirical Software Engineering and Measurements, Android Applications, API Call Usage Patterns

Certificate

This is to certify that the thesis titled **"Mining Android Applications for Discovering API Call Usage Patterns and Trends"** submitted by **Yash Lamba** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by him under my guidance and supervision in the Software and Analytics group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Ashish Sureka
Indraprastha Institute of Information Technology, New Delhi

Abstract

Software libraries and frameworks, consisting of a collection of Class and Interface definitions, provide a mechanism for code reuse by providing methods, APIs, components (generic functionality) and a support structure for developers to build applications, products and solutions. KitKat, Jelly Bean, Ice Cream Sandwich, Honeycomb and Gingerbread are different versions (open-source) of Android, one of the most popular mobile platforms in the world. In this thesis, we present the results of our large-scale (consisting of $1,120$ open-source applications and 17.4 million lines of code) API usage analysis of Android applications. Our work is motivated by the need to mine actual Android API usage, frequent API call usage patterns and trends to understand and generate empirical data on how developers are using the mobile platform in their applications.

Extracting popular and frequently-invoked methods, API packages and API call-usage patterns is useful to both the API Producers and API Consumers. For example, API Producers can view the quantitative data on API usage as a feedback from users on the relevance, usability and applicability of the respective APIs. We conduct a series of experiments on analysing the Android platform API usage (usage of different packages, usage of methods, usage across categories) and present the results of our analysis using graphs such as *Bubble Chart*, *Radar Chart*, *Heat-Map* for effective visualization of the results and for extraction of actionable information.

Acknowledgments

Life is a journey and this thesis is one of the more memorable paths that I have taken till date. A lot of people have been wonderful companions during this journey and I would like to take this opportunity to thank them.

Dr. Ashish has been the ideal mentor and the ideal teacher. I never imagined myself doing Master's, let alone research. This thesis has only been possible because of you. I acknowledge today, that being a part of your research group was a big motivation to do my master's. Thank you for everything that you have taught me directly or indirectly over the past 33 months. Without your encouragement, inspiration and guidance, my life wouldn't have been the same.

Next, I would like to thank Manisha Khattar (PhD Scholar at IIIT-Delhi) for her help and suggestions during the course of the project, and my friends who encouraged me during my work. Finally, a thank you to my parents and sister for their selfless love, support and encouragement. This one is for you!

Contents

List of Figures

List of Tables

Chapter 1

Research Motivation and Aim

1.1 Introduction

In today's world, Android is the biggest name in the smartphone market and is expected to remain in this position for several years to come [1]. One of the many reasons for Android's success has been its Play Store. Since the inception of Android, developers have been heavily contributing to the ecosystem by coming up with new and complex applications that cater to the various needs of the masses. Consequentially, as of August 10, 2014, the number of Android applications on Google Play[1] stands at $1,327,838$.

Developers are greatly motivated, to build more and more applications for Android, by the huge number of downloads that Android applications receive. For instance, between September 2012 - July 2013, the cumulative number of application downloads from the Google Play Store grew from 25 billion to 50 billion [2]. Moreover, going by the current trends, by 2015 Android would be the most used Operating System ahead of Windows 8 [1]. Given all this information, it is no longer feasible to continue to look at Android applications as just 'apps'. There is a need to look at them as untapped software repositories, which are rich sources of information that can be used to draw actionable insights similar to those drawn from repositories of software made for the traditional computers.

1.2 API Call Usage Patterns

An *API Call Usage Pattern* (ACUP) is a pattern of method calls, wherein all the methods are invoked together in the same user-defined method. ACUPs are important since, they can help in the identification of standard library/API usages and fault location [2] [3]. Generally, ACUPs are quite intuitive and of common knowledge to developers, however, they are typically not documented, which remains a big concern [4].

For instance, consider the following commonly occurring ACUP - "measure getMeasuredWidth

[1]http://www.appbrain.com/stats/stats-index
[2]http://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/

getWidth getHeight getMeasuredHeight". This ACUP is widely used to obtain the measurements of a view (basic building component of user interface components that is responsible for drawing and event handling). Based on the measurements, the application takes some decision and adjusts itself. Our study, provides empirical evidence that adjusting the application to support multiple screens, is an important concern for android application developers. Similarly, "getLatitude getLongitude getSpeed getAltitude getBearing getProvider hasAltitude hasBearing hasSpeed" is a frequent API call usage pattern because many smart phones have the functionality to accurately detect the location of the user. Documenting such ACUPs is important and we address this problem as part of our work.

Moreover, at times multiple ACUPs differ only by a few method invocations. Merging such ACUPs to get a list of closely related methods can help in identifying a broad functionality and impacted software components. For example, the methods *getMinimumHeight* and *getMinimumWidth* can easily be combined with the ACUP - "measure getMeasuredWidth getWidth getHeight getMeasuredHeight", because they also deal with the measurements of a view. Merging similar ACUPs can enable us to identify a broad functionality with all possible variations in usage. Since, such a study has not been previously done, we introduce the concept of Merged ACUPs and incorporate it in our work.

1.3 Growing Android API

The Android API itself has grown enormously since the beginning of Android (as of version 20, the API consists of 18,881 classes). The intent of the Android API and other third party APIs for Android, is to provide access to the features available on a device such as WiFi, Bluetooth, GPS and graphics. With advancement in smartphone and tablet technology, the APIs are also becoming more advanced. In order to maintain the level of growth in the Android ecosystem, it is important for API Producers and API Consumers to be in sync with each other. The information that we get after mining Android applications can play a crucial role, by acting as feedback from the developers to API Producers and to other fellow developers.

Moreover, studying the adoption of changes in the API, by application developers is also important. API Producers constantly change the Android API either to improve the quality of the API, or to correct known issues. However, they have no concrete way of knowing whether the changes made by them are being used or not. API Producers also require feedback on the usability, productivity and effectiveness of the changes. Analysing source code for adoption of changes made to the API can help in addressing this issue.

1.4 General Importance for API Producers and API Consumers

Mining important ACUPs, identifying the functionality associated with them, identifying popular methods, classes, interfaces, and API packages is important for both API consumers and

API producers for different reasons:-

- *API Consumer (Developer)* - From a developer's point of view knowing how other developers use APIs in Android applications can tell them of the practices that they should or can potentially follow in their own applications. It can help them choose between libraries and components that provide the same functionality, since they would know which is more popular among their peers. Knowing such minute but important details can help a developer to produce better applications, which would be good for the entire ecosystem.
- *API Producers* - API Producers can view the quantitative data on API usage as feedback from users on the relevance, usability and applicability of the respective APIs. Knowing how their library is being used can help them identify components of their APIs that are not very popular. They can possibly identify the reason(s) behind the lack of popularity and improve those components. Even for the components that are extensively used, they can find out whether some change is required to improve the user experience.

1.5 Research Aim

The research aim of the work presented in this thesis is the following:-

1. To conduct an in-depth and focused empirical analysis for identifying popular ACUPs, popular packages, classes, interfaces and methods in Android applications.
2. To identify the most commonly implemented functionalities/affected components in applications based on the most popular ACUPs and Merged ACUPs.
3. To investigate the effect of changes in Android APIs on applications.
4. To identify popular ACUPs across 12 different categories of applications like Games, Multimedia, Navigation, System, Wallpaper etc.
5. To understand the difference between different categories of application based on the usage of the top packages and classes across different categories.
6. To provide *Advanced Visualizations* for API Producers and API Consumers that communicate the results of our work in an effective manner and help in understanding how developers use the Android API.

In this thesis, we present the results of our large-scale API usage analysis of Android applications. We conduct a series of experiments on analysing Android platform API usage and present the results of our analysis using advanced visualizations that can be used to extract actionable information.

Chapter 2

Related Work and Novel Research Contributions

2.1 Related Work

In this Section, we review work that is closely related to our study, and list the novel contributions of our work in context to existing work. We divide the related work into the following three lines of research:-

2.1.1 Mining API Call Usage Patterns

Xie et al. propose a method to mine frequent and succinct API usage patterns which includes sequencing information among method calls. Their framework is based on existing code search engines and a frequent sequence miner [5]. Zhong et al. extend their own work in [5] to create a recommendation engine that recommends API call usage patterns and code snippets [6]. Kagdi et al. present an approach to mine frequently-occurring ordered sets of method call usages, taking into account their proximal control constructs (e.g., if-statements), in the source code [7].

Liu et al. propose a method to automatically extract software library usage rules [8]. Their approach uses a model checker to check a set of software library usage rule candidates against known good programs using that library, and identifies valid rules based on the model checking results. These valid rules can help programmers learn about common software library usage [8]. Nguyen et al. create a system that guides developers in adapting to changes in APIs, by learning API call usage adaptation patterns from other developers who have adopted the new changes [9]. Zhang et al. create a system that provides API parameter recommendations based on the learning they do from existing code samples [10].

2.1.2 Mining API Usage Trends and Popularity

Holmes et al. propose a technique for quantitatively determining how existing APIs are used, and demonstrate its application to Eclipse. Their technique is aimed at enabling application developers to easily understand how others have used the APIs and API Producers to easily understand how their APIs are being used [11].

Mileva et al. mine hundreds of open-source projects for their library dependencies and determine global trends in library usage as well as show important emerging trends in library usage [12]. In another study, they analyse a large set of open-source projects and their external dependencies in order to observe the popularity of their APIs and to give recommendations of the kind:- "Projects are moving away from this API element. Consider a change." [13].

Lammel et al. describe an approach to large-scale API-usage analysis of open-source Java projects [14]. Roover et al. perform a multi-dimensional exploratory study of API usage using a large corpus of Java projects. One of their dimensions of study includes API metrics like the number of times a class is extended among others [15].

2.1.3 Mining Related to Android API

Minelli et al. perform an in-depth investigation of a corpus of Android application and find that mobile applications are significantly smaller than traditional softwares and rely heavily on external APIs, with two-thirds of methods invocations being API calls [16]. They state that understanding APIs is integral to understanding Android applications, since the reliance of applications on APIs is too much.

Wang et al. try to identify classes that give the most problems to developers by comparing the list of frequently mentioned API classes in posts on stackoverflow.com against the list of API usage frequency [17]. Vásquez et al. study API evolution and try to associate it with fault-proneness and stability of applications [18]. They find out that the more used API components are the ones that undergo less change. Similarly, Mcdonnell et al. study API adoption in the Android ecosystem. They study correlations between API evolution rate, API usage, defect proneness and stability of applications [19].

