A Model Based Approach for Android Design Patterns Detection

Diaeddin Rimawi
Master of Software Engineering
Birzeit University
Palestine
dmrimawi@gmail.com

Samer Zein
Master of Software Engineering
Birzeit University
Palestine
szain@birzeit.edu

***Abstract*— Design patterns in software development have shown great promise in improving software quality. Traditionally, software developers utilize a set of design patterns to foster reusability and better software design. Recently, mobile applications (apps) have become a mainstay of modern computing, as well as, a challenging domain for software engineers. This is because mobile apps now target more critical domains such as health, banking, m-payments, and even military to mention just a few. Android is a very popular mobile platform, and has managed to take over the majority of mobile market. It is true that there are comprehensive studies in the area of design patterns detection in several object-oriented languages such as Java, C# and C++. However, little studies target design patterns in Android apps. As a step toward helping to measure and explore the application of design patterns in Android apps, we introduce PatRoid, an automated framework for detecting design patterns. PatRoid is a model-based approach that is able to detect design patterns laying inside Android apps source code. The model is based on a graph isomorphism approach, where design patterns are divided into sub-patterns that can be aggregated to formulate design patterns. We have conducted a preliminary evaluation and the results show that PatRoid can detect all of the 23 GoF design patterns.**

Keywords—Design Patterns, Detection, Android Apps, Model-Based, PatRoid.

I. INTRODUCTION

Android has become one of the most popular mobile platforms. In the second quarter of 2018, Android extended its lead between the mobile operating systems market to reach 88%, leaving 11.9% for iOS and 0.1% for the other mobile OSs [1]. The turnout to use Android system is not limited by smartphone users, in fact, manufacturing, industry, and software development community have adopted Android platform as well [2]–[7]. As a result, Google Play Store (the official Android apps store) now leads mobile apps stores by having more than 3.1 million apps categorized into more than sixty categories [8], [9].

Mobile apps are not limited to games and entertainment domains. Nowadays, mobile apps are targeting critical domains such as medical, health monitoring, banking, education, traveling, etc., and there are many ongoing types of research studies that are employing Android apps as their subjects (e.g. [5], [10]–[14]).

Android development community brings together developers from different backgrounds (e.g. Desktop and Web) as well as novice developers. However, Android apps development is very different from desktop and web development, and even seasoned developers can find creating reliable Android apps with high quality to be intimidating [3].

On the other hand, Object-Oriented (O-O) Design Patterns are software solutions, which can be used repeatedly [15]. Many researches have proven that using design patterns enhance software quality especially software maintainability and evolution (e.g.[15]–[20]), with precaution when selecting the suitable design pattern(s) to avoid increasing the software complexity.

Studying design patterns in Android apps helps in understanding the quality of these apps. Few studies [20]–[22] talk about user interface (UI) design patterns for Android apps. However, to the best of our knowledge there are no studies that address object-oriented (O-O) design patterns for Android apps.

Despite the existence of many studies that discuss design pattern in web and desktop O-O languages, it is still largely neglected in mobile apps. Android apps are so different than traditional web and desktop paradigms. More specifically, Android employs activities for UI and each activity has its own life cycle. Further, Android supports dynamic inflation of XML UI at runtime.

As a step toward helping to measure and explore design patterns' application in Android apps, we introduce PatRoid. PatRoid is an automated framework that employs an extended model-based approach for detecting design patterns for Android apps source code. This approach was originally proposed by Dongjin et al. [23] for desktop O-O languages and tested with Java applications' source code.

In this research, we extended the approach by adding the ability to work with Android apps source code. It tackles the issues caused from the different structure of Android apps compared with other O-O languages, by analyzing the Android manifest XML files that contains all the information about the Android app activities, then it starts navigating in app source code, to collect and categories the Java code.

PatRoid has the ability to detect between classes associations as well as the dependencies on Android native libraries. Further, it groups Java classes among different packages based on the activities they belong to, which will enhance the tool evolution for new classification.

We have conducted a preliminary evaluation for PatRoid by injecting design patterns into an open-source Android app and then running it using our framework. The results show that PatRoid can detect all GoF design patterns.

The main contributions of PatRoid can be summarized as follows:

- An extended approach that can detect O-O design patterns for Android apps.
- Automated framework that implements extended approach.
- Built-in extensibility that facilitates the detection of new O-O design patters.

II. RELATED WORK

The current state-of-the-art appears to have very limited studies that discuss O-O design patterns in the area of mobile app development. In fact, when it comes to applying automated tools to detect design patterns in Android mobile

978-1-7281-3789-6/19/$31.00 ©2019 IEEE

apps, it appears that this research is the first to address this important topic.

There are many studies discussed the importance of studying O-O design patterns in general. Panca et al. [20] implement case study apps in three categories (learning, health, and survey) once using anti-pattern approach, and secondly, they used seven design patterns to re-implement the same apps. Then they compared both approaches against maintainability and modularity. According to their research, studying the effect of using object-oriented design patterns for Android apps implementation can reveal important insights about code maintenance and understandability. Further, using design pattern can improve the app modularity.

Several studies come to support this claim, Voka´cˇ et al. [16] repeated an old experiment to study design patterns effect on software maintainability. They found that there is no good or bad design pattern. If the design pattern used in a place where it characteristics match, it will have a positive effect on the software maintainability, otherwise it will only add more complexity. On the other hand, they came to conclusion where documented design patterns in any program can improve the quality and speed of its maintenance.

Secondly, a study by Ampatzoglou et al. [17] supports the results from Voka´cˇ study, their study focused on one of the maintainability four characteristics (analyzability, changeability, stability, and testability). Additionally, the study focuses on how GoF (Gang of Four) design patterns affect the instability of classes that are part of one or more design pattern. According to their results, four different factors affect the instability are, pattern type, class role, pattern coupling, and the app domain. On the other hand, an empirical study about design patterns effects on software maintenance and evolution by [24] shows that using design patterns can affect the software quality negatively. They concluded that design patterns are not always affecting the software development positively. Further, the authors argue that applying design patterns should be done carefully because this might actually affect maintenance and evolution negatively. However, their results cannot be generalized. This is because the author gathered data from only twenty software engineers. Thus, even if their experience in using design patterns in development and maintenance is verified, it still relative to their years of experience; the programming languages they used; the skills they have; and many other factors that all were omitted in the paper.

