Software Qual J
DOI 10.1007/s11219-017-9387-1

Mobile GUI testing

Inês Coimbra Morgado[1] · Ana C. R. Paiva[1]

Abstract This paper presents a tool (iMPAcT) that automates testing of mobile applications based on the presence of recurring behaviour, UI Patterns. It combines reverse engineering, pattern matching and testing. The reverse engineering process is responsible for crawling the application, i.e. analysing the state of the application and interacting with it by firing events. The pattern matching tries to identify the presence of UI patterns based on a catalogue of patterns. When a UI Pattern from the catalogue is detected, a test strategy is applied (testing). These test strategies are called UI Test Patterns. These three phases work in an iterative way: the patterns are identified and tested between firing of events, i.e. the process alternates between exploring the application and testing the UI Patterns. The process is dynamic and fully automatic not requiring any previous knowledge about the application under test. This paper presents the results of an experiment studying the reliability of the results obtained by iMPAcT. The experiment involved 25 applications found on Google Play Store and concludes that iMPAcT is successful in identifying failures in the tested patterns and that the degree of certainty of an identified failure being an actual failure is high.

Keywords Mobile testing · Android · Reverse engineering · Case study · UI patterns

1 Introduction

Software testing has always been a major part of software development in order to increase the quality of the software being produced. Nowadays, nearly every organisation provides a mobile application for their users. In fact, the number of applications available for the two main mobile operative systems (iOS and Android) have been increasing over the years having already surpassed one million available applications and fifty billion downloads three

✉ Ana C. R. Paiva
apaiva@fe.up.pt

[1] Faculdade de Engenharia da Universidade do Porto; INESC TEC, Porto, Portugal

years ago (Ingraham 2013). However, testing mobile applications is different than testing desktop or web applications due to mobile peculiarities such as their event-based nature, new development concepts like activities, new interaction gestures and limited memory (Muccini et al. 2012). Hence, existing testing techniques must be updated and new ones must be implemented. Testing may focus on different aspects: security, usability, user interface (UI), *etc.*. UI testing, for instance, is extremely important as the UI is what connects users to the application shaping the way they perceive it. Model-based testing (MBT) is a technique that may be used for UI testing but portraits two main issues: the need for an input model, whose manual construction is a time consuming failure prone process, and the combinatorial explosion of test cases, making test execution infeasible.

In this paper, we present a tool, iMPAcT, that tests Android mobile applications by dynamically reverse engineering the application's behaviour and testing recurring behaviour. Therefore, both problems faced by MBT techniques are surpassed: no prior knowledge about the application is required and the number of tests is limited to a feasible amount. These recurring behaviours are UI Patterns.

We focus on two different types of patterns: UI Patterns and UI Test Patterns. The purpose of an UI Pattern is to define when and how to detect the presence of the pattern. Regardless of its name, a UI Pattern may be associated either to the GUI of the application, such as the presence of the *Action Bar*, *Side Drawer* or *Login/Password*, or to system events, such as rotating the screen, receiving an incoming call or losing wireless connectivity. A Test Pattern is a generic test strategy that verifies if the corresponding UI Pattern is correctly implemented.

The approach presented in this paper automatically injects events on the application under test (AUT) in order to explore it and identify UI Patterns. When an UI Pattern is identified the corresponding test strategy is applied. In order to validate this approach a test case involving 25 applications present on Google Play Store belonging to five different categories compares the results obtained by iMPAcT with the ones obtained by manually testing each UI Pattern. Two UI Patterns are identified and tested: *Side Drawer* and *Tab*.

The remaining of this document is structured as follows. Section 2 presents work related to mobile testing. Section 3 presents the iMPAcT tool. Section 4 presents the catalogue of patterns used in this paper. Section 5 presents the case study. Section 6 presents the discussion of the results obtained. Finally, Section 7 presents the drawn conclusions.

2 Related work

When defining test strategies it is important to define what should be tested and how, i.e. define what is the correct behaviour of the application and what is its actual behaviour. Mobile testing approaches mostly focus on functional sustainability (the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions (ISO/IEC 2011)), reliability (degree to which a system, product or component performs specified functions under specified conditions for a specified period of time (ISO/IEC 2011)), security (degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization (ISO/IEC 2011)) and maintainability (degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers (ISO/IEC 2011)).

When automating the testing of mobile applications, it is necessary to have a model or a test oracle of the AUT. Otherwise, the testing process is prone to only finding crashes. However, the type of information the model/oracle contains differs according to the final goal: it may contain the correct behaviour of the AUT (e.g. Nguyen et al. 2012), or the life cycle an application should follow (e.g. Franke et al. 2012) or the correct filtering of intents (e.g. Avancini and Ceccato 2013) or even information on how to classify the purpose of the application according to its description (e.g. Gorla et al. 2014).

In order to obtain the required model/oracle several approaches use reverse engineering techniques to automatically extract information about the application. This can be done in runtime (dynamic reverse engineering), by analysing the source or byte code (static reverse engineering) or by doing both analyses (hybrid reverse engineering). Even though these techniques may extract information about how the AUT works, they lack information on how it should work and, thus, it restricts the aspects of the AUT that can be tested. In fact, as far as we know, no testing approach using reverse engineering techniques aims at testing the functional sustainability of a mobile application and most approaches focus on detecting crashes (Amalfitano et al. 2015; Zhu et al. 2015; Moran et al. 2016) or unwanted accesses (Batyuk et al. 2011; Dar and Parvez 2014; Ravitch et al. 2014). In order to provide a more thorough analysis when dynamically reverse engineering the AUT, instrumentation is generally used either on the AUT or on the emulator (e.g. the Dalvik VM). Instrumenting the AUT means the application needs to be modified and pre-processed before the testing process. On the other hand, instrumenting the system means that a device/normal emulator can not be used. In both cases, the runtime execution of the application is affected.