2.2 Novel Research Contributions

The novel research contributions of this work are:-

1. While there has been work done on mining API call usage patterns for Java, *we are the first to mine ACUPs for Android applications (as a whole and across different categories of applications). We introduce the concept of Merged ACUPs and show that they can be linked to specific components of Android applications.*

2. While there is data on the popularity of API packages, classes, methods for Java, *this*

thesis gives most popular packages, classes and methods for Android applications. We also see the occurrence of these structural units across 12 different categories.

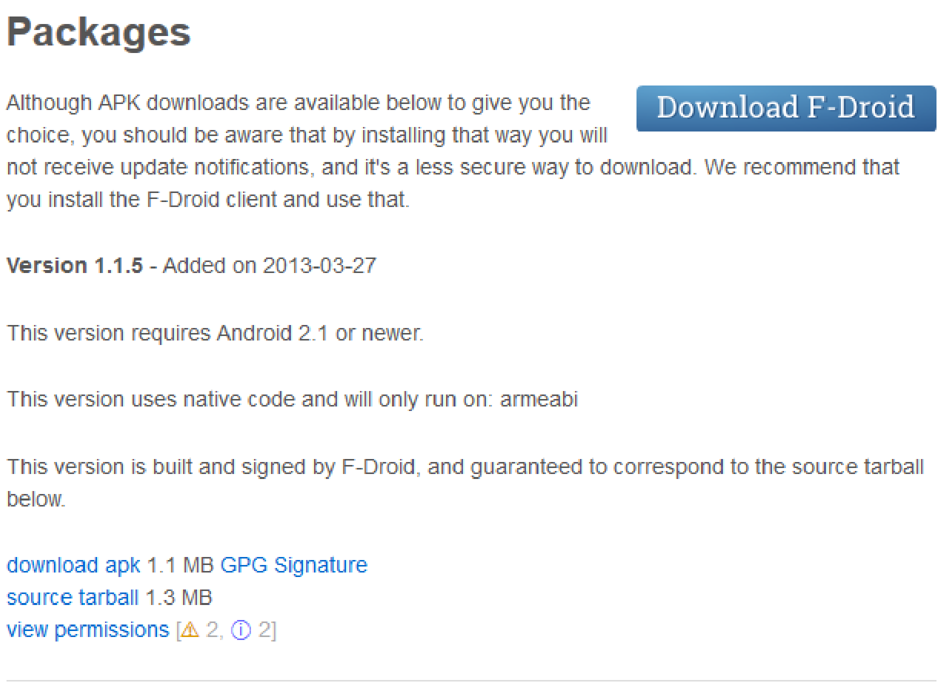
3. Although, Mcdonnell et al. have done a study on API adoption in the Android ecosystem [19], *we focus on identifying the most popular API changes and the extent to which the changes have been adopted by application developers.*

4. Our work is also unique in the fact that we make use of *Advanced Visualizations for effective communication of our research results.* None of the works done before us, use advanced visualizations like *Radar Chart*, *Heat-Map*, *Bubble Chart* etc. to graphically present results on API usage.

Chapter 3

Experimental Dataset

We download the source code of 1, 120 Android applications from *F-Droid*[1] which is a software repository of Free and Open Source applications for the Android platform. We choose *F-Droid*, as it is not only a popular platform containing free and open-source software, but also each application on *F-Droid* is available for download on the Google Play Store.


Packages

Although APK downloads are available below to give you the choice, you should be aware that by installing that way you will not receive update notifications, and it's a less secure way to download. We recommend that you install the F-Droid client and use that.

Version 1.1.5 - Added on 2013-03-27

This version requires Android 2.1 or newer.

This version uses native code and will only run on: armeabi

This version is built and signed by F-Droid, and guaranteed to correspond to the source tarball below.

download apk 1.1 MB GPG Signature
source tarball 1.3 MB
view permissions [⚠ 2, ⓘ 2]

Figure 3.1: **F-Droid Interface**:- *F-Droid* interface showing version number, date and link to download source code

Figure 3.1 shows the interface of *F-Droid*. *F-Droid* lists the most recent versions (max 4) of each application, along with the version number, the date on which the version was added to *F-Droid*, and the links to the apk and source tarball. Generally, we see that developers only

[1]https://f-droid.org/

make the most recent version of their applications available. Nevertheless, a few developers do make the last 3 versions available.

Table 3.1 provides details on the size of the dataset. As of July 5, 2014, there were a total of 13 categories in *F-Droid* and we downloaded the data for 12 categories, as there was only one application for one of the categories (Children). We downloaded the source code of the last available version of each application on July 5, 2014. The download was done manually from the individual pages of the applications, using the source tarball link. The version number and publication dates were also recorded for further use. The oldest application in our dataset is a game called Sokoban (Version 1.11) dated January 5, 2011. The newest application is WiGLE Wifi Wardriving (Version 2.2). It belongs to the category Internet and was added on July 5, 2014.

Table 3.1: Number of Android Applications (#APP), Java Files (#JVF), Transactions (#TRN) and Lines of Code (#LOC) across 12 Categories in F-Droid

	Category	**#APP**	**#JVF**	**#TRN**	**#LOC**
1	Office (OFF)	226	13,518	63,558	2,841,209
2	System (SYS)	217	18,367	65,183	3,135,993
3	Multimedia (MUL)	128	6,974	33,865	1,428,876
4	Games (GMS)	127	4,994	23,717	1,031,776
5	Internet (INT)	126	25,124	96,594	4,879,252
6	Navigation (NVG)	79	5,580	27,543	1,154,570
7	Science & Education (SCN)	61	3,710	17,374	804,686
8	Wallpaper (WLP)	44	824	3,417	164,887
9	Reading (RDG)	37	3,947	17,409	682,701
10	Development (DEV)	33	2,035	9,699	364,351
11	Phone & SMS (PHN)	28	250	10,486	452,583
12	Security (SCR)	14	2,155	11,289	486,592
	SUM	**1,120**	**87,478**	**380,134**	**17,427,476**

As shown in Table 3.1, our dataset consists of 87,478 Java files having 17.42 million lines of code. The dataset also has 380,134 transactions, where a transaction is a group of methods (both user-defined and API methods) that are invoked together in a user-defined method. The table is sorted in the decreasing order of the number of applications in a category. The table reveals that the category *Office* has the highest number of applications then followed by *System*, *Multimedia* and so on.

It is interesting to see that though, the category *Internet* has 100 applications less than category *Office* (which has maximum application), it is way ahead of the other categories in terms of lines of code and the number of transactions. On an average, an application in the category *Internet* has about 38,725 lines of code, with the next best being the category *Security* with 34,756 lines of code. However, if we consider only the categories with 100 or more application, the next best is the category *System* with roughly 14,451 lines of code. Also, we see that the category *Wallpaper* has the smallest average number of transactions per application (about 78). This shows that writing an application that allows users to play around with images, requires little code in comparison to the other categories.

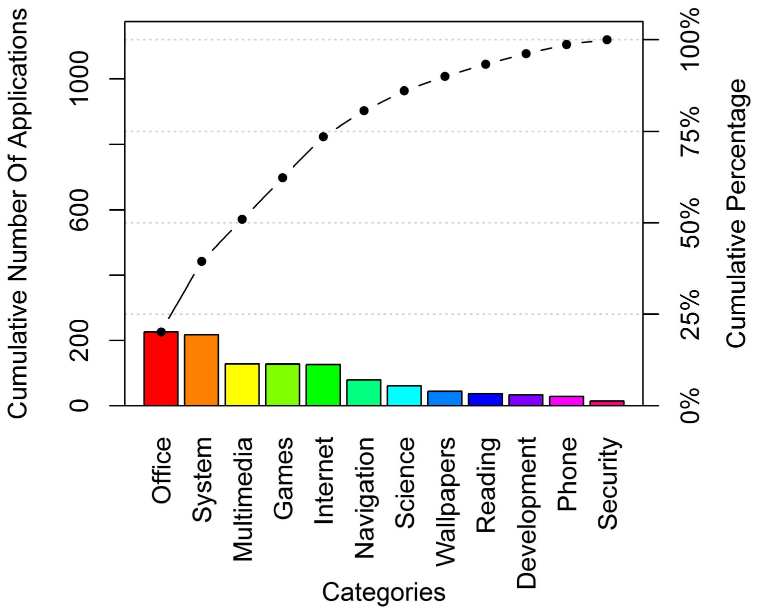


Figure 3.2: **Distribution of dataset across the** 12 **categories**:- *Pareto chart* containing both bar and line graph showing the number of applications downloaded in each of the 12 categories and the cumulative total (represented by the line)

Figure 3.2 gives a graphical distribution of our dataset using a unique combination of a bar and line graph. The *Pareto Chart* shows the total number of Android applications and the distribution of the applications across 12 different categories. The bars are arranged in the descending order of the number of applications and the line represents the cumulative total. The left vertical axis represents the cumulative number of applications and the right vertical axis is the cumulative percentage of applications. The graph reveals that the cumulative function is a concave function and the distribution is skewed, as 50% of the applications in the dataset belong to 3% categories:- *Office*, *System* and *Multimedia*.

Chapter 4

Experimental Results

We conduct a series of experiments to extract useful patterns and actionable information for the API Producers and API Consumers. Each of the following 5 sections, describes multiple experiments consisting of the mining goal, procedure or approach, and the findings. All experiments are conducted on a 64 bit Windows Server 2012 Datacenter having 64 GB RAM and 2 Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50 GHz processors. The code for all the experiments is written in Java and 15.26 GB of memory is made available to the Java Virtual Machine for each experiment.

4.1 Finding Frequent API Call Usage Patterns

As previously mentioned, an ACUP is a pattern of method calls, wherein all the methods are invoked together in the same user-defined method. The aim of the first experiment is to unravel frequent ACUPs that are present in Android applications. We begin with the assumption made by Kagdi et al. in [4] that frequently-occurring ACUPs reflect candidates for standard usages of an API. Hence, we need not separate user-defined methods and API methods at any point of time. The API methods would outnumber the user-defined methods and make their way into the list of common ACUPs. As we shall see later, the results from this experiment completely justify this assumption.

4.1.1 Standard ACUPs

For every Java file in each of the $1,120$ Android applications, we traverse the *Abstract Syntax Tree (AST)* of the source code and make a note of the method declarations. We remove the objects associated with the methods and the method parameters, and focus only on the full method names. For every declared method, we make a note of all the invoked methods (both user-defined and library methods). Each such group of method invocations makes a transaction, with each individual method being an item in the transaction. The transactions are then used to generate frequent itemsets. We use SPMF[1], which is an open-source data mining library written in Java, for mining closed and maximal frequent itemsets. We mine frequent itemsets having support greater than 0.001, since the entire dataset has a very large number of transactions (380,134).

