Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki,
Informatyki i Automatyki

Praca dyplomowa magisterska

Testowalność aplikacji mobilnych na platformę Android

Rafał Sowiak
Nr albumu: 199564

Opiekun pracy: prof. dr hab. inż. Andrzej Napieralski
Dodatkowy opiekun: prof. mgr inż. Michał Włodarczyk

Łódź, 26.02.2016

Spis treści

1 Wprowadzenie **2**

2 Android jako system operacyjny **3**

3 Testowalność oprogramowania **5**

4 Opis problemu **7**

5 Propozycja rozwiązania **11**

6 Opis przykładowej aplikacji **14**

7 Wnioski **15**

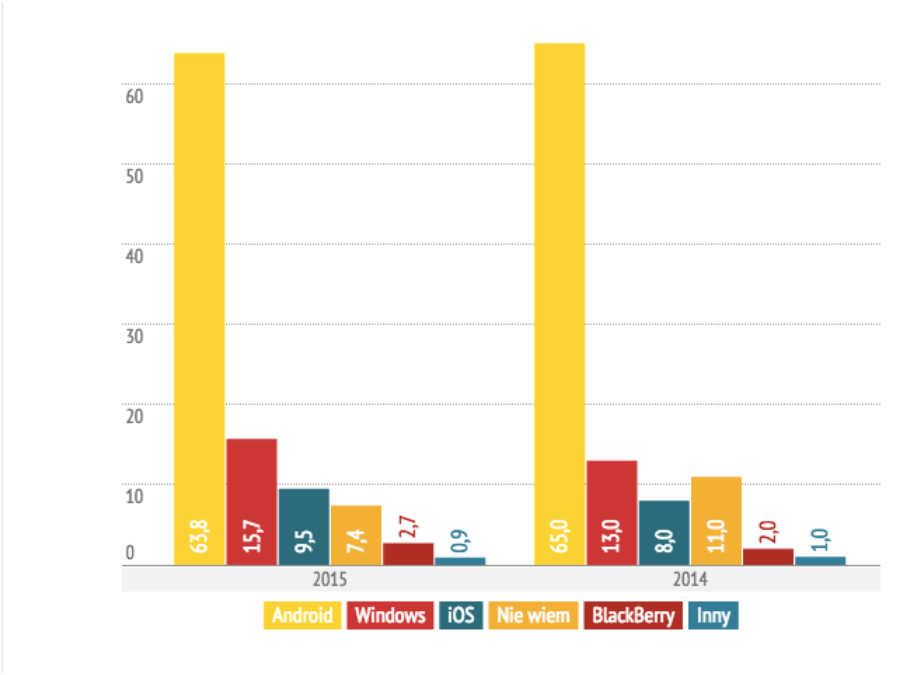
Rozdział 1

Wprowadzenie

Rozdział 2

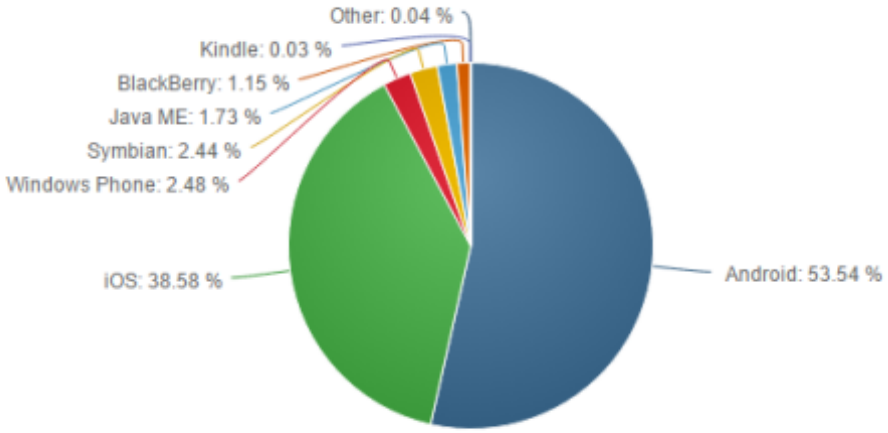
Android jako system operacyjny

Android – według wiadomości zawartych w Wikipedii – to system operacyjny z jądrem Linux, przeznaczony dla urządzeń mobilnych, takich jak telefony komórkowe, smartfony, tablety (tablety PC) i netbooki. Według portalu AntyWeb.pl w maju 2015 roku system ten miał największy udział w rynku urządzeń mobilnych w Polsce, a wartość tego udziału od 2014 roku to około 65



Rysunek 2.1: Android – udział w rynku urządzeń mobilnych w Polsce. (Źródło: portal AntyWeb.pl, 05/2015)

Drugie miejsce, według tego samego portalu, zajmuje system Windows, a trzecie iOS. Na świecie proporcje udziału są nieco inne, ale ciągle na czele jest system Android, z wynikiem ponad 53%, natomiast tutaj na drugim miejscu plasuje się już wyraźnie iOS z niemal 40-procentowym udziałem w rynku.



Rysunek 2.2: Android – udział w rynku urządzeń mobilnych na świecie. (Źródło: portal android.com.pl, 09/2015)

Jako system operacyjny dostępny nieodpłatnie, Android zrzesza przy sobie dużą społeczność deweloperów piszących aplikacje, które poszerzają funkcjonalność urządzeń. W sierpniu 2014 roku w Google Play (wcześniej Android Market), dostępnych było ponad 1,3 miliona aplikacji, zarówno komercyjnych, jak i darmowych.