The approach presented in this document intends to reverse engineer an application in order to know what its actual behaviour is, while at the same time analysing if that behaviour is consistent with patterns defined from design and test guidelines. Thus, this work does not involve dealing with source code in any of its steps, being one of the few that present a completely dynamic instrumentation free approach, and presents a catalogue of patterns that work as a test oracle defining what the correct behaviour of the application should be. Moreover, the vast majority of mobile GUI testing works use GUI testing to test the AUT but does not focus specifically on testing the GUI of the AUT unlike our approach.

3 The iMPAcT tool

The main objective of the testing approach presented in this paper is the improvement of test automation of mobile applications. This approach is a combination of crawlers, model-based testing and patterns (Morgado 2017).

Patterns were originally defined in 1977 by Alexander et al. in the context of architecture as the "current best guess as to what arrangement of the physical environment will work to solve the problem presented" (Alexander et al. 1977). This concept can be generalised for all fields of studies as "an idea that has been useful in one practical context and will probably be useful in others" (Fowler 1997).

In this paper, we consider two types of patterns: UI Patterns and Test Patterns. The UI Patterns define how certain behaviour or layout may be identified and the Test Patterns define test strategies to test the corresponding UI Pattern. Although it is not specific to mobile applications, one of the most well-known UI Patterns is the *Login/Password*. In this case, the UI Pattern should detect when a *Login/Password* form is present on the screen (e.g. there must be a text box for the *username*, a text box that expects a password for the *password* and an *ok* button), and there would be a corresponding Test Pattern, which would verify if when a correctly *username/password* pair is provided the login is successful and it is unsuccessful otherwise (the correct pair would have to be provided by the user).

Both types of patterns (UI Patterns and Test Patterns) can be defined as a tuple *<Goal, P, V, A, C>*:

Goal is the ID of the pattern;
P is the pre-condition (boolean expression) defining the conditions in which the pattern should be applied;
V is a set of pairs [variable, value] relating input data with the variables involved. These must be instantiated by the user at the start;
A is the sequence of actions to perform;
C is the set of checks to perform (also known as the post-condition).

In other words a pattern can be formally defined as

$$Goal[configuration] : P -> A[V] -> C \tag{1}$$

, i.e. for each configuration of a goal $Goal[configuration]$, if the pre-condition (P) is verified, a sequence of actions (A) is executed with the corresponding input values (V). In the end, a set of checks (C) is performed.

In an UI Pattern, *P* defines when to verify if the pattern exists, *A* defines the actions to execute in order to verify if the pattern is present and *C* validates the presence of the UI Pattern. On the other hand, in a Test Pattern, *P* defines when the test should be executed, *A* defines the sequence of actions to execute in order to test if the corresponding UI Pattern is correctly implemented and *C* works as the test oracle and indicates if the test passes or fails, i.e. whether or not the UI Pattern is correctly implemented. For instance, the formal definition of the login/password Test Pattern (Authentication Test Pattern (Moreira et al. 2017)), could be:

- Test Goals: “Valid login” (LG_VAL) and “Invalid login” (LG_INV);
- Precondition: whenever an authentication UI pattern is present and not yet tested;
- Set of pairs [variable, value]: {[username, U], [password, P]}, where U and P are provided by the tester;
- Sequence of actions A: [provide username U; provide password P; press submit];
- Set of possible checks C: {“change to page X”, “pop-up error Y”, “same page”, “label with message K”}, where X is the target page, Y and K the message to be displayed.

The approach is an iterative process that is divided in three main components, *Patterns Catalogue*, *Tool* and *Platform Adaptor*, as depicted in the components diagram of Fig. 1.

The *Patterns Catalogue* contains the set of UI Patterns to be identified in the application and the corresponding Test Patterns. This catalogue can be used to test any mobile application. The user only needs to indicate which patterns should be identified and tested during a specific run.

The tool consists of two main components: the *Explorer* and the *Tester*.

The *Explorer* applies a dynamic reverse engineering process to automatically explore the AUT, i.e. it works like a crawler injecting events onto the device and receiving a response from it that is analysed.

The *Tester* consists of methods used to test the UI Patterns found on the application, i.e. methods used to apply test strategies that verify if the UI Patterns found during the exploration are correctly implemented.