A pattern (sequence of items) is redundant if it is a subsequence of another frequent sequence or pattern. To ensure that our results have no redundancy, we use the AprioriClose algorithm for mining the itemsets [20]. The algorithm does not include a proper subset, if it has exactly the same support as its superset, where support refers to the the number of transactions in which the itemset occurs. Thus, in our case the support is the number of methods in which all the methods of a pattern have been invoked.

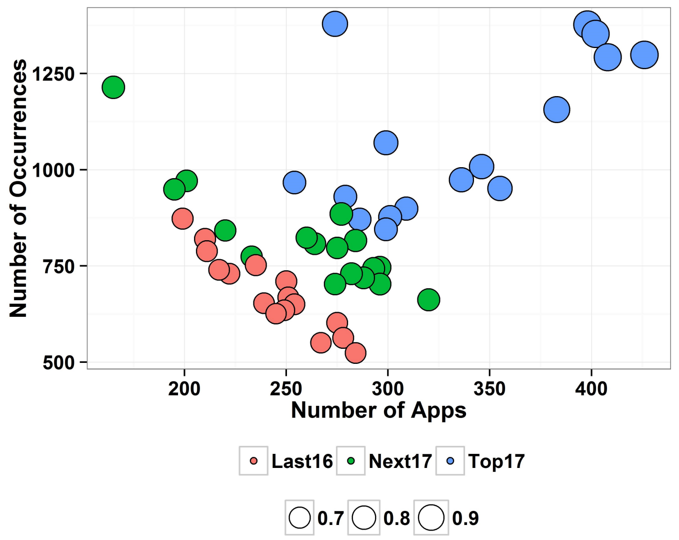


Figure 4.1: **Most Popular ACUPs**:- *Bubble Chart* displaying the top 50 ACUPs found in the dataset.

We finally filter our frequent itemsets based on the size of the itemset. For an itemset to be considered an ACUP, its minimum size must be 4. After applying the size filter, we are left with 1070 ACUPs. On our test bed, finding transactions takes just over 5 hours and 25 minutes. It takes 24 more minutes to find the maximal frequent itemsets. Hence, the entire process of finding ACUPs takes about 5 hours 50 minutes.

Figure 4.1 is a *Bubble Chart* that displays the 50 most popular ACUPs found in the dataset. Each bubble represents an ACUP. The x-axis represents the number of applications in which the ACUP is present, while the y-axis represents the total number of occurrences of each ACUP. The size of the bubble denotes the value of the third dimension, that in our case is the *Significance Index* which is calculated as follows:-

$$SI = \frac{1}{3} * \sum_{i} \frac{i}{Max(i)} \tag{4.1}$$

where SI is *Significance Index* and i $\in \{Num\ Occurrences, App\ Frequency, Num\ Categories\}$.

[1]http://www.philippe-fournier-viger.com/spmf/index.php

Table 4.1: Top 3 ACUPs with No Method in Common

	ACUP	Significance Index	Occurrences	App Freq
1	[setPositiveButton setTitle setMessage setNegativeButton]	0.98	1,298	426
2	[add hasNext next iterator]	0.75	1,214	165
3	[setContentView findViewById setText setOnClickListener]	0.74	662	320

In Equation 4.1, we divide the value of an attribute of the ACUP, by the maximum value of the attribute attained in the result set. The values thus obtained, for all the attributes for a particular ACUP, are multiplied with 0.33 and then summed to get the *Significance Index*. The *Significance Index* incorporates the presence of an ACUP across different categories, the total number of occurrences of an ACUP and the number of applications that use the ACUP.

We observe that a good number of popular ACUPs are present in the 200 to 300 applications range and occur anywhere between 500 to 1,000 times. However, the top 17 ACUPs are well scattered and separate themselves from the rest of ACUPs on the basis of both application frequency and total number of invocations. Moreover, we see that the top ACUPs entirely consist of methods provided by APIs, thus, justifying the assumption that frequently-occurring ACUPs reflect candidates for standard usages of an API.

However, we notice that there are several cases where there is only a difference of one or two method invocations between two ACUPs. We also observe that the broad functionality between the two ACUPs with such minor difference is always the same and hence, we decide to merge such ACUPs. We cover the process of merging ACUPs in the next section.

Table 4.1 refers to the top 3 ACUPs having no function in common. The most popular ACUP in Android is on *AlertDialog*. The functions *setPositiveButton* and *setNegativeButton* are used to set listeners that are invoked depending on the button the user presses, while the *setMessage* and *setTitle* methods are used to set the textual details on the *AlertDialog*. Moreover, *AlertDialog* is a very important component since, the ACUP related to it is present in about 38% of all applications. Similarly, the ACUP given by "setContentView findViewById setText setOnClickListener" also deals with a user interface (UI) component of applications.

4.1.2 Merged ACUPs

Merging the ACUPs is done by applying *Jaccard Index*. The *Jaccard Index* (also known as the Jaccard Similarity Coefficient) is a well-known statistic or metric used for comparing the similarity and diversity of sample sets[2]. The Jaccard Coefficient measures similarity between finite sample sets and is computed as the size of the intersection of the two given sets divided by the size of the union of the sample sets. Thus, the value of *Jaccard Index* always lies between 0 and 1.

[2]http://en.wikipedia.org/wiki/Jaccard_index

Merging ACUPs is an iterative process that takes about 2 seconds to complete on our test bed for the entire dataset (1070 ACUPs). We iteratively keep merging similar ACUPs till the time they satisfy the *Jaccard Index* criteria. We choose a *Jaccard Index* threshold of 0.75. The value is chosen (after experimenting with several values) to ensure that there are absolutely no false positives for ACUPs of any size. However, as a result of this decision choice, we observe that a few similar ACUPs do not merge. Following is a concrete example of the application of *Jaccard Index* and the impact of our threshold on the merging process. Consider the following two ACUPs:-

1. ACUP 1 - [getPaddingBottom, max, getPaddingTop, getVisibility, getMeasuredHeight, getPaddingLeft, getPaddingRight, getLayoutParams].
2. ACUP 2 - [getMeasuredWidth, max, getVisibility, getMeasuredHeight, getPaddingLeft, getPaddingRight, getLayoutParams].

The size of ACUP 1 is 8 and size of ACUP 2 is 7. The intersection of the two ACUPs is:- [max, getVisibility, getMeasuredHeight, getPaddingLeft, getPaddingRight, getLayoutParams]. Similarly, union of the two ACUPs is:- [getPaddingBottom, max, getPaddingTop, getVisibility, getMeasuredHeight, getPaddingLeft, getPaddingRight, getLayoutParams, getMeasuredWidth]. The number of items in the intersection of the two given ACUPs is 6, while the number of items in the union is 9. The *Jaccard Index* value is $6/9 = 0.667 < 0.75$. In this case, we see that even though the two ACUPs are almost similar and merging them cannot be deemed incorrect, yet we do not merge them, since doing the same requires lowering of the *Jaccard Index* threshold, which introduces false positives.

Following is an example where we are successfully able to merge ACUPs and iteratively move from multiple small ACUPs to a large consolidated ACUP. The Merged ACUP [LT, nextTree, setTokenBoundaries, nextNode, add, create, errorNode, recover, consume, id, LA, becomeRoot, hasNext, reset, getTree, addChild, match, pushFollow, reportError, rulePostProcessing, nil] is obtained from similar ACUPs, few of which are listed below:-

- LA, LT, addChild, setTokenBoundaries, add, errorNode, reportError, recover, rulePostProcessing, nil
- becomeRoot, LA, LT, addChild, setTokenBoundaries, add, errorNode, reportError, recover, rulePostProcessing, nil
- becomeRoot, LA, LT, addChild, setTokenBoundaries, nextNode, add, errorNode, reportError, recover, rulePostProcessing, nil
- LA, LT, getTree, addChild, setTokenBoundaries, pushFollow, add, errorNode, reportError, recover, rulePostProcessing, nil
- LT, nextTree, setTokenBoundaries, add, recover, errorNode, becomeRoot, LA, addChild, getTree, pushFollow, reportError, nil, rulePostProcessing

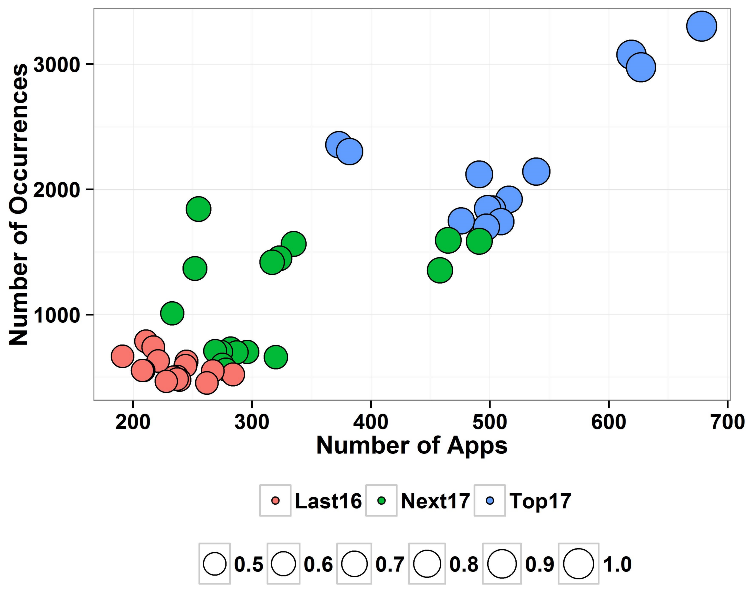


Figure 4.2: **Most Popular Merged ACUPs**:- *Bubble Chart* displaying the top 50 Merged ACUPs found in the dataset.

A Merged ACUP represents a broad functionality, and so the number of occurrences and application frequency of a Merged ACUP is the same as the number of occurrences and application frequency of the broad functionality represented by the Merged ACUP. Since, all methods of a Merged ACUP are not invoked together in some user-defined method, we make note of all the methods invoking any constituent ACUP (an ACUP that was merged to get the Merged ACUP). For each constituent ACUP, we also make note of the application that uses it. These method and application details are then used to come up with an accurate count of the number of occurrences and application frequency of the Merged ACUP.

Figure 4.2 shows the top 50 Merged ACUPs. In an ideal scenario, these 50 ACUPs would have represented the top 50 functionalities/components. However, since our merging procedure does not merge all similar ACUPs, these 50 Merged ACUPs represent a slightly fewer number of functionalities. Nevertheless, the top 17 Merged ACUPs have more than 1500 occurrences, though the number of applications in which they are found, varies from 350 to about 700. A majority of the top 17 ACUPs are found in 350-450 applications. The last 16 of the top 50 Merged ACUPs (35-50) lie in the 150-300 applications range and occur less than 1000 times in our dataset.