Najpopularniejszymi językami programowania, w których piszemy aplikacje na Androida, są Java i C++ ze środowiskiem Android NDK. O ile język Java wydawać się może najrozsądniejszym wyborem na pierwszy rzut oka, o tyle wielu programistów używa również środowiska NDK. Jest to zestaw narzędzi, który pozwala realizować części swojej aplikacji za pomocą kodu macierzystego języków takich jak C i C ++. Zazwyczaj środowisko to wykorzystuje się w celu pisania programów, które intensywnie wykorzystują CPU, takich jak silniki gier, przetwarzania sygnału i symulacji fizyki. Jednak deweloperzy muszą wziąć pod uwagę również wady takiego rozwiązania, które mogą nie do końca zbilansować korzyści. Natywny kod Androida na ogół nie powoduje zauważalnej poprawy wydajności, ale za to zawsze znacznie zwiększa złożoność aplikacji, który to problem i tak już jest dużym wyzwaniem dla programistów Java, co przybliżę w kolejnych rozdziałach. Podsumowując, z NDK należy korzystać tylko wtedy, jeśli uznamy, że jest to niezbędne dla wytwarzanej aplikacji, a nie dlatego, że lepiej czujemy się programując w języku C/C++. Zanim zdecydujemy się na to rozwiązanie, najpierw należy sprawdzić, czy androidowe API zapewnia funkcjonalność, której potrzebujemy.

Java i C czy C++ to jednak nie wszystkie języki, których możemy użyć przy programowaniu aplikacji na Androida. Podczas szukania materiałów do tej pracy spotkałem się z przykładami aplikacji napisanych w C#, Delphi, czy nawet PHP. Stanowią one tak mały procent, że postanowiłem nie brać ich pod uwagę analizując opisywany problem testowalności oprogramowania na ten system operacyjny.

Rozdział 3

Testowalność oprogramowania

Jako analityk testowy z pojęciem testowalności oprogramowania spotykam się od początku mojej pracy w branży. Termin „testowalność oprogramowania" [1] według definicji ISTQB [2] i ISO9126, to najkrócej mówiąc właściwość tego oprogramowania umożliwiająca testowanie go po zmianach. Termin ten ściśle powiązany jest także z innym pojęciem ze słownika testerskiego – pielęgnowalnością. A pielęgnowalność [3] , to łatwość, z którą oprogramowanie może być modyfikowane w celu naprawy defektów, dostosowania do nowych wymagań, modyfikowane w celu ułatwienia przyszłego utrzymania lub dostosowania do zmian zachodzących w jego środowisku.