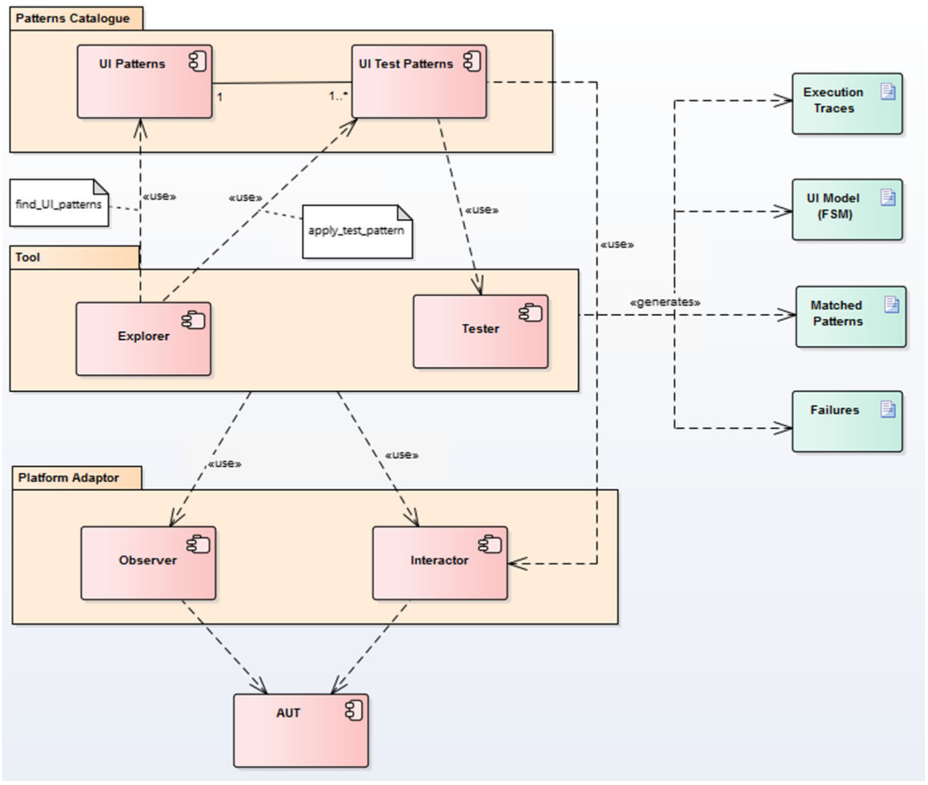


Fig. 1 Components diagram of the approach

Whenever the *Explorer* sends an event to the application, it verifies if there are any UI Patterns currently present in the application by a pattern matching process (*find_UI_patterns*). If one is detected, the *Explorer* accesses the corresponding Test Pattern (read from the catalogue) and applies it to the application under test (*apply_test_pattern*). In this phase, the *checks* of a Test Pattern use methods defined in the *Tester* and, afterwards, a report is produced.

Finally, the *Platform Adaptor* is responsible for bridging the *Tool* with the AUT, i.e. it depends on the operating system (OS) on which the AUT is being run. It consists of two main components: the *Observer* and the *Interactor*. The *Observer* obtains information on the state of the application and the *Interactor* injects events onto the device/application. Both *Observer* and *Interactor* are used by the two components of the *Tool* and the *Interactor* is also used when applying the actions of the Test Patterns.

At the end of the process, four main artefacts are produced:

- Execution log: the log of the exploration, i.e. the sequence of events fired on the application;
- UI Model: a finite state machine representing the different screens of the application and how to navigate through them;
- Matched Patterns: which of the UI Patterns were identified in the application;
- Failures: which patterns were not correctly implemented, i.e. which Test Patterns failed.

3.1 Exploration

The *Explorer* in Fig. 1 is responsible for the exploration of the AUT, which consists of four phases:

1. analyse the current state of the application;
2. identify the events that can be fired;
3. decide which event to fire;
4. fire the selected event.

The analysis of the current state of the application consists in using Google's UiAutomation framework to obtain a tree of the contents of the screen. Each of the nodes of the tree correspond either to a widget (a visual element of the screen) or to a structural element (a container). Each node has a set of properties associated to it (class, text, resource_id, clickable, editable, *etc.*). From these properties the events that can currently be executed on the screen are inferred (e.g. if an element is clickable, then the *click* event can be fired on it). Once the list of possible events is collected, an algorithm is run in order to decide which one. The not yet executed events are the first to be considered. However, if at a certain moment all events available in the current screen have already been executed, the events that lead to a different screen which still has not yet executed events are prioritised. This bit of information is collected when the event was previously executed and is updated whenever the event is executed. To avoid possible cycles, each event can only be executed a fixed amount of times. After all priorities are taken under consideration the final decision is random. iMPAcT has other exploration algorithms that can be used as described in Coimbra Morgado and Paiva (2016).

When no event can be fired the device's *back button* is pressed. When this action leads to the home screen of the device, the exploration stops.

3.2 Pattern matching

After each event is executed, it is necessary to verify which UI Patterns are present (*find_UI_patterns* from Fig. 1). This consists in applying the UI Patterns (assess the pre-conditions, execute the actions and validate the checks).

3.3 Testing

The testing consists in applying the Test Patterns (test strategies) corresponding to the UI Patterns detected in the Pattern Matching phase. At the end of the testing phase a report indicating whether or not the test passed is produced. If no UI Pattern is detected this phase is skipped.

More details on the implementation of iMPAcT can be found in Coimbra Morgado and Paiva (2015c).

4 Patterns catalogue

This paper considers two UI Patterns (more patterns can be found in Coimbra Morgado and Paiva 2015a, b, c): the *Side Drawer* and *Tab* patterns.

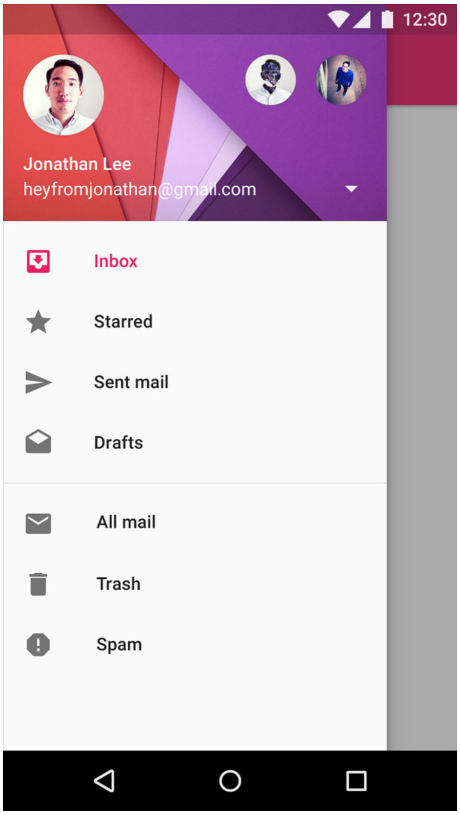


Fig. 2 Example of the *Side Drawer* UI Pattern (Android 2015a)

The *Side Drawer* UI Pattern detects the existence of a *Side Drawer* (see Fig. 2) and its Test Pattern verifies if it occupies the full height of the screen (as defined in the Android Design Guidelines (Android 2015b)).