Table 4.2 displays the top 5 Merged ACUPs. We see that the topmost Merged ACUP contains the topmost ACUP from Table 4.1. In addition to the topmost ACUP, Merged ACUP 1 also contains the methods *show* and *create*, which are used for creating and displaying the *AlertDialog*. Hence, Merged ACUP 1 is a good proof of concept that combining a number of similar ACUPs, can help us represent a broad functionality. With a more sophisticated procedure for merging, it should be possible to identify almost all methods that are related to a particular functionality. Table 4.2 also shows that the application frequency for Merged ACUP 1 is 678, while the application frequency for ACUP 1 is 426 (Table 4.1). Now, we know that *AlertDialog*

Table 4.2: Top 5 Merged ACUPs

	Merged ACUP	SI	Occurrences	App Freq
1	[setMessage, setTitle, setPositiveButton, setNegativeButton, show, create]	1.0	3302	678
2	[append, setView, getText, getResources, setPositiveButton, createAuthorsLink, fromHtml, setMovementMethod, show, setText, getString, getInstance, create]	0.94	2974	627
3	[query, getContentResolver, moveToFirst, getString, close]	0.75	2357	373
4	[setContentView, setOnClickListener, findViewById, setText]	0.55	662	320
5	[setTag, getTag, inflate, setText, findViewById]	0.53	710	269

is used in 60.5% of all applications, which is significantly higher 38%, as suggested by Table 4.1. This demonstrates that not all developers use *AlertDialog* in the same manner. On further investigation, we find that depending on the type of Alert the developers wish to give to the user, they tend to use only *setPositiveButton* or *setNegativeButton* but not both.

Our results indicate that *AlertDialog* is very important to developers. Comparing the constituent ACUPs and the Merged ACUPs can tell us the different ways in which *AlertDialog* is used. The different ways of using a component is very important for API Consumers, since it can enable a developer to choose and implement an ACUP depending on the application requirement. Such information also makes it easier for developers to identify mistakes in existing implementations of the functionality, as they now have a reference ACUP to compare their code with.

4.2 Package, Class and Interface Popularity

The success or popularity of any marketable product is measured in terms of the number of units sold. Going along these lines, we can use the number of times a structural unit is used, to judge its popularity. Our goal is to explore the extent of referencing relationship between Android platform packages and classes, and mobile applications. For our experiment, we count the number of times a structural unit is imported across all Java files in the dataset. The number of applications gives us an idea of the popularity of the structural unit across the dataset, while the number of Java files allows us to understand the extent of use within individual applications. We measure the popularity of the following three structural units:-

1. *Class* - In case of classes, we count the number of *import <Class Name>* statements for each individual class. In addition, for each class we also count the number of times it has been inherited by visiting the class declarations in the source code.

2. *Interface* - For interfaces, we follow the exact procedure as in the case of classes. Here again, we count the number of times an interface is implemented by a class in our dataset.

3. *Package* - In case of packages, we simply count the *import* statements in the source code.

Table 4.3: Popular Packages, Classes and Interfaces

	Top 5 Packages	
	Package Name	**Import Count**
1	java.util	65, 110
2	android.content	56, 425
3	android.view	49, 433
4	android.widget	43, 490
5	java.io	49, 433
	Top 5 Classes	
	Class Name	**Import Count**
1	android.content.Context	22, 111
2	android.view.View	15, 143
3	android.os.Bundle	13, 633
4	android.content.Intent	12, 261
5	java.io.IOException	12, 009
	Top 5 Extended Classes	
	Class Name	**Extend Count**
1	android.app.Activity	3, 659
2	android.os.AsyncTask	1, 513
3	org.bouncycastle.asn1.ASN1Object	1, 328
4	android.preference.PreferenceActivity	1, 220
5	android.content.BroadcastReceiver	1, 198
	Top 5 Implemented Interfaces	
	Interface Name	**Implement Count**
1	android.view.View.OnClickListener	705
2	com.actionbarsherlock.ActionBarSherlock.OnMenuItemSelectedListener	216
3	com.actionbarsherlock.ActionBarSherlock.OnCreatePanelMenuListener	216
4	com.actionbarsherlock.ActionBarSherlock.OnPreparePanelListener	216
5	com.actionbarsherlock.ActionBarSherlock.OnActionModeFinishedListener	215

But, we observe that in all of the 1, 120 applications the developers do not import entire packages by using *import <package name>.**. They import the specific class/interface that they wish to use, which is a good programming practice since it reduces ambiguity in the source code. So in order to get the popularity of a package, we simply add up the number of imports of all the classes and interfaces that the package contains.

Table 4.3 shows the packages and classes with the highest usability. Identifying the most popular packages and classes is important for API Producers as they can focus their resources in making sure that such program/structural elements are well tested and reviewed, as they are high impact elements. Similarly, identifying less popular packages is also useful as the API Producers can then work towards fixing any issue that may be limiting the usage of the element. From the developers' point of view such information is important as it can help them in identifying or preventing any issue arising due to the usage of some less popular element. In case they are using some less popular alternative of one of the popular elements, they can switch to the better alternative in order to improve the quality of the application.

java.util contains the collections framework and utility classes like string tokenizer, random-number generator etc. Since, almost all applications require some sort of collection (HashSets, ArrayLists, HashMaps etc.) for their basic working, it is unsurprising that *java.util* is the most used package. *android.content* contains classes for accessing and publishing data on a device. The importance of the package is established by the fact that *android.content.Context*, the most used class, belongs to this package. The case of *android.view* is also similar. *android.view* contains classes that provide the basic building blocks for UI components. Its class *android.view.View* is the second most used class, which throws light on the UI centric nature of mobile applications.

The 4th package -*android.widget* - contains (mostly visual) UI elements that are used on application screens. A deeper inspection shows us that this package is mostly used for Toasts (a floating view that contains a quick message for the user), again highlighting the importance of the user interface in Android applications. Rounding the top 5 list is *java.io*, which is a standard java package that provides classes for performing input-output operations.

In this list of top classes, we find *android.os.Bundle* at the third place. *android.os.Bundle* is used to store a set of state variable, when the application changes state. For example, it is used to store the state of application, when it is minimized. These stored values are used to bring the application back to its original state, when the application is maximised. This highlights the emphasis that developers put on maintaining application state so as to give the users a better experience. Finally, *android.content.Intent* is used by the applications to hold abstract description of any action to be performed. It allows application components to request functionality from other Android components. For instance, phone calls are made using the intents Intent.ACTION_DIAL and Intent.ACTION_CALL.

Although, the top most classes and packages are along expected lines, the list of most extended classes and interfaces are surprising. The third most extended class does not belong to the Android API. It belongs to the *Bouncy Castle* library[3], which is a library that provides cryptographic functions. This clearly indicates that developers have little faith in the default Android libraries providing security functions (*java.security*). They rely on *Bouncy Castle* for most of the security needs of applications.

Similarly in the case of interfaces, even though, the topmost interface is *android.view.View.-OnClickListener* (which listens to click events), the remaining four interfaces belong to the *ActionBarSherlock* library[4]. *ActionBarSherlock* is known to facilitate the use of the action bar design pattern across all versions of Android with a single API. Therefore, it is considered superior to its Android counterpart. This is another case, where application developers do not find some part of the Android API up to the required standards. For third party API Producers, these results are very encouraging. The results indicate that if the quality and usability of an API is good, developers are willing to use third party APIs rather than the ones provided by Android.

[3]https://www.bouncycastle.org/

[4]http://actionbarsherlock.com/

4.3 Android Applications and Change in API Versions

4.3.1 Android API Evolution

The huge growth in the number of applications available for Android is a direct reflection of the ease of developing applications for Android. The Android API and various third party APIs facilitate the development of new and sophisticated applications. The Android API is a huge collection of libraries that gives access to all features that are available in a smartphone/tablet today. Over the years, the Android platform has grown at a great pace, in order to keep in sync with the consistent hardware and software innovation in the smartphone/tablet industry.

Table 4.4: Recent Android API Versions

Version	**Packages**	**Classes**	**Methods**	**NCSS**
14	416	11,660	85,625	42,924
15	438	11,901	87,270	680,821
16	501	13,050	95,717	744,528
17	527	14,016	101,948	796,979
18	566	15,145	110,183	855,942
19	628	16,659	120,059	944,080
20	667	18,881	133,607	1,066,601

Table 4.4 gives the number of structural elements present in the last 7 versions of the Android API. Android API level 14 was released in October 2011. At that time, the number of *Non Commenting Source Statements (NCSS)* was 42,924. As of Android Level 20, the number of source code statements stands at 1,066,601 which is nearly 25 times of Level 14. Even the number of packages in Level 20 is 60% more than the number of packages in Level 14.

With each release, the Android API undergoes a lot of modifications, that are recorded in the *specification difference sheet*[5] for each version. In each new version, a few new elements (packages, classes, interfaces, methods, fields) are added or deleted due to refactoring, feature addition or deprecation of existing functionality (*API evolution*). Apart from this, a lot of existing elements also undergo modifications. Our objective is to measure the extent to which the framework changes (in terms of addition and deletion of packages, classes and methods only) over various released versions of Android.

Table 4.5 indicates the exact number of additions/removals made in the Android API over the past few versions. We observe a heavy modification between API versions 13 and 14 with 5 packages, 90 classes and 237 new methods being added and 22 methods being removed. In total 354 additions/deletions were made in terms of classes, packages and methods. A similarly large change is observed between API Levels 15 and 16 with a total of 424 changes being made. With each new release, the median number of packages, classes and methods added are 3, 57, 145 respectively. The median number of deletions for packages and classes is 0 and for methods it is 5. We use median instead of average, to mitigate the impact of outliers.

[5]https://developer.android.com/sdk/api_diff/<x>/changes.html, where x is the current version

Table 4.5: Android API Modification History with Number of Components Added or Removed

Version	Packages Added	Classes Added	Classes Removed	Methods Added	Methods Removed	Total Num Changes
11	3	109	0	373	24	509
12	3	5	6	64	2	80
13	0	2	0	18	0	20
14	5	90	0	237	22	354
15	4	12	0	25	0	41
16	0	57	0	349	18	424
17	2	41	2	109	19	173
18	2	61	36	145	4	248
19	6	78	0	223	5	312
Sum	**25**	**455**	**44**	**1,543**	**94**	**2,161**

Such information is very useful to API Consumers as it helps them understand how much adaptation they need to do in their applications to utilize the changes made in the framework. It also allows them to understand the kind of API changes taking place:- methods or fields moving around classes, renaming or changing method signatures, addition and deletion of elements like packages, classes, interfaces and methods (covered in our analysis). Analysing the extent of change is also particularly useful to API Producers, for frameworks like the Android platform, as millions of Android applications (a huge customer base) are dependent on the platform, which might have to be migrated or updated as a result of the API change. API Producers try to reduce the burden of reuse by developing migration tools and by providing solutions to API change problems that break compatibility with older applications.