Po co tak w ogóle właściwie testować oprogramowanie? Czy testowanie jest potrzebne? Jak dużo testów należy przeprowadzić, aby testowanie było wystarczająco skuteczne? To są pytania, które mogą być tematem osobnej pracy, ale ja w skrócie postaram się na nie odpowiedzieć w tym rozdziale.

Człowiek, jako istota żywa i omylna, może popełnić podczas pracy **błąd**, czyli inaczej – **pomyłkę**. Pomyłka w pracy programisty może skutkować pojawieniem się **defektu** (usterki, pluskwy) w kodzie programu, bądź w dokumencie. Do tej pory nic się nie dzieje złego, ale jeżeli kod programu, który posiada w sobie taki defekt, zostanie wykonany, system może nie zrobić tego, co od niego wymagamy, lub wykonać to niezgodnie z założeniami, czyli inaczej rzecz ujmując, ulegnie **awarii**.

Defekty powstają, ponieważ jak już wspomniałem wcześniej, ludzie są omylni. Ale pomyłka człowieka, to nie jedyny powód awarii systemów. Mogą one być również spowodowane przez warunki środowiskowe, takie jak promieniowanie, pole magnetyczne i elektryczne, czy nawet zanieczyszczenia środowiska.

Czy testowanie pomaga nam w całkowitym uniknięciu awarii systemów? Na pewno nie, ale pozwala je drastycznie ograniczyć. Za pomocą zestawu testów możemy zmierzyć jakość oprogramowania wyrażoną przez ilość znalezionych usterek oraz możemy budować zaufanie do jakości oprogramowania, jeżeli jako osoby testujące znajdujemy ich mało, bądź nie znajdujemy ich wcale.

I tutaj najważniejsze zdanie: Testowanie samo w sobie nie poprawia jakości oprogramowania! Dopiero znalezienie, gdzie defekt znajduje się w kodzie programu (zdebugowanie) oraz

[1] Definicja testowalności według standardu ISO9126

[2] International Software Qualification Board

[3] Definicja pielęgnowalności według ISTQB

Tabela 3.1: Koszty znalezienia błędu na poszczególnych etapach projektu

Błąd znaleziony podczas	**Szacowany koszt**
Projektowania	1 PLN
Inspekcji (przeglądu)	10 PLN
W początkowej fazie produkcji	100 PLN
Podczas testów systemowych	1000 PLN
Po dostarczeniu produktu na rynek	10000 PLN
Kiedy produkt musi zostać wycofany z rynku	100000 PLN
Kiedy produkt musi zostać wycofany z rynku po wyroku sądowym	1000000 PLN

naprawa tego błędu przez programistę, poprawi nam tą jakość. Poniżej na rysunku widzimy tabelę (słynną już, gdyż korzysta z niej wielu trenerów prowadzących kursy z testowania oprogramowania), jak testowanie na poszczególnych etapach wytwarzania oprogramowania wpływa na koszty projektu.

Z zestawienia jasno wynika, że praca testerów nie zaczyna się gdy program już jest napisany przez programistów, a zaczyna się już w najwcześniejszej fazie projektu, na etapie projektowania.

Jak to jest więc z tą testowalnością oprogramowania dla Android? Czy aktualna struktura stosowana w większości aplikacji jest testowalna? Czy, o ile nie jest, da się tak poprawić strukturę programów, aby były bardziej testowalne niż obecnie? Jak nie psuć wcześniej działających części aplikacji wprowadzanymi zmianami w kodzie? Jak w ogóle wykrywać takie sytuacje, gdy coś przypadkowo zepsuliśmy? Jak dzielić odpowiedzialność pomiędzy częściami naszego oprogramowania? Jak rozwiązać problem zbyt dużego sprzężenia zarówno w naszym kodzie, jak i pomiędzy naszym kodem i frameworkiem androidowym [4]? No i w ogóle, jak testować aplikacje dla Androida poprawnie?