The main challenge of implementing this pattern lays on how to identify the side drawer element. This was achieved by following an heuristic that identifies an element as the root of the side drawer when it has all of the following characteristics:

– it starts on the left edge of the screen;
– it does not take up the full width of the screen;
– it takes up at least half of the height of the screen;
– it is not the root of the screen
– its position on the screen overlaps other elements of the screen,
– the hierarchy of elements follow one of these aspects:[1]

1. $(FL \bigvee LV) \bigwedge OC \bigwedge parent$ is $V \bigwedge grandparent$ is FL;
2. $(FL \bigvee LL) \bigwedge \sim OC$;

[1]FL-Frame Layout, LV-Linear View, OC-only child, V-View, LL-Linear Layout

The Side Drawer Test Pattern can be formalized as:

– Goal: "Side Drawer occupies full height"
– P: {"Side drawer available && Test Pattern not applied to current activity"}
– V: {}
– A: [open side drawer, observation]
– C: {"covers the screen in full height"}

On the other hand, the *Tab* UI Pattern detects the presence of tabs (see Fig. 3) in the screen and the corresponding Test Patterns (TP) verify if:

1) there is only one set of tabs in the screen

– Goal: "Only one set of patterns"
– P: {"Tab present && TP not applied to current activity"}
– V: {}
– A: [observation]
– C: {"there is only one set of tabs at the same time"}

2) the tabs are located on the upper part of the screen

– Goal: "Tabs position"
– P: {"Tab present && TP not applied to current activity"}
– V: {}
– A: [observation]
– C: {"Tabs are on the upper part of the screen"}

and

3) horizontally swiping the screen changes the selected tab.

– Goal: "Horizontally scrolling the screen should change the selected tab"
– P: {"Tab present && TP not applied to current activity"}
– V: {}
– A: [observation, swipe screen horizontally, observation]
– C: {"the selected tab changed"}

There are two main technological challenges associated to this pattern: 1) identify the presence of tabs; 2) identify if the selected tab changed after horizontally swiping the screen.

In order to solve the first challenge, an element is considered the root of a set of tabs if it is either a *horizontalScrollView* or a *tabWidget*. In order to detect the presence of tabs in a screen, at least one element of one of these classes must be present on the screen.

Verifying if the tab changed after swiping the screen should be as simple as to read the *selected* property of each of the tabs present on the screen being only necessary to verify if the selected tab changed after swiping the screen. However, occasionally developers do not update this property after selecting a tab, resulting in zero or multiple tabs with the *selected* property set to *true*. When this is the case it is not possible to identify the selected tab by

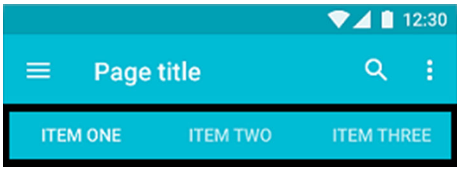


Fig. 3 Example of a set of tabs

analysing the *selected* property. Hence, the content of the screen below the set of tabs before swiping the screen is compared to the content of the same screen section after swiping the screen. If the content changed the Test Pattern passes. Otherwise, it fails.

4.1 Comparison of screens

Both *Side Drawer* and *Tab* patterns present an additional challenge as they are only tested if the screen in which they are presented has not yet been subject to scrutiny. Hence, after identifying a UI Pattern but before testing it, it is necessary to verify if we are in the presence of a new untested screen. However, Android provides no way of automatically identify a new screen. Therefore, an heuristic for comparing screens consisting of five main aspects was defined. An unique screen is referred to as an activity.

The first aspect to be considered is the package name. Each application has a package name associated to it that enables the device to identify the different applications that are installed. If two screens have elements with different package names then they do not belong to the same application and, thus, they do not correspond to the same activity.

Secondly, the dimensions of the root element of each screen are considered: a screen that occupies the whole screen (full screen) is considered different than one that occupies the whole screen except for the notification bar and both are considered different than one that occupies just part of the screen (pop-up).

Thirdly, the presence of a side drawer is detected. If only one of the screens contains a side drawer, then they are considered different.

Next, the widgets (i.e. interactable elements) to be compared are identified. If the screens present a side drawer only the widgets belonging to them are considered. On the other hand, if none of the screens contains a side drawer then all the widgets of the screen are considered except for the ones belonging to the action bar.

Finally, the comparison of the widgets from each screen takes place: considering the previously identified activity is represented by screen A and the new screen is represented by screen B then screen B represents a new activity if and only if none of the widgets considered in screen A are present on screen B (only the widgets selected in the previous step are considered).

The comparison of the widgets disregards the modifiable properties of the widgets. For example, a check-box is not considered different if in one of the screens it is checked and in the other it is unchecked.