Due to the importance of studying design patterns, new research directions started to appear in the design patterns field such as studying design patterns detection [25]–[31], or taking a specific software related field and study the using of design patterns on software quality in that field [18], [19].

For instance, Oruc et al. [28] create new tool (DesPaD[1]), which represents the source code as a high-level model graph to extract and visualize design pattern from it, and test their tool against four different source code, then compare their results with related work.

On the other hand, an empirical experiment done in [18], where they compared the design pattern implementation with its alternative implementation against energy consumption. They were not the first to study this topic; however, as they claim they were different from their related work by the number of design patterns used, non-trivial systems, instances, methods, and a number of parameters. Additionally, they were different in the measurement and investigation levels. In their work, they prove that using design patterns will produce better or similar energy consumption than alternative solutions.

In addition, in a study by [19] a survey on different design patterns has been done to discuss their impact on data mining apps. In this study, three-layered architecture components were analyzed to expose the relationship between data mining systems and design patterns, and finally prove that using design patterns the right circumstances will relatively improve the system quality.

In conclusion, a recent systematic mapping study of literature was done on design patterns state of the art, which covers wider design patterns than only GoF design patterns. The study shows that there is a lack of research in the field of studying of design patterns in mobile app. Further, the goal of systematic mapping studies is to categories the researches topics in the field of study and identify the research direction in the field. Their systematic mapping study shows that in all the literature related to design patterns, only one study talks about design patterns in mobile app which is done by Nilsson [21] and Wesson et al. [22]

However, these studies are related to the user interface (UI) design patterns such as Google's Material Design Patterns Library, and not on object-oriented design patterns like Gang of Four (GoF).

Under the same context of extracting design patterns from source code lays the static analysis methods. However, in Android apps source code analysis, there is no research handled O-O design patterns detection. On the contrary, there are many kinds of research that applied static analysis methods to identify Android apps flaws or to solve some flaws [32]–[42]. Other researchers used static code analysis in to address different dimensions of Android apps behavior, for example Zein et al. [3] performed an empirical study to solve mobile resources handling during the application lifecycle. They compared their tool with related work based on the ability to catch positive and negative mobile resources releasing, and on the tool performance. Their research appears to be the first research, which addresses Android apps source code quality using static analysis method to ensure that Android developers correctly acquire and release system resources.

There are many studies that worked on automated design patterns detection models or tools. Al-Obeidallah et al. [43] published a survey that discusses design patterns detection approaches, this survey shows a comparison between thirty plus different approaches based on GoF design patterns coverage.

The comparison shows that only three approaches succeeded to cover the full GoF patterns, two of them are relatively old researches [44] and [45], and one of them is

[1] DesPaD (design pattern detector): a tool which creates a model graph with 12 relation types (implements, creates object of, extends, overrides, etc.) and only four kind of nodes (class, interface, abstract, and template), available online @ [https://github.com/muratoruc2006/DesPaD].

relatively new [23]. The third approach [23] shows an accuracy that escalate to reach 100% in some cases.

Dongjin et al. [23] present a new approach that provides full detection of all GoF design patterns. The approach detects design patterns by dividing them into easy to detect 15 sub-patterns represented through subgraphs, and to reduce the search space a prime number of joint classes and compositions were used as a base to merge these sub-patterns. Then it shows the structural feature model and signature templates for GoF design patterns to increase the accuracy. The proposed approach was tested over nine open source systems and showed a balanced high accuracy and recognize all 23 GoF design patterns.

In a recent study, an improved search ordered approach has been provided to enhance the design patterns detection time, by leading the search to start with the most representative classes and drop all irrelevant ones to reduce the search space [46]. The approach provided in this paper detect all GoF design patterns, and it was tested with 6 open source systems, two of these systems are considered large-scale projects.

Both researches [23] and [46] handles desktop software only, and was only tested with Java source code, which still leaving the gap open for Mobile and Android code. All other researches that propos design patterns detection approaches for example [47]–[53], all of these approaches are tested with systems implemented with Java or C++ languages. But until this research there are no studies discuss design patterns detection for Android apps.

III. Approuch

PatRoid framework is based on four main phases, (1) it starts by classifying Java code based on activities, then (2) it models the classes relationships, (3) extract sub-patterns instances, and finally (4) aggregates these sub-patterns to formulate the O-O design pattern. Fig. 1 shows the proposed approach model to detect design patterns from Android apps source code.

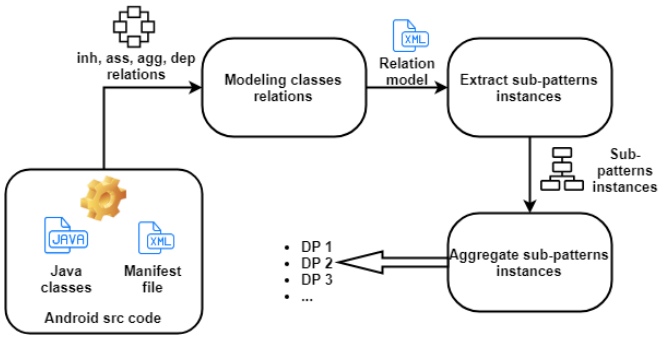

Fig. 1. Android Design Patterns Detection Workflow

A. Classifying Java Code Based on Activities

The things that distinguish Android apps from other desktop and web applications, that Android apps are consist from activities [3], [54]. These activities and other information about the app are laying inside the ManifestAndroid XML file [55].

ManifestAndroid file contains information about the app package name, components, services, activities, HW and SW requirements, etc. and all these information are essential to Android OS, Android build tools, and Google Play [55].

Since design patterns are laying in Android classes, in this study the ManifestAndroid file is used to capture all related activities, then categories all classes based on their activities and the type of each one. This process is considered as the starting point for the search processes to create the relations model, that will be use in further processes. The output from this process is list of all classes classified based on the Android activity they are part of.