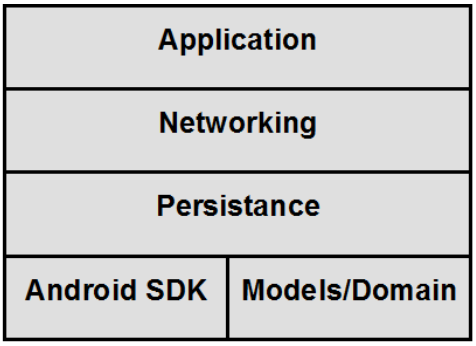
Na te pytania postaram się odpowiedzieć w drugiej części mojej pracy.

[4]Sprzężenie (*ang. coupling*) jest miarą jak bardzo obiekty, podsystemy lub systemy zależą od siebie nawzajem.

Rozdział 4

Opis problemu

W większości przypadków aplikacje z przeznaczeniem dla systemu Android pisane są według następującego schematu:



Rysunek 4.1: Podejście „standardowe” przy tworzeniu aplikacji dla systemu Android

Czyli, na najniższej warstwie mamy Android SDK (Software Development Kit) i na niej budujemy kolejne warstwy. Każda z kolejnych warstw naszego oprogramowania korzysta z warstwy poniżej, a co za tym idzie, dziedziczy również zależności z warstwy Android SDK. Analizując taką strukturę aplikacji dostępnych pod Androidem możemy zaobserwować, że w wielu z nich:

- nie jest zachowana zasada pojedynczej odpowiedzialności,
- warstwa odpowiedzialna za logikę domenową jest pomieszana z warstwą UI (User Interface),
- logika UI jest pomieszana z asynchronicznym pobieraniem danych,
- funkcje callback możemy znaleźć dosłownie wszędzie,
- elementy warstwy UI: Activity i Fragmenty potrafią mieć tysiące linii kodu,
- w większości plików aplikacji na każdej warstwie odwołujemy się do środowiska Android (import android.*)

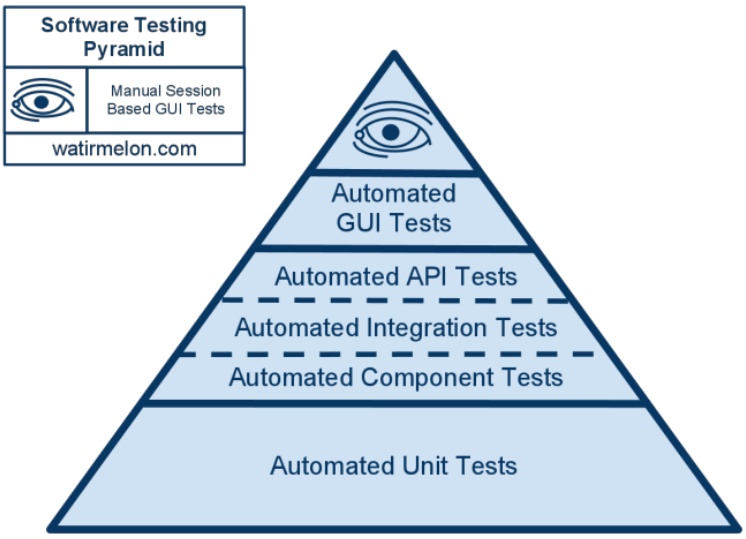
- i ostatnie, ale najważniejsze z punktu widzenia tej pracy: jeżeli w kodzie źródłowym szukamy zestawu unit testów – możemy się rozczarować.