Moreover, in order to avoid considering two screens different just because the position of the widgets changed (e.g. some elements have a contextual menu associated to them, i.e. a pop-up with actions such as *edit* or *delete* that act on the corresponding element; two elements in different positions may open the same menu in different positions as depicted in the example of Fig. 4), a final verification takes place in which both the modifiable properties and the position are disregarded. However, in this case they represent the same activity if and only if all the elements of screen B have an equivalent in screen A.

In summary, two screens represent the same activity if all the following aspects are verified:

- their elements have the same package name;
- their root elements have the same dimensions;
- either both or none contain a side drawer (if both screens contain a side drawer only the widgets belonging to it are considered);
- they have (the widgets that belong to an action bar are not considered):

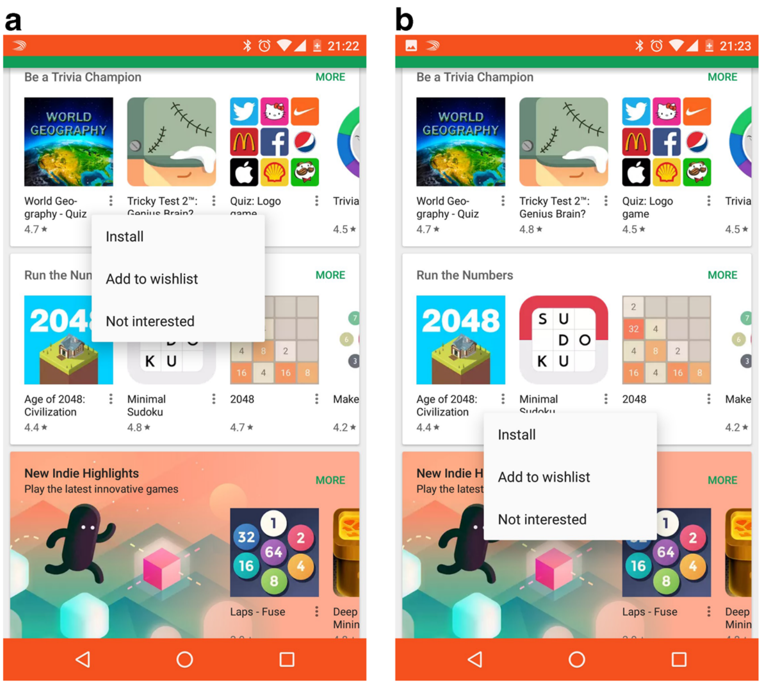


Fig. 4 The same contextual menu displayed in two different positions

- at least one widget in common disregarding their modifiable properties OR
- they have all widgets in common disregarding their modifiable properties and position

5 Case study

In order to evaluate the quality of the results obtained by iMPAcT, a research question was defined: "**Are the results obtained by iMPAcT reliable?**".

In order to answer this research question we designed a case study whose objective is to analyse the capacity of the tool for identifying the patterns and its capacity to accurately classify the identified patterns as correctly or incorrectly implemented.

5.1 Definition of the case study

This experiment must be able to evaluate the results presented by iMPAcT, i.e. it must undergo a manual validation process of both the UI Patterns detected and failures found.

The experiment consisted in:

1. select a set of applications in which to run iMPAcT;
2. manually identify the different UI Patterns in the application;
3. manually classify the identified UI Patterns as correctly or incorrectly implemented;
4. run iMPAcT for each of the UI Patterns on each application;
5. collect information on the UI Patterns identified;

6. collect the information on the failures found;
7. compare the data obtained by each method.

After comparing the data, the information would be divided into:

– UI Pattern detected or not detected;
– if detected, tested or not tested;
– if tested, failure identified or not;
– for both cases, analysis of the accuracy of the classification (correctly identifying a failure and correctly identifying a non-failure).

In order to help the comprehension of the results, this information was resumed in a tree alike the one in Fig. 5.

The first and second branches (*a* and *b*) illustrate the correctly identified and tested patterns, for which iMPAcT reports a failure. The third and fourth branches (*c* and *d*) illustrates the correctly identified and tested patterns for which iMPAcT claims there is no failure. Both *a* and *d* branches correspond to a correct classification unlike branches *b* and *c*.

Branch *e* indicates the patterns that were correctly identified but that were not tested because the pre-condition of the corresponding Test Pattern evaluated as *false*.

Finally, branch *f* illustrates when a pattern is incorrectly not identified.

In order to select the applications an initial selection of five different categories from the Google Play Store was performed: Books & References, Business, Comics, Finance and Health & Fitness. These categories were chosen in order to ensure a wide range of different characteristics amongst the applications.

Each of these categories was then analysed in order to select five applications, summing up to a total of twenty five applications. This selection was fulfilled in May 2016 and was based on the top applications presented by Google Play Store at the time for each of the aforementioned categories. For an application to be selected it would have to fulfil all of the following requirements:

– it is free;
– it is in English;
– it does not depend on other applications;
– it does not require a login to interact with the application;
– it does not require access to the device's camera;
– it has been downloaded at least five hundred thousand times;

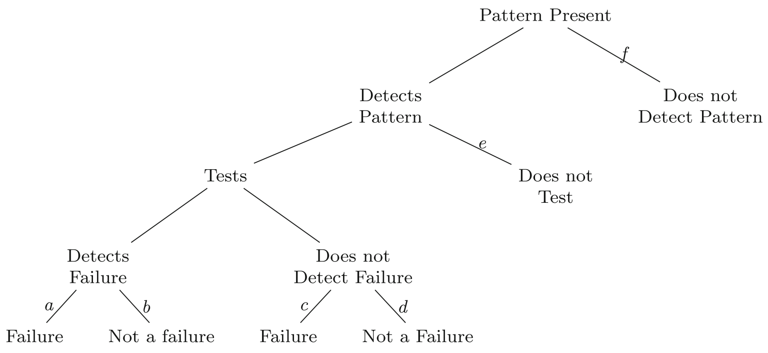


Fig. 5 Visual representation of the results

- it runs on a Nexus 5 with Android 6.0;
- it is a native application;
- it can be read by *uiautomatorviewer*;
- it does not require access to the device's contacts or documents;
- it presents *at least* one of the following characteristics:
    - it responds to the rotation of the device;
    - it presents a side drawer;
    - it presents at least a set of tabs.