4.3.2 Most Popular API Changes

In this experiment, we mine the dataset to identify the most popular API changes made after API Level 10. For this purpose, we mine the *specification difference sheet* for the APIs and make a note of the added/deprecated structural elements in each version. For each API level version, we mine those applications in the dataset whose version date is greater than the release date of the new API, by at least 1 day. So for API Level 19, we use 540 applications, while for API Level 11 we mine $1,114$ applications. For the sub-dataset that we create for each API version, we compute the most frequent ACUPs and identify the most used packages and most used classes.

We do not observe any effect of the release of APIs on frequent ACUPs. The list of frequent ACUPs almost remains the same as we work our way backwards from API Level 19. The only difference that we note is in the support count of the ACUPs. Moreover, we do not notice any major change in the most popular structural elements as well. However, we do observe some interesting results that we report below.

Response to Addition of Packages

Table 4.6: Top 6 Recently Introduced Packages

	Package Name	Num Apps Using Package	Num Files Importing Package	Introduced in Version
1	android.mtp	15	70	API 12
2	android.graphics.pdf	13	62	API 19
3	android.hardware.input	7	43	API 16
4	android.hardware.display	7	7	API 17
5	android.animation	6	34	API 11
6	android.print.pdf	5	10	API 19

Table 4.6 lists the 6 most popular packages that were introduced in the recent versions of the API. We discover that the package *android.mtp* that implements functionality which lets users interact directly with connected cameras and other devices using the PTP (Picture Transfer Protocol) subset of the MTP (Media Transfer Protocol) specification is the most used package among the packages that were introduced after API Level 11. It was introduced in API Level 12 and has been a part of 15 applications since (15 applications in about 30 months). The package *android.graphics.pdf* containing classes for manipulation of PDF content is the second most used package. It was introduced only in API Level 19, and is already being used 13 applications (13 applications in about 6 months). The *android.hardware* package provides support for hardware features, such as the camera and other sensors. Its usage in applications, clearly underlines that developers have started giving more importance to sensor-based advanced applications.

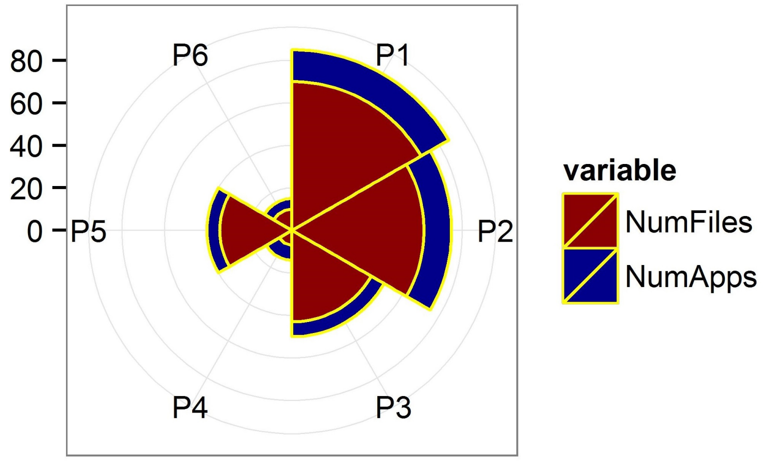


Figure 4.3: **Top 6 Packages Introduced in Recent History**:- *Rose Plot* displaying the most well received 6 packages introduced in different versions of Android API.

Figure 4.3 shows a *Rose plot* which is a circular histogram, that allows us to compare data across multiple variables. In this diagram, the 6 bins P1 - P6 are the top 6 packages introduced in recent times. The angle of each bin from the origin is determined by the number of elements we wish to compare. We compare the data on two frequencies denoting the strength of refer-

Table 4.7: Top 6 Recently Introduced Classes/Interfaces

	Class/Interface Name	**Num Apps Using Class / Interface**	**Num Files Importing Class / Interface**	**Introduced in Version**
1	android.support.v7.widget.SearchView. OnCloseListener	21	21	API 11
2	android.text.style.SuggestionSpan	18	28	API 14
3	android.provider.ContactsContract. - DataUsageFeedback	15	28	API 14
4	android.widget.ShareActionProvider	14	121	API 14
5	android.content.ComponentCallbacks2	12	53	API 14
6	android.speech.tts.TextToSpeech.EngineInfo	11	21	API 14

encing relationship between the API packages and the applications. The first is the number of applications using the package (red wedges), while the other is the number of files importing the package (blue wedges). The length of each bin reflects the number of elements that fall within a group, which ranges from 0 (at origin) to the highest number of elements in the bin.

Response to Addition of Classes/Interfaces

Table 4.7 lists the 6 most popular classes/interfaces that were introduced in the recent versions of the API. We find that the interface *android.support.v7.widget.SearchView.OnCloseListener*, that listens to a user's attempt to close a SearchView is used in 21 applications. The other 5 popular classes/interfaces were introduced in API Level 14. Thus, signifying the importance of API Level 14. *android.widget.ShareActionProvider* is responsible for creating views that enable data sharing, *android.text.style.SuggestionSpan* is used to display a pop-up dialog listing suggestion replacement for text, while *android.speech.tts.TextToSpeech.EngineInfo* is used to gather information about the text-to-speech engines installed on a device. All these elements

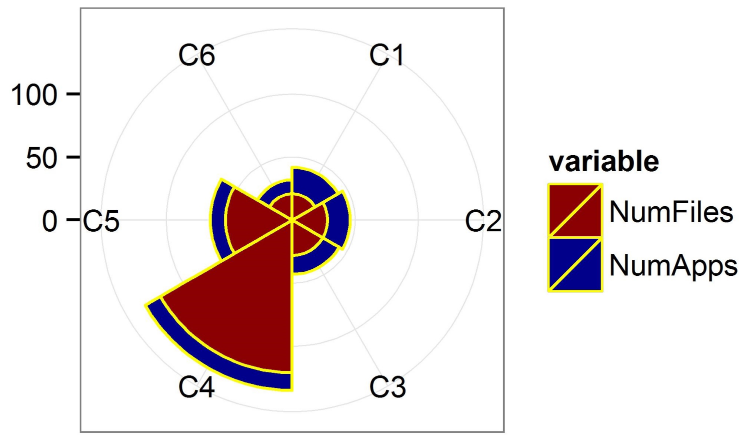


Figure 4.4: **Top 6 Classes Introduced in Recent History**:- *Rose Plot* displaying the most well received 6 classes/interfaces introduced in different versions of Android API.

allow a diverse set of functions and have been incorporated in more than 10 applications.

Figure 4.4 shows the 6 bins C1 - C6, which represent the top 6 classes/interfaces introduced in recent times. The red wedges represent the number of files importing the element, while the blue wedges represent the number of applications using the package. The class *android.widget.ShareActionProvider* has the largest bin because of the huge number of files it is imported in, even though it is fourth in terms of number of applications. Thus, a *Rose Plot* allows us to measure two variables together and draw insightful information.

However, the most interesting point that comes out from this experiment is that there is no relation between the most used recently introduced packages and the most used recently introduced classes/interfaces. This means that the packages that were recently introduced (Table 4.6) are generally used for different purposes and not for the same functionality, else the classes/interfaces provided by them would have made it to the list in Table 4.7.

Response to Deprecation of Classes

Another noteworthy result that we obtain from our analysis is that developers are either reluctant or slow to modify the code of their applications, once a class is removed from an API. We find 51 applications are still using *android.renderscript.Mesh* a class that was targeted at applications working in a high-performance setting across heterogeneous processors. This is a really important take away for API Producers, since, despite them deprecating the class an year ago, we still find it being used in 4.5% of the applications in our dataset.

4.4 Most Popular Methods and ACUPs

In this experiment, we do two things:- we first find out the the most invoked methods in the dataset and then we try to see if some relationship exists between the most invoked methods and the most popular ACUPs.

4.4.1 Most Popular Methods

Similar to the analysis described in Section 4.2, we consider the number of invocations of a method to be a measure of its popularity. In order to identify the most invoked methods, we traverse the AST of all the Java files in the dataset and count the number of invocations of each method. Again, we are not required to filter out the user-defined methods, since, the library methods out-number the user-defined methods quite easily. This time as well, we compute the *Significance Index* using Equation 4.1. The 'number of occurrences' component in the equation is replaced by the number of invocations for the method. Figure 4.5 is a bubble chart representing the top 20 methods. The top 5 bubbles in Figure 4.5, refer to the top 5 methods shown in Table 4.8.

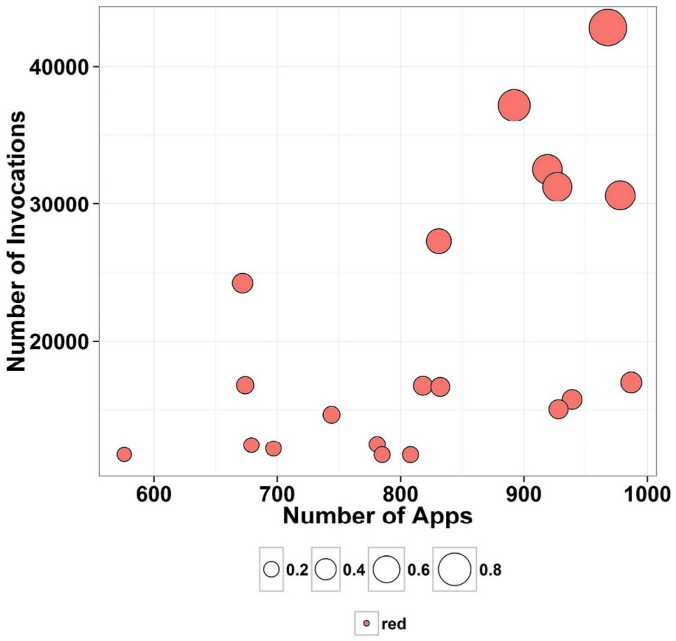


Figure 4.5: **Most Popular Methods**:- *Bubble Chart* displaying the top 20 methods in the dataset.

Figure 4.5 reveals some interesting points. From the top 20 methods, only 7 cross 24, 000 invocations, while the rest are way below 20, 000. Also, these topmost methods generally perform trivial operations. For instance:- *getString* is always used to retrieve the text content from a resource, while, *toString* is used to get the String representation of a Java object. It is important to mention that for this experiment, we do not differentiate between the *add* method of an ArrayList and the *add* method of a HashSet. Since, these methods perform a similar operation i.e. adding an element to a collection, we take them to be the same here. Same is the case for the other methods.

