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: mgr inż. Michał Włodarczyk

**Łódź, 26.02.2016**

# Spis treści


**1 Wstęp** **3**

**2 Android jako system operacyjny** **5**
2.1 Początki systemu . . . . . 5
2.2 Udział w rynku . . . . . 6
2.3 Rozwój systemu . . . . . 7
2.4 Programowanie w systemie w Android . . . . . 8

**3 Testowalność oprogramowania** **10**
3.1 Pojęcie testowalności i pielęgnowalności oprogramowania . . . . . 10
3.2 Czy testowanie jest potrzebne? . . . . . 10
3.3 Obszary testowe . . . . . 11
3.4 Rodzaje testów . . . . . 13
3.4.1 Testy jednostkowe (modułowe) . . . . . 14
3.4.2 Testy integracyjne . . . . . 15
3.4.3 Testy systemowe . . . . . 15
3.4.4 Testy akceptacyjne . . . . . 15
3.5 Zwinne podejście do testowania . . . . . 16
3.5.1 Wytwarzanie sterowane testowaniem - *Test Driven Development* . . . 16
3.5.2 Wytwarzanie sterowane zachowaniem - *Behaviour Driven Development* 17
3.5.3 Alternatywne techniki testowania . . . . . 17
3.6 Testowalność aplikacji androidowych . . . . . 18

**4 Opis problemu** **19**
4.1 Pojęcie architektury w kontekście systemów komputerowych . . . . . 19
4.1.1 Architektura systemowa . . . . . 19
4.1.2 Architektura oprogramowania . . . . . 19
4.2 Architektura Android . . . . . 21
4.3 Standardowe podejście przy tworzeniu aplikacji . . . . . 23
4.4 Trudności w testowaniu aktualnej struktury aplikacji . . . . . 24
4.5 Idealna i odwrócona piramida testowania . . . . . 24
4.6 Przyczyny rezygnacji z testów jednostkowych . . . . . 27
4.7 Pielęgnowalność aplikacji Android . . . . . 28

**5 Propozycja rozwiązania** **29**
5.1 Tworzenie aplikacji od podstaw . . . 29
5.1.1 Nowe podejście do architektury systemu . . . 29
5.1.2 Zastosowanie techniki *Test Driven Development* przy tworzeniu oprogramowania . . . 30
5.1.3 Automatyzacja testów jednostkowych . . . 31
5.1.4 Kod jako dokumentacja programu . . . 31
5.2 Różne podejścia do uporządkowania architektury . . . 32
5.2.1 Architektura cebulowa - *The Onion Architecture* . . . 32
5.2.2 Architektura portów i adapterów - *Ports and Adapters Architecture* . . . 33
5.2.3 Architektura uporządkowana - *The Clean Architecture* . . . 35
5.3 Praca z kodem zastanym . . . 36

**6 Porównanie testowalności na przykładzie aplikacji** **38**
6.1 Opis doświadczenia . . . 38
6.2 Opis aplikacji . . . 38
6.3 Zasada działania . . . 39
6.4 Analiza aplikacji pod względem testowalności . . . 39
6.4.1 Budowa analizowanej aplikacji w wersji pierwotnej . . . 40
6.4.2 Budowa analizowanej aplikacji w wersji poprawionej . . . 40
6.5 Przebieg doświadczenia . . . 43
6.6 Wyniki doświadczenia . . . 43
6.6.1 Wyniki dla testów jednostkowych . . . 43
6.6.2 Testy integracyjne . . . 45
6.7 Wnioski końcowe . . . 46

**7 Wnioski** **47**

# Rozdział 1

# Wstęp

Pojęcie *Android* ostatnio kojarzone jest przede wszystkim z rynkiem nowoczesnych urządzeń elektronicznych. Są to smartfony, tablety, netbooki, odbiorniki GPS, zegarki, a także telewizory, lodówki, pralki i inne urządzenia wykorzystywane obecnie w gospodarstwach domowych. Jako system operacyjny dostępny nieodpłatnie, Android zrzesza przy sobie ogromną społeczność developerów piszących aplikacje, które poszerzają funkcjonalność urządzeń. Aplikacje Android pozwalają developerom wykonywać te czynności bez konieczności sięgania do niższych warstw systemu. W zamian framework Androida dostarcza developerom bogate środowisko użytkownika, które pozwala na dostęp do różnych udogodnień, jakie urządzenie z tym systemem jest w stanie programiście zaoferować.

Tematem pracy jest "Testowalność aplikacji mobilnych na platformę Android". Z doświadczenia własnego autora oraz innych testerów można wnioskować, że aplikacje napisane dla tego systemu nie są łatwe do testowania. Standardowa architektura, z której korzysta obecnie większość programistów, nie pozwala na sprawne pisanie testów jednostkowych. Jeżeli test aplikacji rozpoczyna się od testów integracyjnych, nakład pracy będzie zdecydowanie większy, niż gdy zastosowany zostanie schemat standardowy, czyli zaczynając od *Unit Tests*, kontynuując poprzez testy integracyjne, następnie systemowe, a kończąc na akceptacyjnych.

Pozbawiając się możliwości zastosowania testów jednostkowych na wczesnym etapie projektu z powodu źle zaprojektowanej struktury aplikacji, ryzykujemy utratę jakości, a co za tym idzie - utratę zaufania klientów do naszego oprogramowania. Celem pracy jest przedstawienie alternatywnego podejścia do architektury aplikacji z przeznaczeniem dla systemu Android, a także przedstawienie alternatywnego sposobu podejścia do procesu pisania oprogramowania, które pomogłoby w podniesieniu testowalności oprogramowania na ten system. Z obserwacji autora wynika, że bardzo łatwo jest napisać zły i nietestowalny kod dla Androida. Warto jednak wiedzieć, że zastosowanie pewnych procesów i metodologii może znacznie poprawić testowalność naszych aplikacji.