Besides these requirements, a minimum of one hundred thousands votes was an initial requirement. However, in some categories this requirement had to be disregarded. Thus, this was considered a plus but not necessary.

The final set of selected applications is presented on Table 1.

Table 1 Final applications selection

App	Version
Books & References	
JW Library	1.7.1
Scribd - A world of Books	4.5
Bible	6.9.0
Google Play Books	3.9.37
Wikipedia	2.2.147-r-2016-06-06
Business	
File Commander - File Manager	3.8.14500
Find Work Offers - Trovit Jobs	4.2.4
BZ Reminder	1.5.1
Call Blocker Free - BlackList	5.2.22.00
Comics	
Marvel Comics	3.8.3.38302
Draw Cartoon 2	0.3.35
Draw Anime - Manga Tutorials	1.3.2
DC Comics	3.8.3.38303
Comics	3.9.1.39108
Finance	
Money Lover - Money Manager	android-3.3.9
Investing.com Shares & Forex	3.0.26
Monefy - Money Manager	1.7.3
MSN Money - Stock Quotes	1.1.0
Meta Trader 4	400.954
Health & Fitness	
Clue - Period Tracker	3.1.4
Pedometer	5.1.5
7 Minute Workout	1.29.64
Abs Workout	8.11.1
S Health	4.8.1.0013

The randomness associated to the decision of which event to fire affects the path taken during the exploration. For example, if an item in a list is removed prior to being explored the results obtained may be different than the ones obtained if the item is explored before being eliminated. On the other hand, the number of events identified and executed, and consequently the percentage of events executed, depend on the path taken during the exploration in each execution. Consequently, iMPAcT was run on each application three times for a maximum of twenty minutes each. This limit was imposed to ensure the experiment took a feasible amount of time while still being able to explore a considerable part of the application in each run.

In order to avoid interferences from the other patterns (as studied in a case study presented in Coimbra Morgado and Paiva 2016), iMPAcT tested each of them separately, i.e. it was run three times for each pattern in each application totalling six runs per application.

During these executions, we recorded information on when an UI Pattern was detected, when a Test Pattern was applied and when a failure was detected.

Finally, a manual inspection of each application was performed in order to verify if the detection, test application and results were correct.

5.2 Results of the case study

This experiment was performed using 25 applications in total:

- The Side Drawer UI Pattern was present in 18 applications and it was incorrectly implemented in 13 of them;
- The Tabs UI Pattern was present in 16 applications and it was incorrectly implemented in 5 of them.

Figures 6 and 7 present the final results obtained for side drawer and tab patterns, respectively, i.e. the percentage for each step considering all applications.

The major threat to validity of this experiment is the fact that the validation of the results was processed manually, being possibly subject to errors in the analysis. However, if another automatic process was used, the results would also depend on the reliability of that process.

5.3 Technical specifications

For this experiment, iMPAcT was run on applications installed on a LG Nexus 5, which presents the following main characteristics:

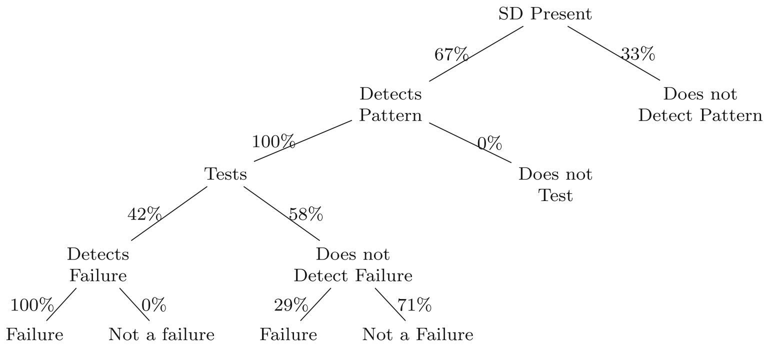


Fig. 6 Final results for the *Side Drawer* UI Pattern

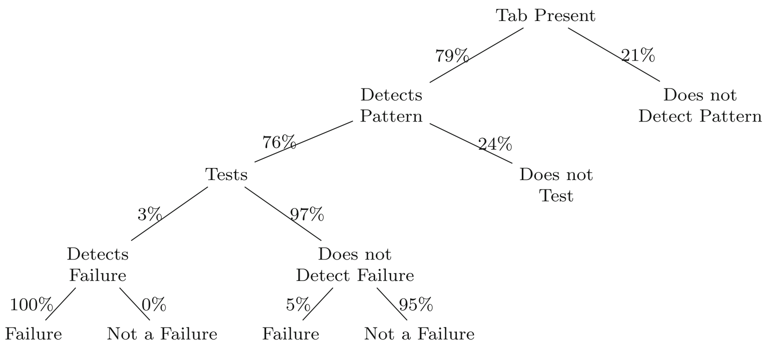


Fig. 7 Final results for the *Tab* UI Pattern

- Operating System: Android 6.0
- CPU: Quad-core 2.3 GHz Krait 400
- Chipset: Qualcomm MSM8974 Snapdragon 800
- GPU: Adreno 300
- RAM: 2GB

The characteristics of the computer used are not relevant because they do not affect neither the performance nor the results as iMPAcT is solely run on the device making use of none of the computer's computing capacity.