Table 4.8: Top 10 Methods in the Dataset

	Method Name	**Significance Index**	**Number of Invocations**	**App Frequency**
1	getString	0.97	42, 847	968
2	get	0.77	37, 198	892
3	toString	0.70	30, 627	978
4	add	0.70	32, 519	919
5	equals	0.68	31, 256	927
6	size	0.53	27, 289	831
7	findViewById	0.39	16, 964	987
8	getInstance	0.38	24, 231	672
9	setText	0.34	15, 737	939
10	show	0.32	15, 302	928

4.4.2 Relationship among Top Methods & Top ACUPs

We check for the presence of the top 20 methods in the top 10 ACUPs (sorted by *Significance Index*). Figure 4.6 is a bar chart where each bar represents one of the top 10 ACUPs. The colours of the bars represent the number of popular methods that are found in the particular ACUP. To our surprise, we find that the 10 top ACUPs mostly contain only 1 of the topmost methods - *findViewById*. This reflects that the topmost methods generally provide a functionality that is standalone, which to some extent corroborates the result represented in Table 4.8. Also, most of the methods of an ACUP may not be among the most popular methods, but when combined with other methods in an ACUP, they perform an indispensable function.

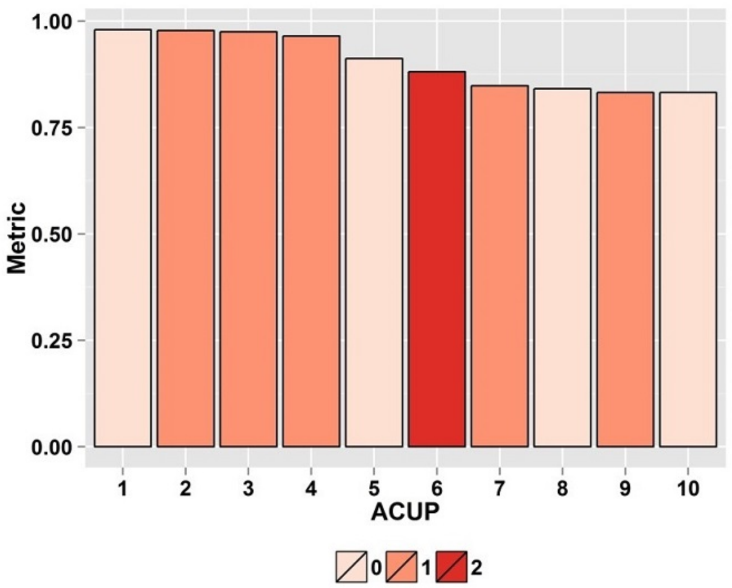


Figure 4.6: **Presence of Most Popular Methods in the Most Popular ACUPs**:- *Bar Chart* depicting the presence of most popular 20 methods in the top 10 ACUPs (sorted by *Significance Index*)

4.5 Category Focused Mining

The objective behind the following experiment is to analyse applications based on categories. Applications on *F-Droid* are categorised based on a general theme. This information can help API Producers to understand how their APIs are used across different categories of applications. This can help them fine tune their applications to cater to the needs of a particular kind of applications. API Consumers will also become more aware of the most popular elements in the category of their interest and can incorporate those elements into their applications.

4.5.1 ACUP Popularity across Categories

We mine ACUPs from the dataset based on their occurrences across various categories. We sort the ACUPs based on the number of categories. Table 4.9 shows the top 5 ACUPs with number

Table 4.9: Top 5 ACUPs Extracted from the Dataset based on Occurrences across 12 Categories

1	**getPaddingLeft getPaddingRight getPaddingTop getPaddingBottom**
	A *View* (rectangular area on the screen) takes into account its padding to measure its dimensions. The padding is expressed in pixels and the given ACUP is used to offset the content of the view by a specific amount of pixels
2	**query moveToFirst getString close**
	The given ACUP queries a given URI, moves the returned cursor to the first row, returns the value of the requested column as a String and closes the cursor, releasing all of its resources
3	**setMessage setPositiveButton getString show**
	The given ACUP displays a message on an *AlertDialog*, sets the listener to be invoked when the positive button is pressed, retrieves the text to be displayed in the positive button and shows the alert
4	**query moveToFirst getInt close**
	The given ACUP queries a given URI, moves the returned cursor to the first row, returns the value of the requested column as an Integer and closes the cursor, releasing all of its resources
5	**setMessage setPositiveButton getString setTitle**
	The given ACUP displays a message on an *AlertDialog*, sets the listener to be invoked when positive button is pressed, retrieves the text to be displayed in the positive button and sets the title of the dialog box

of categories as 9, 6, 6, 5 and 5 respectively. We also compute the total number occurrences of the ACUP in the dataset and the application frequency. The occurrence count and application frequency for the top 5 ACUPs of Table 4.9 are:- (408, 96), (805, 146), (481, 172), (318, 115) and (374, 128). The topmost ACUP in Table 4.9 appears in 9 out of 12 categories. It is based on size, padding and margin of views in an Android application.[6].

Table 4.9 also mentions the functionality or task performed by the extracted ACUPs. The topmost ACUP is related to *View* in Android. A *View* occupies a rectangular area on the screen and is responsible for drawing and event handling. To measure its dimensions, a view takes into account its padding, that is expressed in pixels for the left, right, top and bottom parts of the view. The padding can also be used to offset the content of the view by a specific amount of pixels. Such information is relevant to both the API Producers and Consumers as it informs them that the API on size, padding and margin is used quite frequently across almost all categories of applications and not just within one category.

Figure 4.7 displays a *Radar Chart* (also referred to as a *Spider Chart*) consisting of a sequence of 12 equi-angular spokes (or radii) with each spoke representing one of the 12 categories. We plot 5 data series (multivariate data) on the radar chart representing the ACUPs of Table 4.9. Each data series representing a vector of $w_{a,c}$ values is drawn as one complete circuit of the chart which means that a line is drawn connecting the data values for each spoke for a data series (a represents ACUP and c represents category). Weight $w_{a,c}$ is the value of a point on the *Radar Chart* and represents the distance from the center of the chart. The center represents the minimum value of $w_{a,c}$ which is 0 and the chart edge represents the maximum value of $w_{a,c}$ across all 5 series which is coming out to be 3.33 in our example (the axes are normalized). The weight $w_{a,c}$ is computed as shown in Equation 4.2.

[6]http://developer.android.com/reference/android/view/View.html

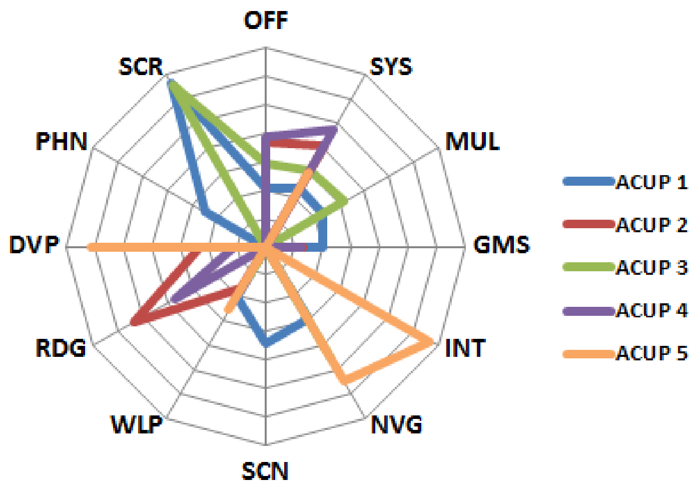


Figure 4.7: **Top 5 ACUPs across Categories**:- *Radar Chart* showing 5 data series representing top 5 ACUPs mined from the dataset (refer to Equation 4.2 and Table 4.9)

$$\left(\frac{SUP_c}{SUP}\right) * \left(\frac{NAP}{NAP_c}\right) \tag{4.2}$$

where SUP_c represents support of an ACUP in a particular category. SUP denotes the total support of an ACUP. NAP denotes the total number of applications and NAP_c represents the number of applications in a category. The maximum $w_{a,c}$ of 3.33 is for ACUP 1 and the category is *Security (SCR)*. It is calculated as [(4/96)*(1120/14)]. The $w_{a,c}$ for ACUP 1 across 12 categories are [1.030, 1.182, 1.137, 1.004, 0.000, 1.474, 1.707, 1.018, 0.000, 0.000, 1.240, 3.328] (starting from OFF till SCR). Experimental results reveal that ACUP 1 has $w_{a,c} > 0$ for 9 categories. The number of categories for which ACUP 2, 3, 4 and 5 have $w_{a,c} > 0$ are 6, 6, 5, 5 respectively.

4.5.2 Merged ACUPs across Categories

Table 4.10 reveals the top 7 Merged ACUPs in terms of application frequencies across all the categories. It also explains the functioning of each Merged ACUP in detail. The most surprising outcome of this experiment is that after merging ACUPs across different categories, *AlertDialog* again comes out to be the most used feature that is present in all 12 categories, while one of the ACUPs representing it, was individually present in only 5 categories. This means that different categories of applications use *AlertDialog* differently.

Figure 4.8 displays a *Heat-Map* which is a graphical representation of the ACUP application frequency data across categories where the individual values contained in the matrix are represented as colours. The more negative values are represented by relatively darker colours in comparison to positive values which are represented by lighter colours. The three dimensional data displayed by the *Heat-Map* in Figure 4.8 reveals the top 7 Merged ACUPs (Table 4.10) in

Table 4.10: Top 7 Merged ACUPs Extracted from the Dataset based on Occurrences across 12 Categories

1	**setMessage, getString, toString, setTitle, create, show**
	Displays a message on the *AlertDialog*, retrieves the text to display in the positive button, converts some integer to string for the purpose of displaying, sets the title of the dialog box, creates the alert and shows the alert
2	**getLayoutInflater, inflate, findViewById, setTag, setText**
	Instantiates a layout XML file into view objects, inflates a new view hierarchy from the specified XML resource, looks for a child View with the given id, sets a tag associated with this view and sets the text on a text view
3	**getMeasuredWidth, getPaddingBottom, getPaddingTop, getMeasuredHeight, getPaddingLeft, getPaddingRight**
	To measure its dimensions, a View takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific amount of pixels. These functions return the respective padding of the given view.
4	**setMessage, getResources, setTitle, setPositiveButton, show**
	Displays a message on the *AlertDialog*, returns Resource instance for application's package (The resource is generally used to get the message displayed in getMessage), sets the title of the dialog box, sets the action to be taken on pressing positive button, shows the alert
5	**parse, select, size, first, child, text, trim, equals, contains, attr, add, get, put**
	Parses HTML into a document, selects the relevant elements, calculates their size, chooses the first element, gets the given child of the node, retrieves and trims the text of the child and compares it with a given String, checks for a substring and performs some action based on the returned value, retrieves the attributes of the nodes, retrieves elements from java collections using get and add and adds some value to a hash-map
6	**iterator, hasNext, next, remove, size, isEmpty**
	Gets an iterator over a collection, returns true if item has more elements and moves to next element, removes last element returned by next, checks for the size of an arraylist, checks if some string in question is empty (using TextUtils.isEmpty)
7	**getActivity, findViewById, setOnClickListener, setText, getString**
	Returns the Activity for the given fragment, looks for a child view with the given id, registers a callback to be invoked when this view is clicked, sets text on a text view, retrieves the text to be displayed from some component of the view

terms of application frequencies across all the categories. The formula for computing the individual values contained in the matrix, represented as colours is [(Support of ACUP in particular Category/Total Support Of ACUP in all categories)*(Total number of Applications / Number of Applications in that Category)]. The API Producers and Consumers can use the *Heat-Map* as a handy tool (as an immediate visual summary of information) to analyse and visualize the multi-dimensional dataset on ACUP popularity across various categories. The highest value is 2.37 for ACUP 2 and category *Security* [(4/135) * (1120/14)] followed by value 2.24 for ACUP 2 and category *Reading* and then value 1.93 for ACUP 7 and category *Internet*.