W części pierwszej pracy, zawierającej trzy rozdziały, przedstawione zostanie aktualne, szeroko stosowane podejście do programowania aplikacji dla systemu Android. Rozdział drugi zawiera ogólne informacje na temat Androida: historię powstania i rozwoju systemu, analizę aktualnej popularności w segmencie urządzeń przenośnych, a także opis możliwości, jakie oferuje dla programistów. Rozdział trzeci dotyczy przede wszystkim testowania. Przypomniane zostaną podstawowe pojęcia testerskie, definicja testowalności i pielęgnowalności oprogramowania, przeprowadzona analiza obszarów testowych i rodzajów testów oraz wprowadzenie

do zwinnych podejść w procesie weryfikacji. W rozdziale czwartym autor opisuje kluczowy problem dla tej publikacji, czyli problem testowalności aplikacji pisanych dla Androida. Przypomina pojęcia architektury systemowej i softwarowej i opisuje aktualnie stosowaną architekturę aplikacji Android. Następnie naświetlone zostają trudności, jakie napotyka się podczas testowania napisanego w ten sposób oprogramowania.

Część druga to propozycja rozwiązania problemu nakreślonego w rozdziale czwartym. W rozdziale piątym autor proponuje zastosowanie techniki *Test Driven Development* w procesie tworzenia aplikacji oraz zachęca do wykorzystania jednej z trzech opisanych koncepcji uporządkowania architektury systemowej: *The Clean Architecture*. Proponuje również zwinne podejście w przypadku modyfikacji już istniejących aplikacji. Rozdział szósty to analiza testowalności jednej z gotowych aplikacji, napisanej raz w architekturze standardowej, i powtórnie z wykorzystaniem podejścia *TDD* i wspomnianej *Clean Architecture*.

Przy pisaniu pracy autor wykorzystywał zarówno pozycje książkowe, jak i artykuły blogowe takich znanych w świecie programistów osób, jak Mike Cohn, Alistair Cockburn, czy Robert Cecil Martin, czyli popularny *Uncle Bob*. Wykorzystano również dokumentację systemową systemu Android, Sylabusa fundacji ISTQB oraz wiele informacji znalezionych na blogach programistów. Definicje niektórych pojęć wyjaśnione zostały z pomocą Wikipedii.

Zdaniem autora, cel pracy został osiągnięty w około siedemdziesięciu procentach. Spowodowane to jest faktem, że w części doświadczalnej zajęto się tylko problemem testowalności w zakresie testów jednostkowych i wczesnych testów integracyjnych. Autor nie miał niestety możliwości przeanalizowania testowalności aplikacji Android na etapie testów systemowych czy akceptacyjnych. Natomiast można z dużym prawdopodobieństwem stwierdzić, że poprawienie testowalności w zakresie testów jednostkowych spowoduje jej polepszenie również na dalszych etapach procesu testowego.

Na koniec autor chciałby podziękować swoim promotorom za fachowe komentarze i nieocenioną pomoc przy rozwiązywaniu problemów technicznych i administracyjnych, bez której praca ta prawdopodobnie by nie powstała.

# Rozdział 2

# Android jako system operacyjny

## 2.1 Początki systemu

Słowo *android* w czasach obecnych używane jest w wielu kontekstach. Może odnosić się do humanoidalnego robota, znanego z literatury science-fiction, ale ostatnio przede wszystkim kojarzone jest z rynkiem nowoczesnych urządzeń elektronicznych. Są to telefony komórkowe z dotykowym ekranem (tzw. smartfony), tablety, netbooki, odbiorniki GPS, zegarki, a także telewizory, lodówki, pralki i inne urządzenia wykorzystywane w gospodarstwie domowym. Android to również nazwa firmy, system operacyjny, projekty *open source*[1], a nawet społeczność programistów.

Początki systemu Android sięgają roku 2003, kiedy to Andy Rubin, Chris White, Nick Sears oraz Rich Miner założyli w Kalifornii firmę Android Inc. Głównym celem powstania przedsiębiorstwa była chęć produkcji urządzeń mobilnych, które bazowałyby na danych lokalizacyjnych i uwzględniały preferencje użytkowników. Jednak zmagająca się z wymaganiami rynku oraz trudnościami finansowymi firma w 2005 roku została przejęta przez Google Inc., amerykańskie przedsiębiorstwo z branży internetowej, słynące z popularnej wyszukiwarki o tej samej nazwie. Wkrótce potem założono grupę *Open Handset Alliance* (OHA), w której skład weszły (oprócz Google) takie znane firmy jak HTC, Intel, Motorola, Qualcomm, T-Mobile, Sprint Nextel oraz NVIDIA, a która stawiała sobie za cel rozwój otwartych standardów dla telefonii mobilnej. Zdecydowanie przyspieszyło to badania nad systemem i jego rozwój. Pierwsza wersja Androida została przedstawiona światu już w 2007 roku, a pierwszym telefonem, który korzystał z tego systemu, był HTC G1.