B. Modelling Classes Relationships

The output from the first process along with all classes source code, will be analyzed to extract four kind of relationships among classes, the extraction process is sensitive to Android related keywords, so it will ignore methods, interfaces and inheritances that is used as Android APIs and not from the app logic:

1) Inheritance (inh): Which indicates that the relation between two classes is a parent and child relation.
   inh(ci, cj) = {class cj inherits class ci}.
2) Association (ass): Which indicates that there is an association relationship between two classes.
   ass(ci, cj) = {class ci has attribute of type class cj or a method that returns class cj object}
3) Aggregation (agg): Which indicates that there is an aggregation relationship between two classes, this relationship is a special kind of association relationship.
   agg(ci, cj) = {class ci is the whole and class cj is the part}
4) Depends (dep): Which indicates a depends relationship between two classes.
   dep(ci, cj) = {local object is defined in class ci of type class cj, object of type class cj is used as a method parameter in class ci, or calling a static method from class cj in class ci}

All four relations among with the class categories are forming the relations model, which is stored in XML format. This model consists of separated tag for each relation and another for the activities and related classes, that have been extracted in the firs process. Fig. 2 shows the general structure for the relations model and its main components.

```
<ADPD>
<depends><relation ci="c1" cj="c2" /></depends>
<aggregation><relation ci="c1" cj="C2" /></aggregation>
<association><relation ci="C1" cj="C2" /></association>
<inheritance><relation ci="C1" cj="C2" /></inheritance>
<manifest>
        <activity category="ACT_TYPE" name="ACT_NAME">
                <related_classes>
                                <class name="C1" />
                                <class name="C2" />
                </related_classes>
        </activity>
</manifest>
</ADPD>
```

Fig. 2. XML Relations Model Structure

C. Extract Sub-Patterns Instances

In this process a fifteen sub-pattern are extracted based on the relations model from the previous process. Each sub-pattern consists from one to three classes with one or two relations between them. The following TABLE I. shows the 15 sub-pattern along with each sub-pattern symbol, description, and the formal representation, where {c1, c2, c3} are classes names. Each sub-pattern aggregated with none, one or more other sub-pattern to formulate a design pattern.

TABLE I. SUB-PATTERNS DESCRIPTION

Sub-pattern	Sym.	Desc.	Formal Rep.
Aggregation Parent Inherited	AGPI	Which describes an inheritance relation where the parent is also aggregates with a third class.	AGPI(c1, c2, c3) = inh(c1, c2) AND agg(c3, c1)
Common Inheritance	CI	This sub-pattern talks about two sub classes shares the same parent.	CI(c1, c2, c3) = inh(c1, c2) AND inh(c1, c3)
Dependency Child Inheritance	DCI	Describes a relation between three classes where two of them has an inheritance relation and the child class depends on a third one.	DCI(c1, c2, c3) = inh(c1, c2) AND dep(c3, c2)
Dependency Parent Inherited	DPI	As dependency child inheritance this relation got an inheritance relation between two class, but the parent class has a depend relation with a third class.	DPI(c1, c2, c3) = inh(c1, c2) AND dep(c3, c1)
Indirect Inheritance Aggregation	IIAGG	Three inheritance relation where the first class is the parent of the second one and the second is the third class parent, and in addition to that there is an aggregation relation between the first and the third classes.	IIAGG(c1, c2, c3) = inh(c1, c2) AND inh(c2, c3) AND agg(c3, c1)
Inheritance Aggregation	IAGG	This sub-pattern describes a relation where the parent class aggregates the child class.	IAGG(c1, c2) = inh(c1, c2) AND agg(c2, c1)
Inheritance Association	IASS	Same as inheritance aggregation sub-pattern but the parent class associates the child class.	IASS(c1, c2) = inh(c1, c2) AND ass(c2, c1)
Inheritance Child Association	ICA	A relation between a parent and a child class where the child class is also associating a third class.	ICA(c1, c2, c3) = inh(c1, c2) AND ass(c2, c3)
Inheritance Child Dependency	ICD	A relation between a parent and a child class where the child class is also depending on a third class.	ICD(c1, c2, c3) = inh(c1, c2) AND dep(c2, c3)
Inheritance Parent Aggregation	IPAG	Parent class that aggregates a third class (not the child).	IPAG(c1, c2, c3) = inh(c1, c2) AND agg(c1, c3)
Inheritance Parent Association	IPAS	This one is a parent that associates a third class (no the child).	IPAS(c1, c2, c3) = inh(c1, c2) AND ass (c1, c3)
Inheritance Parent Dependency	IPD	Parent in this one depends on a third class (not the child).	IPD(c1, c2, c3) = inh(c1, c2) AND dep(c1, c3)
Multi-Level Inheritance	MLI	Three classes where the third class extends the second class and the second extends the first class.	MLI(c1, c2, c3) = inh(c1, c2) AND inh(c2, c3)
Self-Aggregation	SAGG	One class that aggregates itself.	SAGG(c1) = agg(c1, c1)
Self-Association	SASS	Also a one class that associate itself.	SASS(c1) = ass(c1, c1)

D. Design Patterns Detection

The final process in this model based approach is to combine sub-pattern instances with each other to identify design patterns. TABLE II. shows a description of what are the sub-patterns that formulate each design pattern, like Composite that can be either a CI with IAGG sub-patterns, CI with IIAGG sub-patterns, or finally SAGG only. Detailed UML diagrams is available on PatRoid repository on GitHub under the full evaluation results directory.

TABLE II. DESIGN PATTERNS SUB-PATTERNS COMBINATIONS

Design Pattern	Description
Adapter	This design pattern is a combination between ICA and NOT CI, which means that ICA with not inheritance exists between the parent and the third class.
Bridge	It is a combination between both CI and IPAG.
Composite	This design pattern can be one of three combinations, CI and IAGG, SAGG, or CI and IIAGG.
Proxy	It is either CI and ICA or CI and IASS.
Decorator	CI and IAGG or CI, IAGG and MLI.
Flyweight	It is a combination between AGPI and CI.
Façade	Combination of ICD sub-pattern.
Abstract Factory	A combination of ICD and CI and DCI.
Builder	IGPI and ICA combination.
Factory Method	This pattern is a combination between ICD and DCI.
Prototype	AGPI or a combination between CI and AGPI.
Singleton	SASS only.
Chain of Responsibility	A combination between SASS and CI.
Command	Both AGPI and ICA.
Interpreter	Each of CI, IAGG, and IPD sub-patterns.
Iterator	Either a combination between ICA and ICD, or ICA and DCI.
Mediator	The three sub-pattern CI, IPAS, and ICA.
Memento	AGPI and DPI.
Observer	It is a combination between AGPI and ICD.
State	A combination between AGPI and CI.
Strategy	This design pattern is a combination between AGPI and CI sub-patterns.
Template	It is CI only.
Visitor	Three sub-patterns AGPI, DPI, and ICD.