6 Discussion

The answer to the research question (Are the results presented by iMPAcT reliable?) can be divided in two sub-questions:

1. Can iMPAcT correctly identify the UI Patterns?
2. Are the results presented by iMPAcT reliable, i.e.:
   - if iMPAcT reports a failure what is the likelihood of it actually being a failure?
   - when it reports an absence of failures what is the likelihood of it not being a failure?

According to the results presented in Section 5, it is possible to conclude that:

- some *Side Drawers* were not correctly identified;
- all the identified *Side Drawers* were tested;
- all the identified failures for both the *Side Drawer* and *Tab* Patterns corresponded to real failures;
- the majority of the not-a-failure identifications were correct for both patterns (71% for *Side Drawer* and 95% for *Tab*).

After a more careful analysis, it is possible to state that:

- The misidentification of the *Side Drawers* was due to one of two situations:
  1. when the *Side Drawer* is opened it visually pushes the remaining elements and, thus, it is not positioned on top of them;
  2. even though the *Side Drawer* is drawn on top of other elements, the *UIAutomatorViewer* does not see the other elements not considering it to be on top of them.

- The cases in which an existing failure of a *Side Drawer* was not detected were due to a mismatch between the visual representation and the properties, i.e. even though the *Side Drawer* did not occupy the full height of the screen its properties indicated it did;
- The misidentification of the *Tab* Pattern was not detected due to one of two situations:
  1. The exploration stopped before reaching it;
  2. The heuristic did not correctly identify it as developers often do not coherently define them;
- The non-testing of some of the *Tab* Patterns was due to the misidentification of a new activity;
- The cases in which an existing failure in the *Tab* Pattern was not detected were due to the incorrect identification of the pattern: the failures that were not detected were the ones in which two sets of tabs were present on the same screen. However, in all these situations only one of the sets was correctly identified as a *Tab* Pattern (the others fell on the 21% that were not identified) and, thus, the failure was not detected.

These results show that it is important to improve the heuristics responsible for the identification of the UI Patterns. Nevertheless, when a Pattern is correctly identified it is correctly classified in the vast majority of the cases.

In summary, the answer to the research question is affirmative when it comes to identifying failures. When iMPAcT reports an absence of failure it is necessary to verify if the corresponding UI Pattern was correctly identified. If so, then the results are also reliable (71% confidence for *Side Drawer* and 95% confidence for *Tab*).

This work focuses only in Side Drawer and Tab UI Patterns. However there are more guidelines that the developers should know in order to correctly build mobile applications. These guidelines can be found in Android (2015b). They are divided in three big modules: Design, Develop and Distribute.

On the Develop module, the first part is focused on training, in which the guidelines aim to give Android developers the basics and the best practices they should follow while developing their applications. Moreover, these documents are also divided in smaller inner modules, where the modules for best practices are included. These modules are:

- Best Practices for Interaction and Engagement
- Best Practices for User Interface
- Best Practices for Background Jobs
- Best Practices for Performance
- Best Practices for Security and Privacy
- Best Practices Permissions and Identities

This work focuses mainly on “Best Practices for Interaction and Commitment”. To verify automatically all good practices, there is still a lot of work and do. However, this work shows the feasibility of the approach that can be extended in the future.

7 Conclusions

Considering the importance of mobile applications in our daily lives and the importance of ensuring their quality, this paper presented a tool that is capable of automatically testing recurring behaviour present on Android mobile applications.

Moreover, we present a case study in order to analyse the reliability of results presented by the tool. The quality of the results of the tests depends on the capacity of the tool to correctly detect a UI Pattern, which is affected by the lack of standards used in the implementation of these UI Patterns, resulting in some failures not being detected because the pattern is not identified. On the other hand, when iMPAcT detects a failure in one of these two patterns, the confidence of it actually being a failure is 100%. Nevertheless, an improvement of the identification of the UI Patterns should result in a high confidence when iMPAcT reports the absence of a failure given that in 71% or 95% of the cases (for *Side Drawer* and *Tab*, respectively), it is correct.

As future work we intend to improve the definition of the side drawer and tab UI Patterns and to improve the catalogue by adding more patterns. Moreover, the exploration can also be improved by defining new exploration algorithms. In addition, we aim to perform the experiment using different devices as we know the behaviour of the apps can be different.

References

Alexander, C.W., Ishikawa, S., Silverstein, M., & Jacobson, M. (1977). *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure)*, 1st edn. Oxford: Oxford University Press.

Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B., & Memon, A. (2015). MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software*, *32*(5), 53–59. https://doi.org/10.1109/MS.2014.55. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6786194.

Android, G. (2015a). Android navigation drawer. http://goo.gl/nnJOoj.

Android, G. (2015b). Up and running with material design.

Avancini, A., & Ceccato, M. (2013). Security testing of the communication among android applications. In *Automation of Software Test (AST), 2013 8th International Workshop on, IEEE Press, San Francisco, California, pp 57–63*. https://doi.org/10.1109/IWAST.2013.6595792.

Batyuk, L., Herpich, M., Camtepe, S.A., Raddatz, K., Schmidt, A.D., & Albayrak, S. (2011). Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on, IEEE, Fajardo, Puerto Rico, pp 66–72*. https://doi.org/10.1109/MALWARE.2011.6112328.