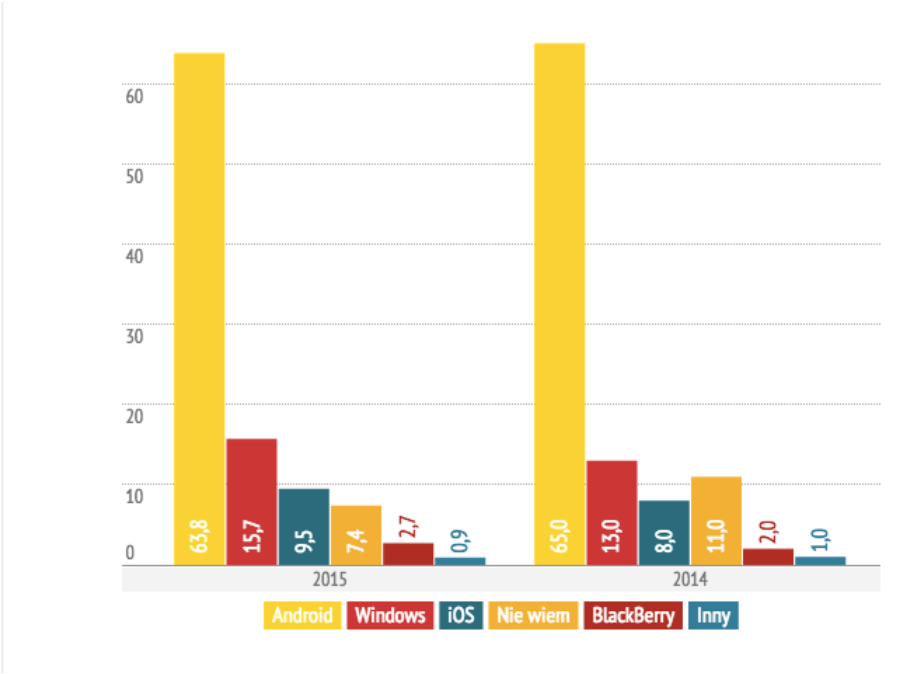
Dwie premierowe wersje systemu nie miały nazw nadanych zgodnych z konwencją, do której przywykli użytkownicy z całego świata. Dopiero począwszy od wersji 1.5, która została wydana 30 kwietnia 2009 roku, oficjalne wersje systemu otrzymywały już nazwy słodkich

[1]Otwarte oprogramowanie (ang. open source movement, dosł. ruch otwartych źródeł) – odłam ruchu wolnego oprogramowania (ang. free software), który proponuje nazwę open source software jako alternatywną dla free software, głównie z przyczyn praktycznych, a nie filozoficznych. Obok darmowego udostępniania może ono być sprzedawane i wykorzystywane w sposób komercyjny. Sposób osiągania zysków można także ograniczyć do sprzedaży dodatkowych usług, takich jak szkolenia z obsługi, wsparcie klienta czy dostęp do dodatkowych rozszerzeń, wtyczek, dodatków i modułów. Możliwe jest też wykorzystanie bezpłatnej wersji open source, jako sposób na zachęcenie do kupna bardziej rozbudowanej wersji dostarczanej na licencji komercyjnej. Źródło: Wikipedia

deserów lub smakołyków. Określenia przyznawane są zgodnie z porządkiem alfabetycznym, więc wersję 1.5 nazwano *Cupcake*.

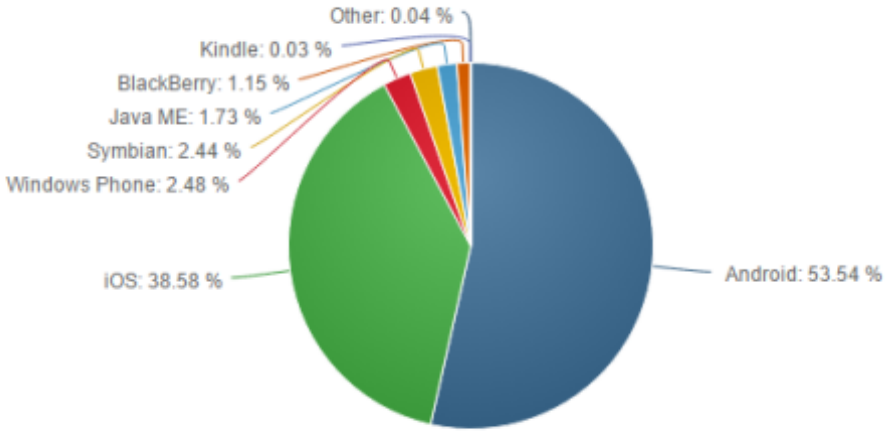
## 2.2 Udział w rynku

Android, jako system operacyjny przeznaczony dla urządzeń mobilnych, według [5] w maju 2015 roku miał największy udział w rynku tych urządzeń w Polsce, a jego wartość w 2014 roku przekroczyłą 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%. Na drugim miejscu natomiast 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)

## 2.3 Rozwój systemu

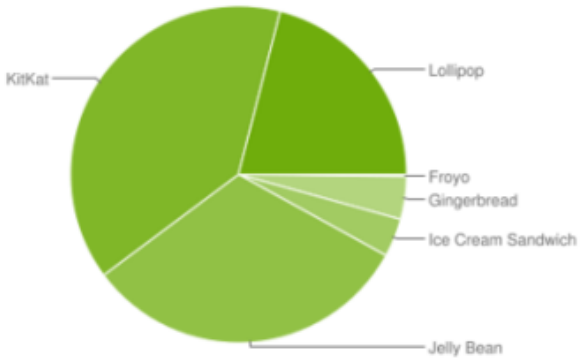
Niezwykle cenną informacją z punktu widzenia developerów, jest informacja o udziale poszczególnych wersji Android na urządzeniach posiadanych przez użytkowników. Dzięki niej mogą oni lepiej zoptymalizować swoje aplikacje pod kątem sprzętu. Dane z września 2015 przedstawiają się następująco:

Z urządzeń, na których instalowany jest Android, większość stanowią oczywiście smartfony i tablety. Ale nie tylko. Również urządzenia takie jak *smart-watches*, *smart-TVs*, akcesoria telewizyjne, konsole do gier, piekarniki, pralki, lodówki, satelity wysyłane w kosmos, a także najnowsze dziecko firmy Google - Google Glass, wspomagane są tym popularnym systemem operacyjnym. Co więcej, firmy motoryzacyjne zaczynają instalować Androida w swoich samochodach, rozbudowując w ten sposób platformę informacyjną i rozrywkową.

Zgodnie z manifestem założonej przez Google w 2007 roku grupy *Open Handset Alliance* (OHA), Android został zbudowany z wykorzystaniem wielu różnych komponentów na licencji *open source*. Zalicza się do nich przede wszystkim jądro Linux, niezliczone biblioteki programistyczne, kompletne interfejsy użytkownika, aplikacje i wiele wiele innych. Większość kodu systemu Android jest wydana na licencji *Apache Software License* (ASL, v 2.0). Wyjątkiem jest jądro Linuxa, które wykorzystuje GPLv2 oraz projekt *WebKit* korzystający z licencji BSD.