Spowodowane jest to dwoma czynnikami: pierwszy to trudność w wyodrębnieniu obszarów testowych z powodu zbyt dużego sprzężenia między warstwami (wspomnianego już *couplingu*), a drugi – pracochłonność w pisaniu testów. Jeżeli granica pomiędzy kolejnymi warstwami oprogramowania nie jest jasno wyznaczona, liczba testów do zaprojektowania rośnie drastycznie. Wyjaśnię to na poniższym przykładzie:

Weźmy dwie funkcjonalności: funkcjonalność **A** opisaną za pomocą kodu z jedną instrukcją warunkową *„if”*, oraz funkcjonalność **B**, w której mamy dwie zależne od siebie instrukcje *„if”*, czyli cztery możliwe decyzje programowe. Testując te funkcjonalności razem (z powodu sprzężenia nie mamy innego wyjścia) musimy wykonać łącznie $2 * 4 = 8$ testów. Testy te równocześnie przestają być unit testami, gdyż łączą w sobie kilka funkcjonalności i stają się przez to testami integracyjnymi. Testując natomiast te funkcjonalności osobno, wykonamy $2 + 4 = 6$ unit testów. W tym przykładzie oczywiście nie widać zbyt wielkiej optymalizacji, ale w aplikacjach z kodem, który dostarcza setki czy tysiące decyzji, różnica będzie znacząca.

(dodać rysunki i ilustracje przedstawiające szerzej przykład)

Z doświadczenia własnego oraz kolegów testerów wiem, że jeżeli chcemy aplikację przetestować zaczynając od testów integracyjnych zamiast od unit testu, nakład pracy będzie zdecydowanie większy, niż gdy zastosujemy schemat standardowy, czyli zaczynając od unit testów, kontynuując poprzez testy integracyjne, następnie systemowe, a kończąc na akceptacyjnych (etapów może być więcej). Pozbawiając się możliwości zastosowania unit testów we wczesnym etapie projektu z powodu źle zaprojektowanej struktury aplikacji, ryzykujemy utratę jakości, a co za tym idzie - utratę zaufania klientów do naszego oprogramowania. Idealna piramida testowania, spopularyzowana przez Mike'a Cohna [1] w książce „Succeeding with Agile" [1], przedstawiona została na rysunku 4.2.



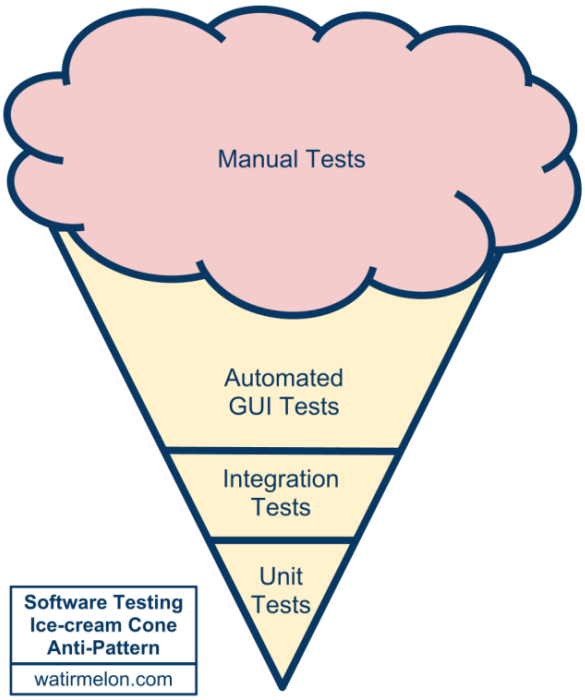
Rysunek 4.2: Idealna piramida testowania według Mike'a Cohna[1]

Z powyższego schematu wynika, że idealnie byłoby, gdyby wszystkie etapy testów zostały zautomatyzowane (wszystkie, z wyjątkiem manualnych testów akceptacyjnych, ale w idealnym świecie powinno być ich tak mało, że ich automatyzacja nie miałaby większego sensu). Oczywiście piramida ta może przybierać różne formy, poziomów testowania może być więcej lub mniej, mogą być one zautomatyzowane lub nie, ale idea jest cały czas ta sama: najwięcej testów powinno być na najniższym poziomie. Powinny być one również najprostsze do zaprojektowania. Im dalsza faza projektu, tym testy stają się bardziej pracochłonne, a koszt usunięcia znalezionego błędu wyższy, do czego autor nawiązał już w tabeli 3.1.