4.5.3 Presence of Most Popular Packages, Classes across Different Categories

In this Section, we investigate the distribution of the top 10 packages and top 10 classes (from the entire dataset), across different categories. The idea behind this experiment is to see the contribution of each category in pushing the respective packages and classes to the top 10. Such information, helps in identifying the importance of a particular package or class for a particular kind of applications. This can help API Producers to target their product better and fine tune their product to match the expectations of the API Consumers. Moreover, if a library is crucial

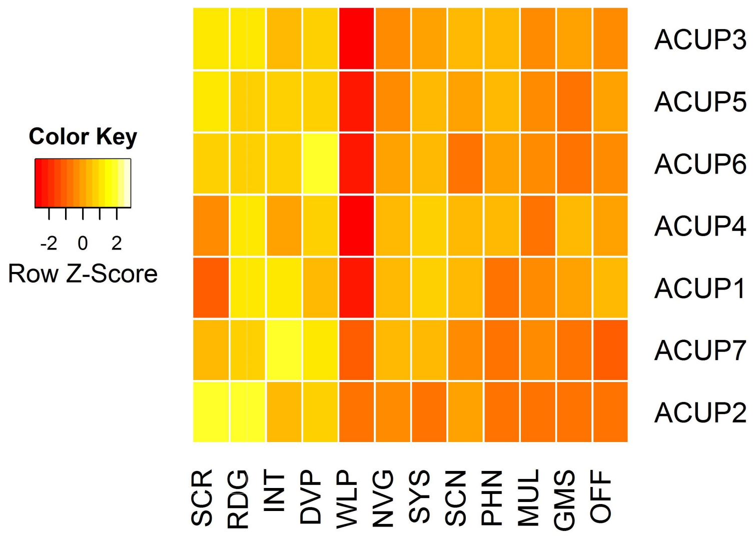


Figure 4.8: **Top 7 Merged ACUPs across Categories**:- *Heat-Map* showing the most popular ACUPs sorted by category

to all kinds of applications then it is possible that the library provides a commonly required crucial functionality and that there is sufficient available help and documentation regarding that library.

Distribution of Most Popular Packages across Categories

Table 4.11 lists the top 10 packages in our dataset and gives the *Significance Index* of the packages across the different categories. Higher the *Significance Index* for a particular category, higher is the rank of the package amongst the most referenced packages in the category. For instance, *java.util* has a high *Significance Index* for all categories and therefore, we can conclude that it is used widely across all categories and is important to every kind of application. However, *org.spongycastle.asn1* is used only in the categories *System* and *Internet*. Hence, we know that the *Bouncy Castle* library is private to two categories, where security is of vital importance. In no

Table 4.11: Top 10 Packages Across 12 Categories sorted by Significance Index

Package	**OFF**	**SYS**	**MUL**	**GMS**	**INT**	**NVG**	**SCN**	**WLP**	**RDG**	**DEV**	**PHN**	**SCR**
java.util	1.00	0.83	0.86	0.84	1.00	1.00	0.91	0.67	1.00	0.91	0.77	1.00
android.content	0.87	1.00	0.95	0.85	0.61	0.68	0.82	1.00	0.69	0.96	1.00	0.29
android.view	0.81	0.49	1.00	1.00	0.57	0.67	1.00	0.64	0.89	1.00	0.80	0.43
android.widget	0.77	0.44	0.82	0.78	0.50	0.66	0.96	0.41	0.74	0.86	0.66	0.31
java.io	0.50	0.57	0.58	0.59	0.83	0.48	0.57	0.38	0.63	0.52	0.42	0.74
android.os	0.49	0.46	0.60	0.48	0.37	0.42	0.53	0.53	0.47	0.65	0.53	0.24
android.app	0.42	0.32	0.39	0.37	0.24	0.30	0.36	0.33	0.30	0.38	0.34	0.10
android.util	0.25	0.19	0.35	0.29	0.21	0.23	0.34	0.38	0.29	0.39	0.29	0.12
android.graphics	0.15	0.14	0.35	0.43	0.12	0.29	0.28	0.75	0.33	0.24	0.10	0.05
org.spongycastle.asn1	0.00	0.33	0.00	0.00	0.49	0.00	0.00	0.00	0.00	0.00	0.00	0.00

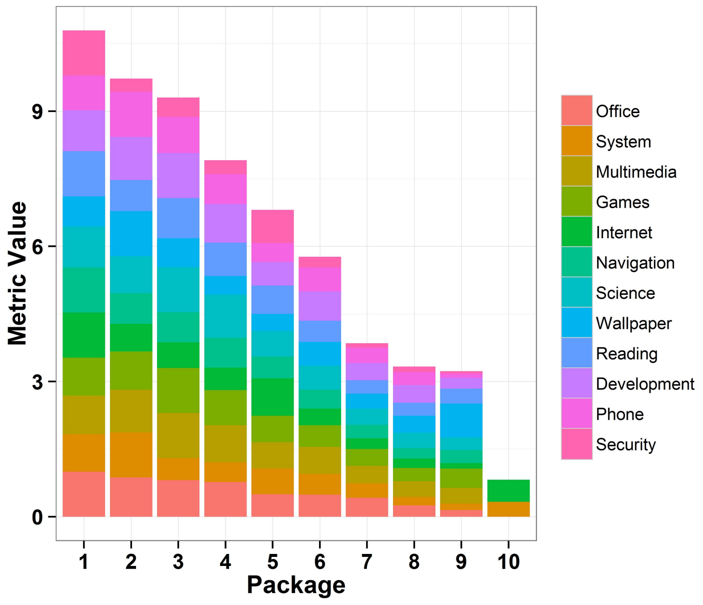


Figure 4.9: **Top 10 Packages sorted by Significance Index**:- *Stack Bar Chart* showing the top 10 packages in the dataset sorted by Significance Index

other category is security as big a concern as it is in *System* and *Internet*. However, surprisingly *Bouncy Castle* is not referenced in any application in the category *Security*. Although, this might be the result of our dataset having a very less number of *Security* applications.

Figure 4.9 is a *Stack Bar Chart* that shows the top 10 packages in the datset and the *Significance Index* across the different categories. The y-axis shows the cumulative *Significance Index*, while each number on the x-axis represents a package (1 being Package 1 in Table 4.11). Taller the bar, more significant the package is in the dataset. The different coloured stacks in a bar represent the *Significance Index* for a category. The broader the stack for a category, the more significant the package is in the category. The figure shows a steady decline in the cumulative *Significance Index*, as we move from *java.util* to *org.spongycastle.asn1*.

Table 4.12: Number of Occurrences of Top 10 Packages Across 12 Categories

Package	**OFF**	**SYS**	**MUL**	**GMS**	**INT**	**NVG**	**SCn**	**WLP**	**RDG**	**DEV**	**PHN**	**SCR**	**TOC**
java.util	11,494	12,343	5,091	3,371	16,090	5,540	2,700	533	2,784	1,488	1,295	2,381	65,110
android.content	9,996	14,828	5,588	3,385	9,801	3,752	2,427	796	1,919	1,568	1,686	679	56,425
android.view	9,322	7,274	5,897	4,005	9,225	3,730	2,967	506	2,489	1,638	1,356	1,024	49,433
android.widget	8,856	6,537	4,833	3,136	7,975	3,649	2,853	328	2,062	1,414	1,114	733	43,490
java.io	5,798	8,406	3,434	2,374	13,291	2,674	1,688	299	1,751	850	713	1,754	43,032
android.os	5,580	6,884	3,510	1,905	5,982	2,320	1,574	425	1,322	1,066	899	562	32,029
android.app	4,807	4,755	2,276	1,491	3,786	1,673	1,082	264	826	624	567	237	22,388
android.util	2,868	2,792	2,077	1,164	3,366	1,267	1,006	303	809	633	497	289	17,071
android.graphics	1,779	2,108	2,064	1,733	1,970	1,634	833	598	926	391	163	126	14,325
org.spongycastle.asn1	4	4,929	2	0	7,918	0	0	0	0	0	0	0	12,853

Table 4.12 shows the number of occurrences of the top 10 packages in the entire dataset and the distribution across the 12 categories. The data is plotted in Figure 4.10. The idea behind a second *Stack Bar Chart* is to demonstrate the impact of the *Significance Index*. As previously

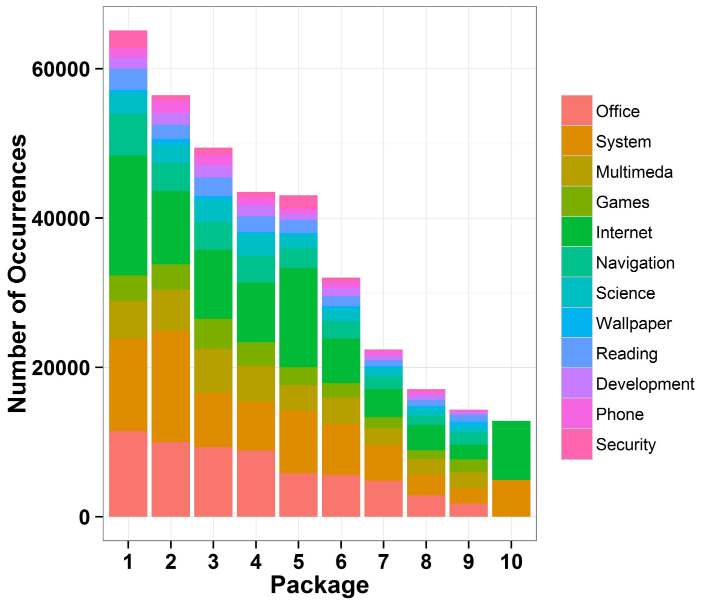


Figure 4.10: **Top 10 Packages sorted by Number of Imports**:- *Stack Bar Chart* showing the top 10 packages in the dataset sorted by Number of Imports/Occurrences

mentioned, the *Significance Index* combines the number of occurrences of a component (irrespective of the application), the number of applications using the component and the number of categories the component is found in. In short, it covers the all-round spread of a component and tries to provide a more normalised measure of popularity.