IV. PATROID STRUCTURE AND IMPLEMENTATION

This research proposes a new model based approach, which works with Android apps source code and detect design patterns used in these apps. During the first stage of this research the literature shows a lot of studies that discuss design patterns detection with desktop object oriented languages, and some of them works on designing tools that automate design patterns detection process. Based on that, this research plan was to take one of these tools and modify it to meet the new requirements to apply the model with Android source code.

However, we couldn't find any open source tool to work with in the sake of serving this research goals. So, we finally had to implement the tool from scratch to proceed with this research. The model was implemented using Python

language, which is a fast and strong open source language, easy to learn, has a wild responsive community, and runs on all platforms [56]. The model is named PatRoid (Android Design Patterns Detection), and can be found on GitHub[2] as an open source model, to be used in further studies.

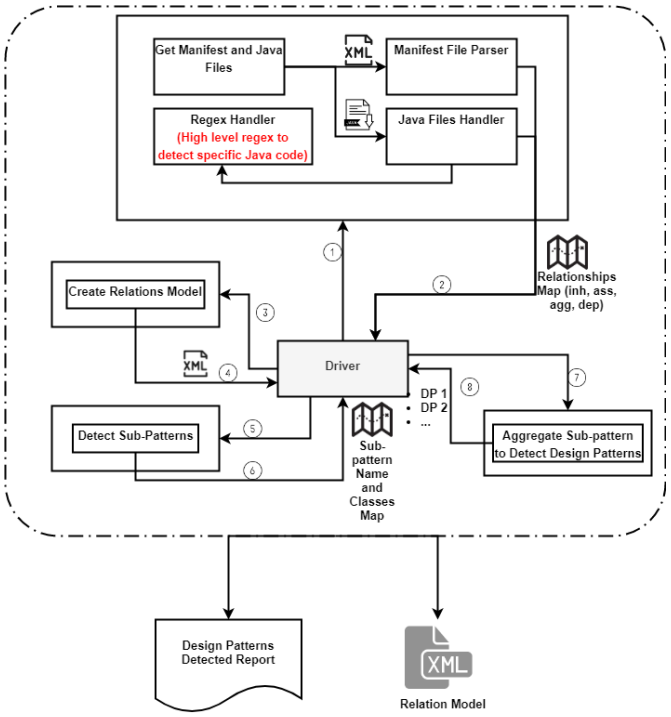

Fig. 3. Structure Diagram of PatRoid

As can be noticed in Fig. 3 a *Driver* class was implemented to manage the relationships and calls among the model overall components. It first starts by passing the Android project directory to *GetManiAndJava* class, which goes over the given project directory recursively filtering the files into two categories (1) Java files that contains all Java classes paths, and (2) ManifestAndroid XML file that contains the app activities and their information.

The list of Java files become an input to the *JavaFilesInfo* class and ManifestAndroid file becomes input to *ManifestParser* class. In *ManifestParser* class the app activity will be extracted and all Java files will be categories based on these activities. Then the *JavaFilesInfo* class will use the *RegexHandler* class to apply predefined regular expressions on the Java code, to extract the relationships among them. The following are some regexes that have been used in the tool:

1) Methods:
   r"((?:(?:public|private|protected|static|final|abstract|synchronized|volatile)\s+)*)\s*(\w+)\s+(\w+)\(([\w|\s|,|@]*)\)\s*({(?:{[^{}]*(?:{[^{}]*}|.)*?[^{}]*}|.)*?})[^;]"
2) Class Attributes:
   r"(\w*)\s*(\w+)\[{0,1}\]{0,1}\s+(\w+)\s*=.*?;"
3) Sub-class and Super-class:
   r"class\s+(\w+)(?:\s+extends\s+(\w+))*"
4) Static Methods Calls:
   r"(\w+)\.\w+\(.*\);"

The following sequence diagram appears in Fig. 4, shows how flow goes from and to the Driver class to prepare the relation dictionary.

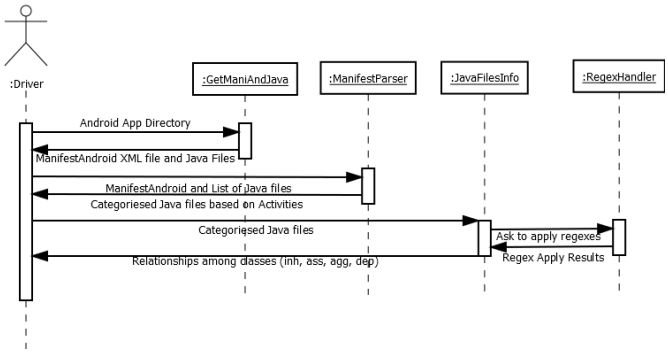

Fig. 4. Extract Relationships Among Java Classes Based on Activities Sequence Diagram

The second component is Modeling Classes Relationships, where the *Driver* class uses *CreateRelationsModel* class to analyze the relationships created by the previous components and formalize them into XML format as appears in **Error! Reference source not found.**.

Third component a series of fifteen methods inside the *SubPatterns* class in the Extract Sub-Patterns Instances component are called, in order to analyze the relations model and create the sub-patterns as the pre-step of design patterns detection. A list of tuples identifies the sub-pattern instance between one, two, and three classes, for example the ICD sub-pattern describes an instance child dependency between three classes, so the list of tuples will look as follows:

[(class1.1, class1.2, class1.3), (class2.1, class2.2, class2.3), ...]

An example of a sub-pattern with only two classes is IAGG that describes an inheritance aggregation relationship between two classes, the list of tuples will look as follows:

[(class1.1, class1.2), (class2.1, class2.2), ...]

On the other hand, SASS sub-pattern describes a self-association relationship, which means an association relation between the class and itself. So the list of tuples will look as follows:

[(class1), (class2), ...]

The fourth component is the Design Pattern Detection component, which is the final stage of the PatRoid model. In this component a class called *DetectDP* is implemented to aggregate the sub-patterns to formulate and identify the design patterns. *DetectDP* class is fully implemented to cover all design patterns.

Another helper component is implemented, which contains classes that are used in all the project, like I/O related methods, and exceptions handler, and it worth to mention that the *Logger* class contains four logging levels to ease debugging the model, INFO for printing information on screen, WARNING to draw the user attention for something specific, ERROR to indicate problem while running, and DEBUG for more details about what is going on and this

[2] PatRoid Repository on GitHub: https://github.com/dmrimawi/PatRoid

level is only printed to the log file, that can be found under the PatRoid model directory.