Wracając do analizy oprogramowania na platformę Android, aktualnie w większości przypadków schemat ten wygląda jednak tak jak na rysunku 4.3.

Na rysunku 4.3 widać, że z powodu zbyt dużego *couplingu* unit testy zastąpione zostają testami integracyjnymi, a najwięcej przypadków testowych wykonywanych jest na interfejsie użytkownika (w czym znacznie pomagają frameworki testowe pozwalające na zautomatyzowanie pewnych czynności, nagranie makr według specyfikacji lub *„user stories"*) oraz testy manualne.

[1]Mike Cohn jest jednym twórców metodologii tworzenia oprogramowania Scrum. Jest jednym z założycieli Scrum Alliance oraz właścicielem Mountain Goat Software, firmy, która oferuje szkolenia na Scrum i technik Agile.



Rysunek 4.3: Software Testing Ice Cream AntiPatern. Źródło: watirmelon.com

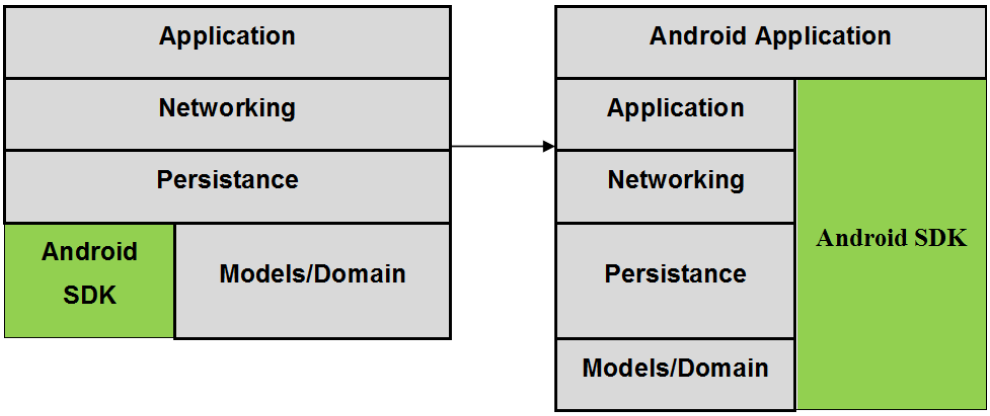
Autor nie zaprzecza, że tym sposobem nie da się dobrze przetestować aplikacji, szczególnie że według podręczników testerskich testowanie gruntowne nie jest możliwe, jakiejkolwiek metody nie użylibyśmy. Lecz ryzyko znalezienia błędu na dalszym etapie projektu jest w tym przypadku znacznie większe niż w przypadku oprogramowania o usystematyzowanej strukturze, a co za tym idzie – koszty jego usunięcia są również znacznie wyższe.

Propozycję rozwiązania problemu autor przybliży w kolejnym rozdziale.