Take *android.widget* and *java.io* for instance. In terms of the number of imports, both of them are actually at par with each other. In terms of presence in the various categories, they are also at par. However, if we take into account the number of applications in which they are used *android.widget* has a lead. Hence, Figure 4.9 shows a taller bar for *android.widget*. We also observe that *java.io* is used more in applications belonging to the categories:- (*Internet* and *Security*), and it is generally imported multiple times within an application.

Distribution of Most Popular Classes across Categories

The top 10 classes are the top 10 classes in 6 out of the 12 categories. In those 6 categories, the top 10 classes are present, but in a slightly shuffled order, as can be seen from Table 4.13. However, *android.Content.Context* stays the top class in all categories except Internet and Security, where it is beaten by *java.io.exception*. In each of the remaining categories other than *Security* (*System*, *Internet*, *Wallpaper*, *Reading*, *Development*, *Security*), there are only 2 or 3 classes that are pushed out of the top 10. In case of *Security*, since we have less number of applications, an application with a very large source code (23 MB) induces false positives.

In the category *System*, *org.projectmaxs.shared.global.util.Log* $(1,743)$ and *android.content.SharedPreferences* $(1,741)$ come into the top 10 at the expense of *android.app.Activity* $(1,524)$ and *android.widget.TextView* $(1,435)$, which lie at the 11th and 12th position respectively. In case of

Table 4.13: Number of Occurrences of Top 10 Classes Across 12 Categories

Package	OFF	SYS	MUL	GMS	INT	NVG	SCI	WLP	RDG	DEV	PHN	SEC	TOC
android.content.Context	3,535	5,689	2,248	1,445	4,015	1,513	949	321	667	783	636	310	22,111
android.view.View	2,804	2,454	1,547	1,109	3,036	1,087	920	152	741	570	386	337	15,143
android.os.Bundle	2,558	2,682	1,219	9,38	2,521	927	81	151	612	518	381	275	13,633
android.content.Intent	2,044	3,267	1,142	663	2,111	865	592	155	432	420	418	152	12,261
java.io.IOException	1,501	2,359	960	543	4,210	612	372	67	460	269	192	464	12,009
java.util.List	1,785	2,008	1,009	640	2,456	1,176	500	130	600	385	261	352	11,302
java.util.ArrayList	1,868	1,906	1,054	765	2,426	1,017	623	116	556	287	234	303	11,155
android.util.Log	1,698	1,808	1,339	624	1866,	729	622	169	464	399	314	117	10,149
android.widget.TextView	1,469	1,435	706	550	1,383	583	486	53	309	285	198	154	7,611
android.app.Activity	1,540	1,524	764	579	1,217	561	381	66	316	221	201	102	7,472

the category *Internet*, *java.math.BigInteger* $(1, 595)$ and *java.io.InputStream* $(1, 467)$ jump into the top 10. *android.widget.TextView* $(1, 383)$ moves to number 11, while *android.app.Activity* $(1, 217)$ moves to number 13 with *android.view.ViewGroup* $(1, 294)$ separating the two. *android.view.ViewGroup* is the class for a special type of view that can contain other views.

Our results indicate that the category *Wallpaper* is unique. Its top 7 classes are from the list in Table 4.13. However, the last of those 10 - *android.widget.TextView* (53) is positioned at 19$^{\text{th}}$. In between we have the following classes:- *android.content.SharedPreferences* (110), *android.graphics.Canvas* (89), *android.graphics.Bitmap* (86), *android.util.AttributeSet*(83), *android.net.Uri* (82), *android.graphics.Paint* (70), *java.io.IOException* (67), *android.app.Activity* (66), *android.preference.Preference- Manager* (65), *android.graphics.Color* (65) and *android.os.-Handler* (64). This shows that any image or wallpaper oriented application finds image/graphics based classes more relevant than *java.io.- IOException*, *android.app.Activity* and *android.widget.-TextView*.

In the case of the category *Reading*, *java.io.File* (327) and *android.view.ViewGroup* (326) just manage to topple *android.app.Activity* (316) and *android.widget.TextView* (309) from the top 10. This is not very surprising, since most of the reading applications do some sort of a file read from the storage present on the device. In *Development* *android.view.ViewGroup* (237) and *android.view.LayoutInflater* (225) keep *android.app.Activity* (221) out of the top 10.

Chapter 5

Limitations and Future Work

Our dataset comprises of $1,120$ applications divided across 12 categories. However, we would like to point out that this dataset may or may not be an accurate representation of the distribution of applications on the Android Play Store. Although, Android is an open-source platform, Android applications need not be open-source. In fact, a large number of applications are not open-source. As a result, our dataset has not been chosen at random and may be biased. In our future work, we would try to work with a larger and more randomly chosen dataset.

In our work, we introduce the idea of Merged ACUPs and use the *Jaccard Index* to successfully merge a lot of ACUPs, with no false positives. However, the threshold of 0.75 does not allow us to merge all similar ACUPs. A more sophisticated approach should be able to help in improving the accuracy while keeping the false positives at nil. Doing so would allow for an even more accurate representation of a functionality. In our future work, we would try to successfully identify all methods that represent a functionality. This shall be very useful to new developers, since, they would get a good starting point.

Also, while dealing with popular methods, we largely combine API methods having the same name and performing similar functions irrespective of the classes they belong to. Hence, the results for the same can again be treated as methods performing a particular operation irrespective of the class they are defined in. In spite of this, we find that most of the popular ACUPs do not contain the most popular methods. In our future work, we would try to understand the reasons for the same.

In our study, while analysing API evolution, we have focused only on addition and removal of packages and classes in the Android API. We have not touched upon the changes in existing methods and classes. In our future work, we plan to do a detailed study on how such changes impact applications. We also intend to study the impact of addition of new methods and removal of old ones.

Chapter 6

Conclusion

In our work, we mine 1, 120 Android applications, belonging to 12 different categories in the quest to draw actionable insights from their source code. We use various advanced visualisations (*Bubble Chart*, *Radar Chart* etc.) to convey the important results. We uncover frequent ACUPs that reveal standard usages of the API. Experimental results reveal that the top 20 popular ACUPs are present in about 15% to 25% of the applications in the dataset. The most popular ACUP turns out to be on *AlertDialog*, a UI based component. In fact, a lot of top ACUPs are related to UI components. We then merge ACUPs which have methods in common using *Jaccard Index* and find that by merging we are able to accurately identify different methods related to a component. We observe that a popular ACUP on *AlertDialog* was present in 38% of the applications, but after merging similar ACUPs we find that the actual number of applications implementing *AlertDialog* is about 60%.

The most popular packages are *java.util* and *android.content*, classes are *android.content.Context* and *android.view.View*, and interfaces are *android.view.View.OnClickListerner* and *com.action - barsherlock.ActionBarSherlock.OnMenuItemSelectedListener*. Notably, we find that the *Bouncy Castle* cryptographic library is used more than the *java.security* library, while *ActionBarSherlock* is used extensively in Android applications. Both these APIs do not belong to the Android API, but are still used more than their Android API counterparts, showing that if the product is good developers are willing to adopt third party APIs.

We statistically see that the Android API is rapidly growing. The number of source code statements has increased by 25 times from Android Level 14 to 20 and the number of packages has grown by 60%. The most popular API changes are packages *android.mtp* and *android.graphics.pdf*. Our findings reveal that often the migration to a new version of the framework is slow, for example, about 5% of the applications in the dataset were found using a class (*android.renderscript.Mesh*) which was deprecated one year ago. The top 3 most popular methods *getString*, *get* and *toString* have more than 30, 000 invocations each. These methods generally perform a standalone operation and are not a part of any of the top 10 ACUPs. The only top 10 method seen in the Top 10 ACUPs is *findViewById*.

The ACUP that appears in 9 out of 12 categories is related to the size, padding and margins of a view in the UI. Unsurprisingly, the top most ACUPs across the 12 categories are related to the UI. Also, the Top 10 classes in the dataset remain the top 10 in 6 out of 12 categories. The category *Wallpaper* turns out to be the most unique category. Applications falling in this category have the least lines of code and the least number of transactions. Wallpaper applications also, make a lot more use of different classes like *android.content.SharedPreferences* and *android.graphics.Canvas*, as compared to the other categories.

Bibliography

[1] David CeArley and Carl Claunch. The top 10 strategic technology trends for 2013. *The Top 10 Strategic Technology Trends*, 2012.

[2] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[3] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM, 2005.

[4] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. Comparing approaches to mining source code for call-usage patterns. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 20–27. IEEE, 2007.

[5] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 54–57, Shanghai, China, 2006. ACM.

[6] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009.

[7] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. An approach to mining call-usage patterns with syntactic context. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 457–460, Atlanta, Georgia, USA, 2007. ACM.

[8] Chang Liu, En Ye, and Debra J. Richardson. Ltrules: an automated software library usage rule extraction tool. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 823–826, Shanghai, China, 2006. ACM.

[9] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. A graph-based approach to api usage adaptation. In *ACM Sigplan Notices*, volume 45, pages 302–321. ACM, 2010.

[10] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[11] Reid Holmes and Robert J. Walker. Informing eclipse api production and consumption. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, OOPSLA'07 eTX, pages 70–74, Montral, Canada, 2007. ACM.

[12] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol'09, pages 57–62, Amsterdam, The Netherlands, 2009. ACM.

[13] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. In *Testing–Practice and Research Techniques*, pages 173–180. Springer - Verlag, Berlin Heidelberg, 2010.

[14] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC'11, pages 1317–1324, TaiChung, Taiwan, 2011. ACM.

[15] Coen De Roover, Ralf Lammel, and Ekaterina Pek. Multi-dimensional exploration of api usage. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 152–161. IEEE, 2013.

[16] Roberto Minelli and Michele Lanza. Software analytics for mobile applications–insights &amp; lessons learned. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 144–153. IEEE, 2013.

[17] Wei Wang and Michael W Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 61–64. IEEE, 2013.

[18] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 477–487. ACM, 2013.

[19] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[20] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 398–416, London, UK, UK, 1999. Springer-Verlag.