At the end, it is important to point at the high scalability of PatRoid, where it has the ability to add new design patterns easily. Only two places to modify, the first one is in the extract sub-patterns class, which needs the new sub-patterns if exist. Secondly is the detect design patterns class, which needs a new method to aggregate the sub-patterns to detect the new design pattern.

In Fig. 5 the general python code to add sub-patterns is shown, where there are two methods, the main method which calls the helper method and finally return a list of tuples in which the sub pattern consist of, the tuples can have one, two, or three classes.

```
def SUB_PATTERN(self):
    """
    :return: return list of tuples for classes that have
    SUB_PATTERN relation
    """
    # relation1_list and relation2_list are both represent
    # relations lists between classes these relations can
    # be read from the relations model
    logger.info("SUB_PATTERN (SUB_PATTERN description)")
    logger.info("Step1: Get relation1_list")
    relation1_list = self.get_node_by_name("relation1")
    logger.debug("relation1_list: %s" % relation1_list)
    logger.info("Step2: Get relation2_list")
    relation2_list = self.get_node_by_name("relation2")
    logger.debug("relation2_list: %s" % relation2_list)
    logger.info("Step3: Find SUB_PATTERN Tuples")
    return self.__SUB_PATTERN_helper(relation1_list, \
         relation2_list)

# Method args are <list(s) of relations formalize the
# sub-pattern>
def __SUB_PATTERN_helper(self, relation1_list, \
    relation2_list):
    """
    :param relation1_list: description of relation1_list
    :param relation2_list: description of relation2_list
    :return: list of tuples
    """
    # Check if the list(s) provided are None
    list_of_sub_pattern_relation = list()
    if relation1_list is None or relation2_list is None:
        logger.info("There are no SUB_PATTERN relations")
    else:
        # Place here the logic for checking the relations
        # provided to capture the sub-pattern, then save
        # the classes (one, two, or three classes) as a
        # tuple and finally append to
        # list_of_sub_pattern_relation
    # The following statment is used to remove duplication
    list_of_sub_pattern_relation = \
         list(dict.fromkeys(list_of_sub_pattern_relation))
    return list_of_sub_pattern_relation
```

Fig. 5. Sub-Pattern General Code

On the other hand, in Fig. 6 appears the general python code for adding new design pattern, where one method need to be added that takes the sub-pattern(s) it combines and return a list of the design pattern occurrences.

```
def detect_DESIGN_PATTERN(self, sub_pattern1, sub_pattern2,
…):
    """
    This method works on detecting DESIGN_PATTERN and
checks
    if this patterns exists or not
    :param sub_pattern1: description of the sub_pattern1
    :param sub_pattern2: description of the sub_pattern2
    :return: DP location
    """
    DESIGN_PATTERN_dp = list()
    logger.info("Checking for Iterator Design Pattern")
    logger.info("DESIGN_PATTERN Sub-patterns Description")
    # Place the logic behind aggregarting sub-patterns here
    # If the a match found then append to DESIGN_PATTERN_dp
    if len(DESIGN_PATTERN_dp):
        logger.info("DESIGN PATTERN has been detected:%s" \
                    % DESIGN_PATTERN_dp)
    else:
        logger.warning("Couldn't find any DESIGN \
                        "PATTERN_pattern in the code")
    return DESIGN_PATTERN_dp
```

Fig. 6. Design Pattern Aggregation Method General Code

V. Evaluation

In this research, a preliminary evaluation was done to evaluate PatRoid. The selected app under-test is Android SharingShortcuts Sample[3][57]. This app is a demo on how to show sharing options in a list of sharing candidate object. It has three activities (MainActivity, SendMessageActivity, and SelectContactActivity), it also contains other three classes (Contact, ContactViewBinder, SharingShortcutsManager).

In order to test PatRoid, firstly, each O-O design pattern skeleton was implemented and injected in the under-test app, by adding the java classes. Secondly, we link the main class with one of the AndroidManifest XML file activities. Finally, PatRoid was run against the app under-test, and then checking the results log for the detection of the injected design patterns. This process was repeatedly done with all 23 GoF design pattern, and PatRoid managed to detect them all.

For the full list of result logs of each of the design patterns, please refer to PatRoid GitHub page, since it can't fit inside this paper due to space constraints. The rest of this section shows a detailed example for one of the design patterns from implementation to the final results from PatRoid.

The example shows the flow of detecting the Composite design pattern. In terms of sub-patterns, the Composite design pattern can be one of three sub-patterns combinations:

1) SAGG: a self-aggregation, where the *Comp* class aggregates with itself. Fig. 7 shows the UML diagram of SAGG sub-pattern.

[3] Android SharingShortcuts Sample GitHub Link: https://github.com/googlesamples/android-SharingShortcuts

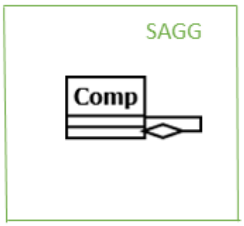

Fig. 7. Composite Design Pattern Sub-patterns Combination UML - 1

2) CI and IAGG: a combination between common inheritance and inheritance aggregation sub-patterns, where there are two classes (*ConcreteComp* and *Composite*) shares the same parent (*Comp*) and Comp aggregates with one of these classes (*Composite*). Fig. 8 shows the UML diagram for the CI and IAGG combination.

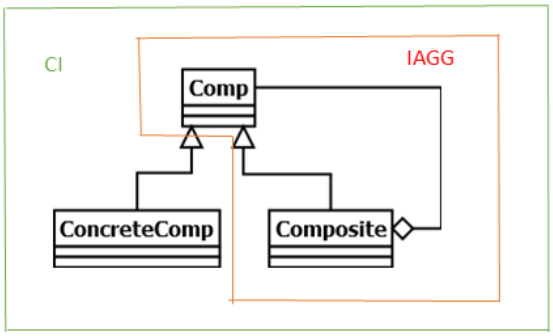

Fig. 8. Composite Design Pattern Sub-patterns Combination UML - 2

3) CI and IIAGG: a combination between a common inheretance and an indirict inheretance aggrigation, where two classes (*ConcreteComp* and *Composite*) shares tha same parent class (*Comp*) and in the same time the parent aggregates with a third class (*Composite1*) that inhiret from *Composite* class. Fig. 9 shows the UML diagram of the CI and IIAGG combination.