Coimbra Morgado, I., & Paiva, A.C.R. (2015a). Test patterns for android mobile applications. In *20th European Conference on Pattern Languages of Programs (Europlop 2015), Irsee, Germany*. http://dl.acm.org/citation.cfm?id=2855354.

Coimbra Morgado, I., & Paiva, A.C.R. (2015b). Testing approach for mobile applications through reverse engineering of UI patterns. In *Sixth International Workshop on Testing Techniques for Event BasED Software*.

Coimbra Morgado, I., & Paiva, A.C.R. (2015c). The iMPAcT Tool: testing UI patterns on mobile applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Lincoln, NE, USA*. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7372083.

Coimbra Morgado, I., & Paiva, A.C.R. (2016). Impact of execution modes on finding android failures. *Procedia Computer Science*, *83*, 284–291. https://doi.org/10.1016/j.procs.2016.04.127. http://www.sciencedirect.com/science/article/pii/S1877050916301508.

Dar, M.A., & Parvez, J. (2014). Enhancing security of android & IOS by implementing need-based security (NBS). In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), IEEE, Kanyakumari, India, pp 728–733*. https://doi.org/10.1109/ICCICCT.2014.6993055.

Fowler, M. (1997). *Analysis Patterns - Reusable Object Models*. Boston: Addison-Wesley.

Franke, D., Kowalewski, S., Weise, C., & Prakobkosol, N. (2012). Testing conformance of life cycle dependent properties of mobile applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, Montreal, QC, Canada, pp 241–250*. https://doi.org/10.1109/ICST.2012.104. http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6200126.

Gorla, A., Tavecchia, I., Gross, F., & Zeller, A. (2014). Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), ACM Press, Hyderabad, India, pp 1025–1035*.

Ingraham, N. (2013). Apple announces 1 million apps in the App Store, more than 1 billion songs played on iTunes radio. https://goo.gl/MYwnYo.

ISO/IEC (2011). ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — system and software quality models. Tech. rep. https://www.iso.org/standard/35733.html.

Moran, K., Linares-Vasquez, M., Bernal-Cardenas, C., Vendome, C., & Poshyvanyk, D. (2016). Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, Chicago, IL, USA, pp 33–44*. https://doi.org/10.1109/ICST.2016.34. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7515457.

Moreira, R.M.L.M., Paiva, A.C.R., Nabuco, M., & Memon, A. (2017). Pattern-based GUI testing: bridging the gap between design and quality assurance. Softw Test, Verif Reliab; 27(3). https://doi.org/10.1002/stvr.1629.

Morgado, I.C. (2017). *Automated pattern-based testing of mobile applications*. Porto: PhD thesis. Faculty of Engineering of the University of Porto.

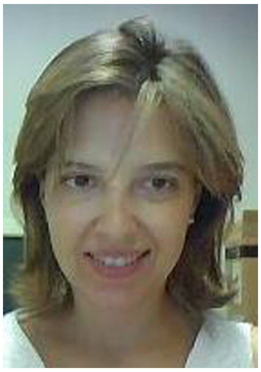
Muccini, H., di Francesco, A., & Esposito, P. (2012). Software testing of mobile applications: challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on, IEEE, Zurich, Switzerland, pp 29–35*. https://doi.org/10.1109/IWAST.2012.6228987.

Nguyen, C.D., Marchetto, A., & Tonella, P. (2012). Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), ACM Press, Minneapolis, MN, USA, pp 100–110*. https://doi.org/10.1145/2338965.2336765.

Ravitch, T., Creswick, E.R., Tomb, A., Foltzer, A., Elliott, T., & Casburn, L. (2014). Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW-4), ACM, New Orleans, LA, USA, pp 4:1–4:10*. https://doi.org/10.1145/2689702.2689705.

Zhu, H., Ye, X., Zhang, X., & Shen, K. (2015). A context-aware approach for dynamic GUI testing of android applications. In *2015 IEEE 39th annual computer software and applications conference (COMPSAC), IEEE, TaiChung, Taiwan, vol 2, pp 248–253*. https://doi.org/10.1109/COMPSAC.2015.77.

Inês Coimbra Morgado is an Informatics Engineering PhD student under the supervision of professor Ana C. R. Paiva and an Assistant Professor at the Department of Informatics Engineering at Faculty of Engineering of University of Porto. She teaches subjects like Logic Programming, Information Systems and Web Languages and Technologies. Her main research interest is testing automation but she is also interested in Math, Logic and Programming.



Ana C. R. Paiva is Assistant Professor at the Informatics Engineering Department of the Faculty of Engineering of University of Porto (FEUP) where she works since 1999. She is a researcher at INESC TEC in the Software Engineering area and member of the Software Engineering Group which gathers researchers and post graduate students with common interests in software engineering. She teaches subjects like Software Testing, Formal Methods and Software Engineering, among others. She has a PhD in Electrical and Computer Engineering from FEUP with a thesis titled "Automated Specification Based Testing of Graphical User Interfaces". Her expertise is on the implementation and automation of the model based testing process. She has been developing research work in collaboration with Foundation of Software Engineering research group within Microsoft Research where she had the opportunity to extend Microsoft's model-based testing tool, Spec Explorer, for GUI testing. She is PI of a National Science Foundation funded project on Pattern-Based GUI Testing (PBGT). She is a member of the PSTQB (Portuguese Software Testing Qualification Board) board general assembly, member of TBok, Glossary, and Examination Working Groups of the ISTQB (International Software Testing Qualification Board), and member of the Council of the Department of Informatics Engineering of FEUP.