Rozdział 5

Propozycja rozwiązania

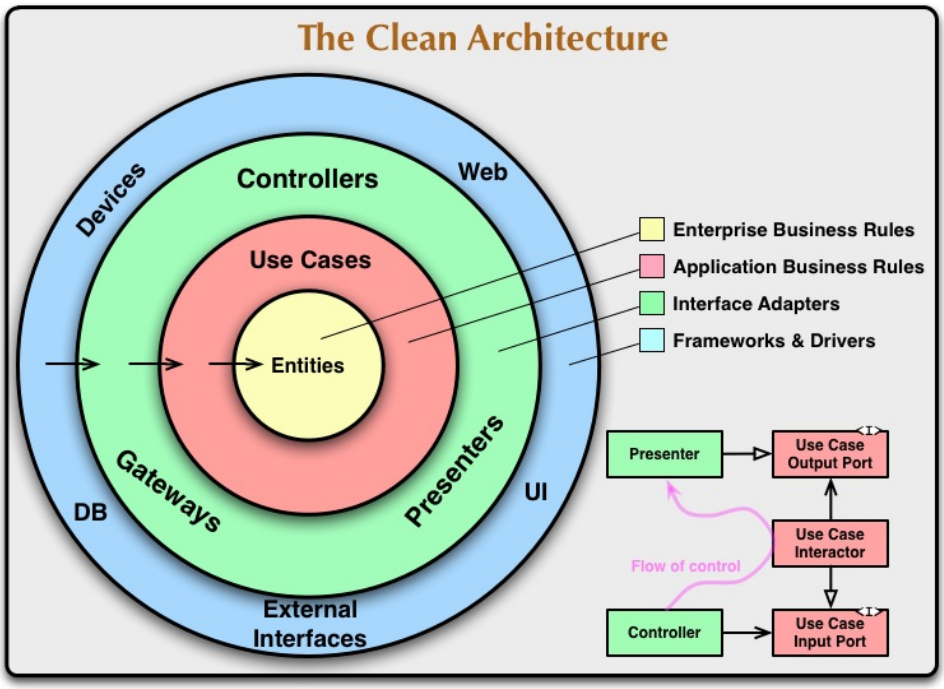
Największą przeszkodą w testowaniu aplikacji androidowych jest niestety sam Android. Im większy *coupling* między warstwami, tym trudniej jest pisać testy jednostkowe. Rozważmy więc przekształcenie modelu aplikacji Android opisanego na początku poprzedniego rozdziału w następujący sposób:



Rysunek 5.1: Zmodyfikowana struktura aplikacji Android: Clean Architecture

Nowy model aplikacji oddziela (przynajmniej w teorii) warstwy Application, Networking, Persistance oraz Model/Domain od środowiska Android SDK, a co za tym idzie nie musimy umieszczać odwołania do tego środowiska praktycznie w każdym pliku (wyrażenie `import android.*`). Dopiero najwyższą warstwą jest warstwa Aplikacja Android. W takim podejściu staramy się użyć Androida jako pewnego rodzaju *pluginu* do naszej aplikacji i uniezależnić warstwę logiczną od reszty warstw. Czyli najpierw napiszemy aplikację, która będzie oddzielona od Android SDK (na przykład w czystej Javie), potem dołączymy do tego android SDK i złączymy to wszystko w Aplikację Android. W ten sposób zapewnimy, że kod, który stanowi podstawę naszej aplikacji, czyli jej logikę działania, był przetestowany tak jak należy, czyli w oddzieleniu od reszty warstw.

Robert Cecil Martin [1], znany szerzej w środowisku informatycznym jako Uncle Bob, opublikował w 2012 roku na swoim blogu propozycję usystematyzowania kodu Android. Schemat przedstawiony jest na poniższym rysunku:



Rysunek 5.2: Clean architecture of Android według Uncle Ben. Źródło: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

Idea tego schematu jest następująca:

- Najważniejszym elementem jest środek, czyli warstwa *Entities*. W warstwie tej znajdują się kluczowe dla tworzonej aplikacji reguły biznesowe gwarantujące poprawność działania programu. Krótko mówiąc – znajduje się tam logika działania aplikacji.
- Drugą warstwą są *Use Cases*. W wolnym tłumaczeniu są to przypadki użycia, ale lepiej chyba używać nazwy intencje biznesowe. Jest to zbiór wszystkich zachowań, jakich oczekujemy od tworzonego systemu. W przypadku aplikacji bankowej, może być to na przykład funkcja wykonywania przelewu, jako jeden *use case*, a innym przypadkiem użycia byłoby sprawdzenie stanu konta.
- Wyższa, otaczająca warstwa dotyczy wszystkich kontrolerów i prezenterów. Warto zauważyć, że nadal na tym etapie nie mamy powiązania z systemem Android.