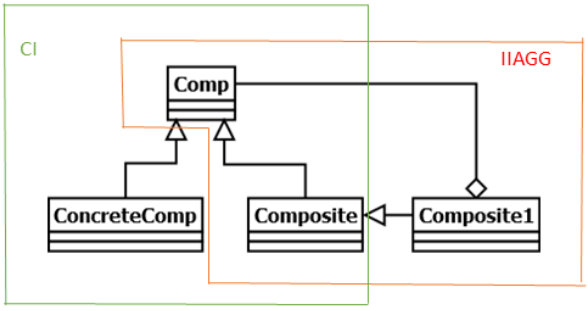

Fig. 9. Composite Design Pattern Sub-patterns Combination UML - 3

The second combination was chosen in this evaluation. Where three classes were implemented (*Comp, ConcreteComp,* and *Composite*). Both *ConcreteComp* and *Composite* classes extends the *Comp* class, and the *Comp* class aggregates the *Composite* class. The *Comp, Composite,* and *ConcreteComp* classes implementation is shown in Fig. 10, Fig. 11, and Fig. 12.

Then in the *MainActivity* class, we injected three dummy methods that return an object of each class. This will include the three injected classes in the analysis process, as they are now part of the main activity mentioned in the AndroidManifest XML file. The three dumy methods are shown in Fig. 13.

```
public class Comp {
    public static final
Composite composite = new
Composite();
    public static
Composite aggComposite()
    {
        return composite;
    }
}
```

Fig. 10. Comp class implementation

```
public class Composite
extends Comp {
}
```

Fig. 11. Composite class implementation

```
public class ConcreteComp
extends Comp{
}
```

Fig. 12. ConcreteComp class implementation

```
public class MainActivity
extends Activity {
…
   public Comp dpComp()
         {
            return new
Comp();
         }
   public ConcreteComp
dpConcreteComp()
         {
            return new
ConcreteComp();
         }
   public Composite
dpComposite()
         {
            return new
Composite();
         }
}
```

Fig. 13. Inject design pattern into MainActivity class

After injecting the composite design pattern in the app under-test app, we ran PatRoid against source code. During the run, a relation model was generated, showing all the relations among classes. Fig. 14 shows part of the relation model, where we can notice that the three classes are part of the MainActivity classes (hightlighted in yellow).

The green highlight appears in in both aggregation and inheritance relation. The aggregation relation shows that the *Comp (cj)* aggregates the *Composite (ci)*, while in inheritance relation, appears two relations (i) *Composite (ci)* inherits from *Comp (cj)* and (ii) *ConcreteComp (ci)* inherits from *Comp (cj)*, and as it noticable that cj in both inheritance relation is *Comp*, thus this relation is a Common Inheritance (CI) sub-pattern, and the parent class (*Comp*) aggregates one of the children classes (*Composite*) which mean that there is Inheritance Aggregation (IAGG) sub-pattern.

Hence, a composite design pattern is detected in the *Comp, ConcreteComp*, and *Composite* classes. In addtion, the pink hightlight shows another aggregation relation between the class *Contact* and itself, which means that there is a Self Aggregation (SAGG) sub-pattern, a.k.a. another combination of the composite design pattern. This

implementation is originally implemented in the app under-test.

```
<ADPD>
<depends>
        <relation ci="Contact" cj="ContactViewBinder"
/>
</depends>
<aggregation>
        <relation ci="Composite" cj="Comp" />
        <relation ci="Contact" cj="Contact" />
</aggregation>
<association>
        <relation ci="Composite" cj="Comp" />
        <relation ci="Contact" cj="Contact" />
        <relation ci="Comp" cj="MainActivity" />
        ...
</association>
<inheritance>
        <relation ci="Composite" cj="Comp" />
        <relation ci="ConcreteComp" cj="Comp" />
</inheritance>
<manifest>
        <activity category="LAUNCHER"
name="MainActivity">
                <related_classes>
                    <activity name="Composite" />
                    <activity name="Comp" />
                    <activity name="ConcreteComp" />
                        <activity
name="SharingShortcutsManager" />
                        <activity name="MainActivity"
/>
                </related_classes>
        </activity>
        <activity category="DEFAULT"
name="SendMessageActivity">
                <related_classes>
                        ...
                </related_classes>
        </activity>
        <activity category="None"
name="SelectContactActivity">
                <related_classes>
                        ...
                </related_classes>
        </activity>
</manifest>
</ADPD>
```

Fig. 14. Android SharingShortcut Sample Relation Model

PatRoid managed to successfully identify the two forms of composite design pattern, where it managed to detect the sub-patterns and successfully combined them to formulate the composite design pattern. Fig. 15 shows the related chunks from PatRoid results log. The yellow highlighted section shows the classes categorising based on the activities process. The green section shows the relations extraction among the classes. The blue section on the other hand, shows the sub-patterns detection. Finally, the orange section shows the search process to identify the design patterns, in this section the last line from the singleton design pattern appears to show that there is not singleton founded, then it continues to check the composite design pattern.