Niestety nie wszystkie części kodu Androida są otwarte dla programistów. Nawet urządzenia z należącej do Google linii Nexus zawierają komercyjne sterowniki, kodeki, a nawet całe aplikacje. Jest to duże utrudnienie dla programistów pragnących poznać tajniki budowy tego systemu i próbujących wykorzystać je do pisania własnych aplikacji. Mimo wszystko, wielu developerów, nie pracując dla Google bezpośrednio, zaangażowanych jest w tworzenie kodu nowych wersji systemu Android. Nie jest to proste, bo z niezrozumiałych przyczyn firma utrzymuje większość informacji dotyczących rozwoju swojego flagowego produktu w tajemnicy.

| Version | Codename | API | Distribution |
|---|---|---|---|
| 2.2 | Froyo | 8 | 0.2% |
| 2.3.3 - 2.3.7 | Gingerbread | 10 | 4.1% |
| 4.0.3 - 4.0.4 | Ice Cream Sandwich | 15 | 3.7% |
| 4.1.x | Jelly Bean | 16 | 12.1% |
| 4.2.x | | 17 | 15.2% |
| 4.3 | | 18 | 4.5% |
| 4.4 | KitKat | 19 | 39.2% |
| 5.0 | Lollipop | 21 | 15.9% |
| 5.1 | | 22 | 5.1% |




Rysunek 2.3: Statystyki dotyczące używania poszczególnych wersji systemu Android przez użytkowników (Źródło: portal androidnow.pl, 09/2015). Ostatnio doszła wersja 6.0, ale jej udział na rynku urządzeń w obecnej chwili jest znikomy.

W rozwijanie systemu są zaangażowani jednak nie tylko programiści. Wokół Androida zrzeszeni są również producenci procesorów, kości pamięci, urządzeń, ekranów i wielu innych części składających się na gotowy produkt.

## 2.4 Programowanie w systemie w Android

Jako system operacyjny dostępny nieodpłatnie, Android zrzesza przy sobie również ogromną społeczność developerów piszących aplikacje, które poszerzają funkcjonalność urządzeń. W sierpniu 2014 roku w internetowym sklepie Google Play (wcześniej Android Market)[2], dostępnych było ponad 1,3 miliona aplikacji, zarówno komercyjnych, jak i darmowych.

Najpopularniejszymi językami programowania, które wykorzystuje się do pisania aplikacji na Androida, są Java i C++ ze środowiskiem *Android NDK (Native Development Kit)*. 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 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 czy symulacji fizyki. Jednak deweloperzy muszą wziąć pod uwagę również wady takiego rozwiązania, które mogą nie

[2]Google Play (dawniej Android Market) – internetowy sklep Google z aplikacjami, grami, muzyką, książkami, magazynami, filmami i programami TV. Treści ze sklepu są przeznaczone do korzystania za pomocą urządzeń działających pod kontrolą systemu operacyjnego Android, ale z niektórych można także korzystać na laptopie czy komputerze stacjonarnym. Źródło: Wikipedia

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 autor postara się przybliżyć 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 programista zdecyduje się na to rozwiązanie, najpierw powinien sprawdzić, czy androidowe *API* nie zapewnia już funkcjonalności, jakiej potrzebuje.

Programując aplikacje dla Androida w języku Java, programiści wykorzystują Android SDK[3]. Ten zestaw narzędzi dla programistów przeznaczony do tworzenia aplikacji dedykowanych na tę platformę składa się z dwóch części: SDK Tools – wymaganej do tworzenia aplikacji niezależnie od wersji Androida, oraz Platform Tools – czyli narzędzi zmodyfikowanych pod kątem konkretnych wersji systemu. W skład środowiska programistycznego wchodzi dokumentacja, przykładowe programy, samouczki dla początkujących, biblioteki, emulator oparty na QEMU[4] oraz debugger. SDK dostępne jest zarówno dla Windows, jak i dla Linuxa.

Android SDK ma budowę modularną. Modułami są np. obrazy konkretnych wersji Androida, dodatkowe sterowniki, źródła SDK, czy przykładowe programy. Szczególnie pomocne są obrazy systemu uruchamiane na emulatorze, dzięki którym programiści mogą testować zachowanie aplikacji na różnych wersjach systemu Android, bez użycia fizycznych urządzeń [2].

Dużym ułatwieniem dla programistów Javy jest również Android Studio, czyli środowisko oparte na IntelliJ IDEA[5]. Jego główne zalety to zarządzanie wieloma projektami, współdzielenie plików, możliwość podłączenia do repozytorium, użycie *Gradle* do budowy oprogramowania, możliwość konfiguracji budowy programu w kilku wariantach dla jednego projektu, możliwość automatycznego uzupełniania kodu oraz wykonywania testów jednostkowych i integracyjnych bezpośrednio z narzędzia.

Wspomniany *Gradle* to narzędzie służące do budowania projektów w Javie. W przeciwieństwie do jego poprzedników: *Ant*-a i *Maven*-a, działa w oparciu o regułę *„convention over configuration”* polegającą na zminimalizowaniu potrzebnej konfiguracji, poprzez używanie gotowych wartości domyślnych: jeżeli czegoś nie ma w skrypcie konfiguracyjnym nie znaczy, że nie zostało skonfigurowane i nie zostanie wykorzystane. Oznacza to tylko, że nie zostały zmienione wartości domyślne.

Java i C czy C++ to jednak nie wszystkie języki, których można użyć przy programowaniu aplikacji na Androida. Podczas szukania materiałów do tej pracy autor spotkał się z przykładami aplikacji napisanych w C#, Delphi, czy nawet PHP. Stanowią one jednak tak mały udział, że nie będą one brane pod uwagę podczas analizy opisywanego problemu testowalności oprogramowania na ten system operacyjny.