[1] Robert Cecil Martin (Uncle Bob) to amerykański inżynier programista oraz autor książek o programowaniu. Jest również współautorem „Agile manifesto"

- Dopiero na ostatniej, najbardziej zewnętrznej warstwie pojawia nam się powiązanie z Android SDK. Korzystamy z niego aby dostać się do baz danych, skorzystać z interfejsów użytkownika dla danego urządzenia, uzyskać dostęp do zewnętrznych interfejsów lub urządzeń.

I teraz rzecz najważniejsza: pomiędzy warstwami na schemacie narysowane są strzałki, które informują nas o kierunku przekazywania informacji. Ze schematu wynika, że *Entities* nie wiedzą nic o istnieniu *Use Cases*, *Use Cases* nie posiadają informacji o kontrolerach i prezenterach, a te z kolei nie wiedzą o istnieniu interfejsu użytkownika. Patrząc w drugą stronę, każda warstwa wyższa posiada wszystkie informacje o warstwie niższej, czyli UI wie wszystko o prezenterach, te wiedzą wszystko o *Use Cases*, no i przypadki użycia mogą korzystać z całej warstwy logicznej.

Przekładając to na język języków programowania, jeżeli używamy instrukcji `import android.*`, to tylko w stosunku do wyższej warstwy. Nie wolno nam tego robić w stosunku do warstwy niższej, bo wtedy cała koncepcja zostanie zachwiana.

W ten sposób teoretycznie jesteśmy w stanie stworzyć architekturę, która:

- jest niezależna od frameworka (tutaj Android SDK) – i zachowujemy zasadę zależności,
- jest niezależna od UI, bazy danych lub innych urządzeń zewnętrznych,
- jest testowalna, w oderwaniu od warstwy zawierającej interfejsy wymienione powyżej, czyli od wszystkiego, co czyni testy na Androidzie trudnymi.

Jeżeli ułożymy naszą strukturę aplikacji w powyższy sposób – testowanie powinno odbywać się jak w przypadku kodu napisanego w czystej Javie czy C++.

Rozdział 6

Opis przykładowej aplikacji

Na przykładzie jednej z aplikacji postaram się przedstawić różnicę w podejściu do testowania w przypadku programu dla systemu Android napisanego w architekturze „standardowej" i tego samego programu napisanego przy użyciu Clean Architecture.

Do tego celu autor wykorzysta aplikację „JSON Web Token Authentication for Android" napisanej przez Victora Albertosa , a której źródła udostępnione są w serwisie GitHub na licencji open source.

https://github.com/VictorAlbertos/RestAPIParseAuthAndroid https://github.com/VictorAlbertos/R

(tutaj jest cała część do opracowania jeszcze)

Rozdział 7

Wnioski

Bibliografia

[1] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, Michigan, wydanie first edition, 2013.

Spis rysunków

2.1 Android – udział w rynku urządzeń mobilnych w Polsce. (Źródło: portal AntyWeb.pl, 05/2015) 3
2.2 Android – udział w rynku urządzeń mobilnych na świecie. (Źródło: portal android.com.pl, 09/2015) 4

4.1 Podejście „standardowe” przy tworzeniu aplikacji dla systemu Android . . . 7
4.2 Idealna piramida testowania według Mike'a Cohna[1] 9
4.3 Software Testing Ice Cream AntiPatern. Źródło: watirmelon.com 10

5.1 Zmodyfikowana struktura aplikacji Android: Clean Architecture 11
5.2 Clean architecture of Android według Uncle Ben. Źródło: http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html 12

Spis tabel

3.1 Koszty znalezienia błędu na poszczególnych etapach projektu 6