PatRoid starts by checking with the first combination SAGG, where it finds SAGG in *Contact* class. Then it moves further to check the combination if CI and IAGG, where if finds IAGG in *Composite* and *Comp*, and the CI in *Comp, Composite,* and *ConcreteComp*. Finally, it checks the final combination that formulate composite that is CI and IIAGG, where it doesn't find anything. So it finalizes the composite searching process by printing where it found this design patterns. After that it moves to search for next design pattern (e.g. Template).

```
2019-09-03 00:08:26: -I- Manifest file is:  android-
SharingShortcuts\Application\src\main\AndroidManifest.xml
2019-09-03 00:08:26: -I- Activities are: [{'category':
'LAUNCHER', 'classes': ['Composite', 'Comp', 'ConcreteComp'
'SharingShortcutsManager', 'MainActivity'], 'name':
'MainActivity'}, {'category': 'DEFAULT', 'classes':
['ConcreteComp', 'Composite', 'Comp', 'SendMessageActivity'
'MainActivity', 'ContactViewBinder', 'Contact',
'SelectContactActivity', 'SharingShortcutsManager'], 'name'
'SendMessageActivity'}, {'category': None, 'classes':
['ConcreteComp', 'Composite', 'Comp', 'SendMessageActivity'
'MainActivity', 'ContactViewBinder', 'Contact',
'SelectContactActivity', 'SharingShortcutsManager'], 'name'
'SelectContactActivity'}]
…
2019-09-03 00:08:26: -I- Inheritance: [{'Composite': 'Comp'
{'ConcreteComp': 'Comp'}]
2019-09-03 00:08:26: -I- Association relationships are
between: [{'Composite': 'Comp'}, {'Contact': 'Contact'},
{'Comp': 'MainActivity'}, {'ConcreteComp': 'MainActivity'},
{'Composite': 'MainActivity'}, {'Contact':
'SelectContactActivity'}, {'Contact': 'SendMessageActivity'
{'Contact': 'SharingShortcutsManager'}]
2019-09-03 00:08:26: -I- Aggregation relationships are
between: [{'Composite': 'Comp'}, {'Contact': 'Contact'}]
2019-09-03 00:08:26: -I- Depends relationships are between:
[{'Contact': 'ContactViewBinder'}]
…
2019-09-03 00:08:26: -I- CI (Common Inheritance)
2019-09-03 00:08:26: -I- Step1: Get all parent and child
classes
…
2019-09-03 00:08:26: -I- Step2: Find all children shares th
same parent
2019-09-03 00:08:26: -D- Found CI: (Comp, Composite,
ConcreteComp)
2019-09-03 00:08:26: -D- Found CI: (Comp, ConcreteComp,
Composite)
2019-09-03 00:08:26: -I- 2. CI relations: [('Comp',
'Composite', 'ConcreteComp')]
2019-09-03 00:08:26: -I- IAGG (Inheritance AGGregation)
2019-09-03 00:08:26: -I- Step1: Get all parent and child
classes
…
2019-09-03 00:08:26: -I- Step2: Get all classes with
aggregation relation
…
2019-09-03 00:08:26: -I- Step3: Find classes that have both
inheritance and aggregation
2019-09-03 00:08:26: -D- Found IAGG: (Composite, Comp)
2019-09-03 00:08:27: -I- 3. IAGG relations: [('Composite',
'Comp')]
…
2019-09-03 00:08:27: -I- SAGG (Self-Aggregation)
2019-09-03 00:08:27: -I- Step1: Get all Aggregation relatic
classes
…
2019-09-03 00:08:27: -D- Found SAGG: (Contact)
2019-09-03 00:08:27: -I- 7. SAGG relations: [('Contact',)]
…
2019-09-03 00:08:27: -W- Couldn't find any Singleton patter
in the code
2019-09-03 00:08:27: -I- Checking for Composite Design Patt
2019-09-03 00:08:27: -I- Composite can be founded by
combination of SAGG, CI & IAGG, CI & IIAGG
2019-09-03 00:08:27: -D- Step1: checking for SAGG
2019-09-03 00:08:27: -D- Composite DP in: {'SAGG':
('Contact',)}
2019-09-03 00:08:27: -D- Step2: checking for CI & IAGG
2019-09-03 00:08:27: -D- Composite DP in: {'IAGG':
('Composite', 'Comp'), 'CI': ('Comp', 'Composite',
'ConcreteComp')}
2019-09-03 00:08:27: -D- Step3: checking for CI & IIAGG
2019-09-03 00:08:27: -I- Composite design pattern has been
detected: [{'SAGG': ('Contact',)}, {'IAGG': ('Composite',
'Comp'), 'CI': ('Comp', 'Composite', 'ConcreteComp')}]
2019-09-03 00:08:27: -I- Checking for Template Design Patte
…
```

Fig. 15. Composite Design Pattern Detection Log

Of course, this is one example of the design patterns detection, however, PatRoid managed to detect all 23 GoF design patterns successfully.

VI. Future Work

In the same context of this research, additional enhancement can be done to the tool itself, such as enhancing the accuracy, and the performance. Accordingly, our next step will be enhancing the accuracy of this tool by not depending only on the relations among classes, but also on the methods calls orders among them.

Furthermore, this research opens a new research field of studying design patterns in Android apps. Many potential studies can be conducted based on this research. We plan to explore the design patterns applied in real world Android apps.

VII. Threats to Validity

The first threat to validity faced in this research is the lack of a proper benchmarks to test the implemented framework. Additionally, the fact that the main author had to inject the design patterns may have resulted in biased results. Accordingly, the framework has to be run against real-world Android apps.

VIII. Conclusion

In this study an extended model-based approach is proposed to detect O-O design patterns instances that are implemented inside Android app source code, which has a different structure than desktop and web object-oriented languages.

The model divides each design pattern into combination of sub-patterns, each sub-pattern consists of one or more relationship among Java classes that are categorized and gathered based on XML ManifestAndroid file.

This research reveals a new research area in design patterns detection approaches that is largely missing in Android apps. Exploring and searching for design patterns is very important for better understanding of software quality for Android app. Especially, with the expansion of Android apps subjects toward complex systems, like health, banking, etc.

References

[1] "Gartner Says Huawei Secured No. 2 Worldwide Smartphone Vendor Spot, Surpassing Apple in Second Quarter 2018," *Gartner*. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-08-28-gartner-says-huawei-secured-no-2-worldwide-smartphone-vendor-spot-surpassing-apple-in-second-quarter. [Accessed: 13-May-2019].

[2] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[3] S. Zein, N. Salleh, and J. Grundy, "Static analysis of android apps for lifecycle conformance," in *Information Technology (ICIT), 2017 8th International Conference on*, 2017, pp. 102–109.

[4] L. Li *et al.*, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[5] F. M. Kundi, A. Habib, A. Habib, and M. Z. Asghar, "Android-Based Health Care Management System," *International Journal of Computer Science and Information Security*, vol. 14, no. 7, p. 77, 2016.

[6] S. Zein, N. Salleh, and J. Grundy, "Mobile application testing in industrial contexts: an exploratory multiple case-study," in *International conference on intelligent software methodologies, tools, and techniques*, 2015, pp. 30–41.

[7] A. Asfour, S. Zain, N. Salleh, and J. Grundy, "Exploring Agile Mobile App Development in Industrial Contexts: A Qualitative Study," *International Journal of Technology in Education and Science*, vol. 3, no. 1, pp. 29–46, 2019.

[8] "App stores: number of apps in leading app stores 2018," *Statista*. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Accessed: 13-May-2019].

[9] "Android Apps on Google Play." [Online]. Available: https://play.google.com/store/apps?hl=en. [Accessed: 13-May-2019].