[3]SDK - Software Development Kit

[4]QEMU - szybki emulator napisany przez Fabrice Bellarda.

[5]IntelliJ IDEA – komercyjne zintegrowane środowisko programistyczne (IDE) dla Javy firmy JetBrains.

# Rozdział 3

# Testowalność oprogramowania

## 3.1 Pojęcie testowalności i pielęgnowalności oprogramowania

Jako analityk testowy, z pojęciem testowalności oprogramowania autor spotyka się już od początku pracy w branży. Termin „testowalność oprogramowania" [1] według definicji ISTQB [2] i ISO9126, to właściwość tego oprogramowania umożliwiająca testowanie go po zmianach. Termin ten ściśle powiązany jest także z innym pojęciem ze słownika testerskiego – pielęgnowalnością. Pielęgnowalność [3]definiowana jest jako 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.

## 3.2 Czy testowanie jest potrzebne?

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 publikacji, w tym rozdziale poruszone zostaną więc tylko pojęcia, które wykorzystane są w dalszej części tej pracy.

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 dokumentacji. 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 się wymaga, lub wykonać to niezgodnie z założeniami. Czyli inaczej rzecz ujmując, ulegnie **awarii**.

Defekty powstają, ponieważ 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.

[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 |

Czy testowanie powoduje całkowite wyeliminowanie awarii systemów? Na pewno nie, ale pozwala je drastycznie ograniczyć. Za pomocą zestawu testów można zmierzyć jakość oprogramowania wyrażoną przez ilość znalezionych usterek oraz budować zaufanie do jakości oprogramowania, jeżeli testerzy znajdują ich mało, bądź nie znajdują ich wcale.

Należy jednak pamiętać, że testowanie samo w sobie nie poprawia jakości oprogramowania. Dopiero umiejscowienie defektu w kodzie programu (zdebugowanie) oraz naprawa tego błędu przez programistę poprawi jakość. Tabela 3.1 przedstawia poglądowo, jak testowanie na poszczególnych etapach wytwarzania oprogramowania może wpłynąć na koszty projektu.

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

## 3.3 Obszary testowe

Z punktu widzenia projektu, idealnie byłoby mieć możliwość przetestowania każdej linijki kodu. Autor zdaje sobie sprawę, że w wielu przypadkach jest to niemożliwe, a nawet bezcelowe, więc dopuszcza się możliwość ograniczenia testowania do kluczowych elementów kodu źródłowego i kluczowych funkcjonalności. Na przykład zwykle nie jest potrzebne testowanie *setterów*, *getterów*, czy funkcji należących do standardowych bibliotek, gdyż zapewne zostały przetestowane już podczas procesu ich tworzenia.

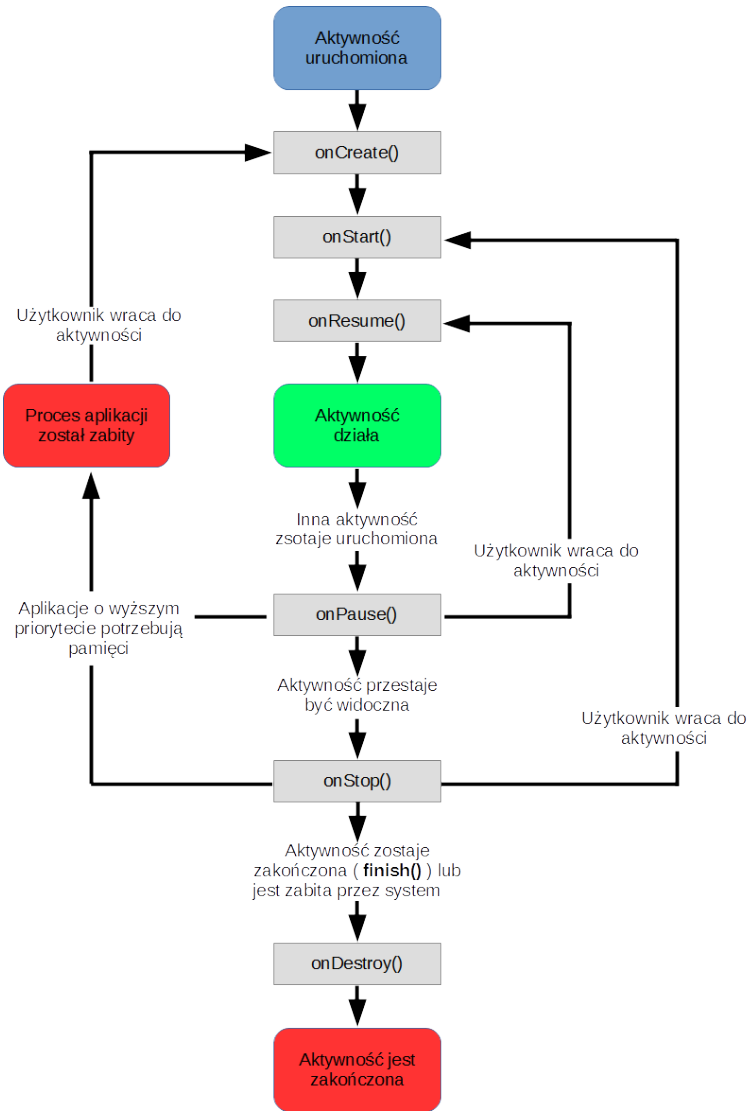
Bazując na publikacji [15], autor wyróżnił następujące trzy główne obszary testowania:

- Cykle życia aktywności

  Należy przetestować, czy aktywność przechodzi prawidłowo przez swoje cykle życia. Jeżeli od tworzonej aplikacji wymagamy, żeby zachowywała swój stan podczas `onPause()` i poźniej odtwarzała zachowany stan podczas wykonywania funkcji `onCreate()`, wtedy należy zaprojektować testy pokrywające wszystkie te przypadki i sprawdzające, czy zachowany status został odtworzony poprawnie. Trzy główne pętle do przetestowania znajdują się na schemacie 3.1:

- Dostęp do baz danych i do systemu plików

  Należy sprawdzić, czy operacje dostępowe przeprowadzane są poprawnie, a jeżeli nie, to czy również poprawnie działa obsługa błędów. Można to testować na dwa sposoby:




Rysunek 3.1: Cykl życia *Activity*. Źródło: Dokumentacja Android

albo na niskim poziomie, izolując warstwę użytkownika, albo bezpośrednio z aplikacji. Do testowania na niskim poziomie można wykorzystywać dostarczane przez framework Androida w pakiecie `android.test.mock` *zaślepki*[4]

Według dokumentacji Android[8] możliwe są następujące opcje przechowywania danych:

- *Shared Preferences*, czyli zachowywanie podstawowych danych w parach klucz - wartość,

- pamięć wewnętrzna urządzenia - do zachowywania danych niepublicznych,

- zewnętrzna karta pamięci - do zachowywania danych publicznych,

[4]Zaślepka (stub) - szkieletowa albo specjalna implementacja modułu używana podczas produkcji lub testów innego modułu, który tę zaślepkę wywołuje albo jest w inny sposób od niej zależny. Zaślepka zastępuje wywoływany moduł. [wg. IEEE 610]

- baza danych SQLite - do przechowywania danych w prywatnej bazie danych,

- zasoby sieciowe - jako baza danych współdzielona pomiędzy urządzeniami.

Wszystkie te opcje korzystają ze wspólnego zestawu funkcji[5], które tester powinien wziąć pod uwagę przy tworzeniu przypadków testowych.

- Fizyczną charakterystykę urządzenia

  Android został zaprojektowany do pracy na wielu różnych typach urządzeń, od telefonów do tabletów i telewizorów. Deweloperzy muszą tolerować pewną zmienność zachowań projektowanych funkcji i zapewnić elastyczny interfejs użytkownika, który dostosowuje się do różnych konfiguracji ekranu czy sieci.

  Należy sprawdzić, czy aplikacja działa poprawnie na wszystkich urządzeniach, na których można ją uruchomić. To, że działa świetnie na smartfonie, nie znaczy że działa poprawnie na tablecie, a to że działa na tablecie jednej firmy nie wyklucza awarii na tym samym urządzeniu wyprodukowanym przez kogoś innego. Elementy, które należy testować w tym zakresie, to:

  - możliwości sieciowe,

  - rozdzielczość ekranu,

  - gęstość ekranu,

  - rozmiar ekranu,

  - czułość sensorów,

  - klawiaturę i inne urządzenia wejściowe,

  - lokalizację GPS,

  - zewnętrzne karty pamięci.

  Android framework dostarcza rozwiązań pozwalających dostosowywać pewne rzeczy automatycznie, ale nie zwalnia to projektantów przed wykonaniem zestawu niezbędnych testów również w tym obszarze.

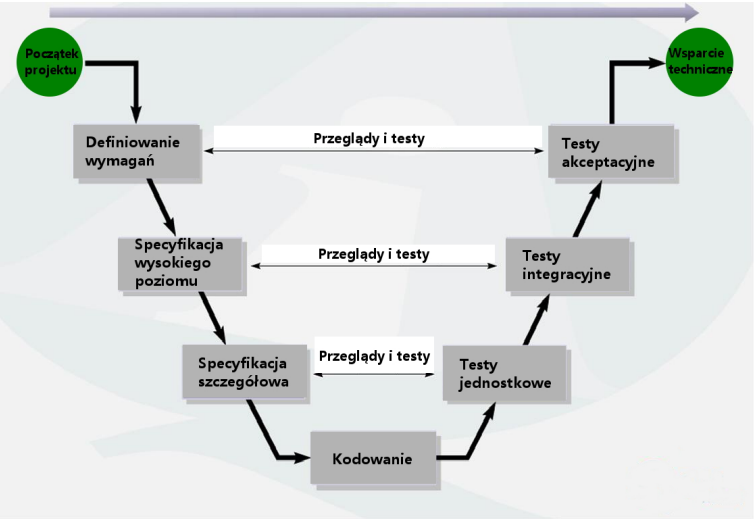
## 3.4 Rodzaje testów

Proces testowania może zostać wprowadzony na każdym etapie tworzenia aplikacji, w zależności od wybranej strategii testu. Jednakże, jak już autor wspomniał w podrozdziale 3.2, najlepiej jest rozpoczynać testowanie tak szybko jak to jest możliwe, w możliwie najwcześniejszym stadium projektu. Nawet jeżeli nie wszystkie wymagania systemowe zostały uzgodnione, a proces pisania oprogramowania się jeszcze nie rozpoczął. Już sam etap tworzenia dokumentacji i wymagań, czy to na poziomie klienta, systemowym, czy softwarowym, powinien zostać poddany testowaniu. Pozwala to unikać błędów już na etapie wczesnego projektowania, a które mogłyby wpływać na proces tworzenia przez długi czas.

[5]Na przykład do obsługi plików używa się *getFileDir(), getDir(), deleteFile()* itp.

Rozpoznaje się kilka poziomów testowania, w zależności od modelu zarządzania projektem. Rozważając model V przedstawiony na schemacie 3.2, wyróżnić możemy cztery główne interesujące z punktu widzenia testowalności obszary:

- testy jednostkowe (*Unit tests*)
- testy integracyjne
- testy systemowe
- testy akceptacyjne




Rysunek 3.2: Model V - najpopularniejszy model zarządzania projektem informatycznym (*Vmodel*). Źródło: [16]

### 3.4.1 Testy jednostkowe (modułowe)

Według [9], testy modułowe polegają na wyszukiwaniu błędów i weryfikacji funkcjonalności oprogramowania (np. modułów, programów, obiektów, klas), które można testować oddzielnie. Testowanie może być wykonywane w izolacji od reszty systemu, w zależności od kontekstu cyklu rozwoju oprogramowania i od samego systemu. Podczas testów można użyć zaślepek, sterowników testowych oraz symulatorów.

Mogą one zawierać testy funkcjonalności oraz niektórych atrybutów niefunkcjonalnych, takich jak stopień wykorzystania zasobów (np. wycieków pamięci) lub odporności. Wlicza się w nie również testy strukturalne - pokrycia linii kodu, decyzji lub gałęzi. Do projektowania przypadków testowych bardzo przydatna jest specyfikacja fumkcji. Wtedy jest możliwość zaprojektowania testów zanim zostanie napisany kod programu (tzw. "wytwarzanie sterowane testowaniem" - *Test Driven Development*, opisane szerzej w sekcji 3.5.1). Testy modułowe zwykle wykonuje się mając dostęp do kodu źródłowego i przy wsparciu środowiska rozwojowego (np. bibliotek do testów jednostkowych, narzędzi do debagowania). Testy jednostkowe

w praktyce zwykle angażują też programistę, który jest autorem kodu. Usterki są usuwane jak tylko zostaną wykryte, bez systemu formalnego nimi zarządzania nimi.

Najczęściej wykorzystywanym środowiskiem testowym dla Unit Testów pod Androidem jest *JUnit*. To proste i użyteczne narzędzie, pozwalające automatyzować testy jednostkowe, zostało zaprojektowane przez Ericha Gamma[6] i Kenta Becka i wydane na licencji *open source*[1].

Głównie badaniem tego rodzaju testów autor zajmował się będzie w części doświadczalnej pracy magisterskiej, w rozdziale 6.

### 3.4.2 Testy integracyjne

Testy integracyjne służą do sprawdzania interfejsów pomiędzy modułami, interakcji z innymi częściami systemu (takimi jak system operacyjny, system plików i sprzęt) oraz zależności pomiędzy systemami.

Wykonuje je się zwykle, jeżeli tylko jest możliwość przetestowania integracji gotowych już modułów, przetestowanych testami jednostkowymi. Pojęcie testowania integracyjnego można rozszerzyć również na testowanie integracji pomiędzy różnymi systemami, a nawet produktami różnych producentów.[9]

### 3.4.3 Testy systemowe

Testy systemowe zajmują się zachowaniem produktu lub systemu. Zakres takich testów powinien być jasno określony w głównym planie testów oraz w planach testów poszczególnych poziomów. Testy systemowe mogą zawierać testy oparte na ryzyku lub wymaganiach, procesie biznesowym, przypadkach użycia lub wysokopoziomowych opisach słownych i modelach zachowania systemu, interakcji z systemem operacyjnym i zasobami systemowymi.

Dąży się do tego, aby środowisko testowe w przypadku testów systemowych było maksymalnie zbliżone do środowiska docelowego, w którym projektowana aplikacja ma działać. Optymalnie byłoby pokryć wszystkie możliwe konfiguracje sprzętowe, aczkolwiek z wielu powodów, głównie finansowych, firmy skupiają się na najczęściej wykorzystywanych przez użytkownikach urządzeniach.[9]

### 3.4.4 Testy akceptacyjne

Celem testów akceptacyjnych jest nabranie zaufania do systemu, jego części lub pewnych atrybutów niefunkcjonalnych. Wyszukiwanie usterek nie jest głównym celem tego rodzaju testowania. Testy akceptacyjne mogą oceniać gotowość systemu do wdrożenia i użycia, chociaż nie muszą być ostatnim poziomem testowania. Na przykład może po nich następować testowanie integracji systemów w większej skali.

Odpowiedzialność za testy akceptacyjne, w przeciwieństwie do testów opisanych w poprzednich paragrafach, leży po stronie klientów. Zwykle testy akceptacyjne dzieli się na dwa etapy: testy *alfa* oraz *beta*. Testy *alfa* przeprowadzane są przez przyszłych użytkowników

[6]Erich Gamma - szwajcarski informatyk, współautor książki "Wzorce projektowe: elementy oprogramowania obiektowego wielokrotnego użytku". Wspólnie z Kentem Beckiem napisał narzędzie do tworzenia testów jednostkowych JUnit. Rozwijał także środowisko Eclipse.

w siedzibie producenta, natomiast *beta* - w środowisku docelowym. W praktyce producenci oprogramowania przekazują darmowe wersje swoich aplikacji o ograniczonych możliwościach, oczekując w zamian od użytkowników raportowania błędów lub propozycji usprawnień, bądź uzyskując to raportowanie automatycznie, kiedy aplikacja sama wysyła do producenta informację o zaistniałych defektach.[9]

## 3.5 Zwinne podejście do testowania

### 3.5.1 Wytwarzanie sterowane testowaniem - *Test Driven Development*

Technika wytwarzania sterowanego zarządzaniem zyskuje coraz więcej popularności. Związane to jest pośrednio z nowymi modelami zarządzania projektami informatycznymi opartymi na metodykach zwinnych, w tym Agile[7]. W założeniu podejście takie pozwala:

- znaleźć więcej błędów na wcześniejszym etapie procesu developmentu;
- świadomie zrobić zmiany wtedy, kiedy są potrzebne;
- łatwo wykonać testy regresywne bazując na wcześniej napisanych procedurach;
- przedłużyć żywotność kodu.

Jeżeli istnieją unit testy i pokrywają znaczącą część kodu źródłowego, wtedy jasne jest, że więcej błędów zostanie znalezionych i poprawionych. W ten sam sposób przy podejściu *Test Driven Development (TDD)* można wywnioskować, że skoro unit testy są już napisane, z dużym prawdopodobieństwem pokrycie kodu będzie zadowalające. Co więcej, raz napisane testy można wykorzystywać do przeprowadzania testów regresywnych, aby się upewnić, czy zmiany w naszym oprogramowaniu zgodne są z założeniami i wymaganiami systemu.