[10] M. Prakash, U. Gowshika, and T. Ravichandran, "A smart device integrated with an android for alerting a person's health condition: Internet of Things," *Indian Journal of Science and Technology*, vol. 9, no. 6, 2016.

[11] S. Papadakis, M. Kalogiannakis, and N. Zaranis, "Educational apps from the Android Google Play for Greek preschoolers: A systematic review," *Computers & Education*, vol. 116, pp. 139–160, 2018.

[12] P. Kirci and M. O. Kahraman, "Game based education with android mobile devices," in *Modeling, Simulation, and Applied Optimization (ICMSAO), 2015 6th International Conference on*, 2015, pp. 1–4.

[13] S. Bojjagani and V. N. Sastry, "Stamba: Security testing for Android mobile banking apps," in *Advances in Signal Processing and Intelligent Recognition Systems*, Springer, 2016, pp. 671–683.

[14] J. B. ek Jørgensen, B. Knudsen, L. Sloth, J. R. Vase, and H. B. erbak Christensen, "Variability Handling for Mobile Banking Apps on iOS and Android," in *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, 2016, pp. 283–286.

[15] B. B. Mayvan, A. Rasoolzadegan, and Z. G. Yazdi, "The state of the art on design patterns: A systematic mapping of the literature," *Journal of Systems and Software*, vol. 125, pp. 93–118, 2017.

[16] M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, pp. 149–195, 2004.

[17] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of GoF design patterns on stability: a case study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 781–802, 2015.

[18] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Investigating the effect of design patterns on energy consumption," *Journal of Software: Evolution and Process*, vol. 29, no. 2, 2017.

[19] N. P. Prabhakar, D. Rani, A. H. Narayanan, and M. V. Judy, "Analyzing the Impact of Software Design Patterns in Data Mining Application," in *Artificial Intelligence and Evolutionary Computations in Engineering Systems*, Springer, 2017, pp. 73–80.

[20] B. S. Panca, S. Mardiyanto, and B. Hendradjaya, "Evaluation of software design pattern on mobile application based service development related to the value of maintainability and modularity," in *Data and Software Engineering (ICoDSE), 2016 International Conference on*, 2016, pp. 1–5.

[21] E. G. Nilsson, "Design patterns for user interface for mobile applications," *Advances in engineering software*, vol. 40, no. 12, pp. 1318–1328, 2009.

[22] J. L. Wesson, N. L. O. Cowley, and C. E. Brooks, "Extending a mobile prototyping tool to support user interface design patterns and reusability," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, 2017, p. 39.

[23] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," *Journal of Systems and Software*, vol. 103, pp. 1–16, 2015.

[24] F. Khomh and Y.-G. Gueheneuce, "Do design patterns impact software quality positively?," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, 2008, pp. 274–278.

[25] M. Elaasar, "Detecting design patterns in models by utilizing transformation language," Apr-2014.

[26] K. Kumar and S. Jarzabek, "Detecting design similarity patterns using program execution traces," in *Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems,*

Programming, and Applications: Software for Humanity, 2014, pp. 55–56.

[27] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Detecting the Behavior of Design Patterns through Model Checking and Dynamic Analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 4, p. 13, 2018.

[28] M. Oruc, F. Akal, and H. Sever, "Detecting design patterns in object-oriented design models by using a graph mining approach," in *Software Engineering Research and Innovation (CONISOFT), 2016 4th International Conference in*, 2016, pp. 115–121.

[29] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa, "Detecting design patterns in object-oriented program source code by using metrics and machine learning," *Journal of Software Engineering and Applications*, vol. 7, no. 12, p. 983, 2014.

[30] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," *Applied Soft Computing*, vol. 26, pp. 357–367, 2015.

[31] M. Elaasar, L. C. Briand, and Y. Labiche, "VPML: an approach to detect design patterns of MOF-based modeling languages," *Software & Systems Modeling*, vol. 14, no. 2, pp. 735–764, 2015.

[32] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[33] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014, pp. 1–6.

[34] L. Li, A. Bartel, J. Klein, and Y. Le Traon, "Automatically exploiting potential component leaks in android applications," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, 2014, pp. 388–397.

[35] L. Li *et al.*, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 280–291.

[36] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229–240.

[37] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 77–88.

[38] Y. Z. X. Jiang and Z. Xuxian, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[39] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[40] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 78–89.

[41] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*, 2012, pp. 37–54.

[42] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 73–84.

[43] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," *International Journal of Software Engineering (IJSE)*, vol. 7, no. 3, 2016.

[44] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineering of design patterns," *Software & Systems Modeling*, vol. 4, no. 1, pp. 55–70, 2005.

[45] H. Kim and C. Boldyreff, "A method to recover design patterns using software product metrics," in *International Conference on Software Reuse*, 2000, pp. 318–335.

[46] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, "Efficiently detecting structural design pattern instances based on ordered sequences," *Journal of Systems and Software*, vol. 142, pp. 35–56, 2018.

[47] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Advances in Engineering Software*, vol. 41, no. 4, pp. 519–526, 2010.

[48] K. Stencel and P. Wegrzynowicz, "Detection of diverse design pattern variants," in *2008 15th Asia-Pacific Software Engineering Conference*, 2008, pp. 25–32.

[49] M. Vokác, "An efficient tool for recovering Design Patterns from C++ Code.," *Journal of Object Technology*, vol. 5, no. 1, pp. 139–157, 2006.

[50] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, 1999, pp. 226–235.

[51] M. von Detten and S. Becker, "Combining clustering and pattern detection for the reengineering of component-based software systems," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, 2011, pp. 23–32.

[52] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[53] M. Zanoni, "Data mining techniques for design pattern detection.," 2012.

[54] I. Musleha, S. Zainb, M. Nawahdahb, and N. Sallehc, "Automatic generation of Android SQLite database components," in *New Trends in Intelligent Software Methodologies, Tools and Techniques: Proceedings of the 17th International Conference SoMeT_18*, 2018, vol. 303, p. 3.

[55] "App Manifest Overview," *Android Developers*. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed: 20-May-2019].

[56] "Welcome to Python.org," *Python.org*. [Online]. Available: https://www.python.org/. [Accessed: 23-May-2019].

[57] "Android SharingShortcuts Sample," *GitHub*, 16-Aug-2019. [Online]. Available: https://github.com/googlesamples/android-SharingShortcuts. [Accessed: 02-Sep-2019].