Testy jednostkowe w podejściu TDD zapewniają, że jakość oprogramowania nie jest oparta na domysłach i przekonaniu programistów, a na rzetelnych raportach. Bez obaw można dokonywać modyfikacji takich, jak zmiana serwera baz danych, zmiana designu, interfejsu użytkownika, czy nawet API. Jeżeli wszystkie testy zostaną sprawdzone pozytywnie - oprogramowanie będzie działać nadal bezbłędnie.

Ponadto TDD zmusza developera, żeby napisał tylko tyle linijek kodu, ile jest niezbędne dla powstania i działania danej fukcjonalności. Nie ma tu miejsca na wymyślanie własnych ścieżek i innowacji. Należy trzymać się prostych rozwiązań tak mocno, jak to możliwe, a co za tym idzie nie komplikować niepotrzebnie aplikacji.

Z powyższego można wnioskować, że podejście *TDD* może znacznie pomóc w zwiększeniu testowalności i pielęgnowalności aplikacji.

[7] Agile software development) – grupa metodyk wytwarzania oprogramowania opartego na programowaniu iteracyjno-przyrostowym, powstałe jako alternatywa do tradycyjnych metod typu *waterfall*[8]. Najważniejszym założeniem metodyk zwinnych jest obserwacja, że wymagania odbiorcy (klienta) często ewoluują podczas trwania projektu. Oprogramowanie wytwarzane jest przy współpracy samozarządzalnych zespołów, których celem jest przeprowadzanie procesów wytwarzania oprogramowania. (Źródło: Wikipedia)

### 3.5.2 Wytwarzanie sterowane zachowaniem - *Behaviour Driven Development*

BDD do kolejna technika testowania mająca swoje korzenie w metodykach zwinnych. Wywodzi się bezpośrednio z opisywanej wcześniej TDD, a także nawiązuje pośrednio do techniki *Acceptance Test Driven Development* (ATDD), opisanej w skrócie w sekcji 3.5.3.

BDD rozszerza TDD o następujące elementy:

- Wprowadza zasadę "pięciu pytań *Dlaczego*" dla każdego proponowanego przypadku użycia (*user story*). W ten sposób łatwiej jest ustalić rzeczywisty cel biznesowy.
- Stosuje strategię myślenia "z zewnątrz do wewnątrz", czyli programowania tylko tego, co jest rzeczywiście potrzebne z punktu widzenia biznesowego.
- Wprowadza jednoznaczny opis zachowania, tak aby z tej samej dokumentacji lub notatki mogli korzystać zarówno programiści, jak i testerzy i eksperci systemowi.
- Ten sposób działania wprowadzany jest od najwyższego (wymagania użytkownika) aż do najniższego (wymagania funkcjonalne) poziomu abstrakcji.

Dokumentacja w podejściu BDD oparta jest w znacznej mierze na przypadkach użytkownika (*user stories*). Oczywiście na niższych poziomach abstrakcji muszą one być odpowiednio uszczegółowione.

Główna różnica pomiędzy TDD a BDD jest więc taka, że TDD odnosi się do testów, a BDD do scenariuszy użycia, lub innych przypadków opisanych za pomocą charakterystycznych dla *Behaviour Driven Development* form *Given-When-Then*[9] (GWT).

Wszystkie te elementy powinny doprowadzić do zwiększonej współpracy programistów, testerów oraz specjalistów i ekspertów dziedzinowych. Zamiast terminu "testy jednostkowe", zwolennicy BDD wolą używać "specyfikacja zachowania klasy", a zamiast "testy funkcjonalne" - "specyfikacja zachowania produktu".

Stosowanie techniki *Behaviour Driven Development* nie wymaga żadnych specjalnych narzędzi ani języków programowania. Jest to podejście czysto koncepcyjne.[4]

### 3.5.3 Alternatywne techniki testowania

TDD i BDD to nie jedyne strategie testowania, które można przyjąć zwiększając testowalność aplikacji androidowych. Warto wspomnieć również o może nie tak popularnej, ale równie skutecznej strategii dotyczącej szczególnie testów akceptacyjnych, a jest nią *Acceptance Test-Driven Development* (ATDD). Jest to metodologia rozwoju opartego na komunikacji między klientami biznesowymi, programistami i testerami.

ATDD jest ściśle związane z Test-Driven Development. Nacisk na współpracę z wymaganiami klienckimi i systemowymi jest tutaj zdecydowanie większy niż w przypadku TDD, a wykorzystywana jest przede wszystkim przy tworzeniu testów akceptacyjnych.

Analizując jeszcze głębiej metodyki zwinne, możemy doszukać się również podejść *Example Driven Development* (EDD) oraz *Story test-Driven Development* (SDD). Różnice między

[9]GWT - forma tworzenia specyfikacji na podstawie przykładów

nimi a ATDD są tak niewielkie, że autor pominie ich szczegółowy opis sygnalizując tylko, że zastosowanie takich taktyk testowania również może mieć wpływ na poprawienie testowalności aplikacji pod Androidem.

## 3.6 Testowalność aplikacji androidowych

Z obserwacji autora wynika, że bardzo łatwo jest napisać zły i nietestowalny kod, nie tylko dla Androida. Warto jednak pamiętać, że zastosowanie procesów z metodologii wymienionych w poprzednich podrozdziałach zdecydowanie może podnieść testowalność naszych aplikacji. Techniki te znacznie pomagają deweloperom i testerom w zrozumieniu potrzeb klienta.

Ale czy zastosowanie metody TDD lub BDD jest wystarczające, aby podnieść testowalność aplikacji Android? Czy może potrzebna jest zmiana całej struktury aplikacji? Jak dzielić odpowiedzialność pomiędzy częściami oprogramowania? Jak rozwiązać problem zbyt dużego sprzężenia zarówno w napisanym kodzie, jak i pomiędzy kodem aplikacji i frameworkiem androidowym [10]? Oraz najważniejsze pytanie: jak testować aplikacje dla Androida skutecznie?

Na te pytania autor postara się odpowiedzieć w dalszej części pracy.

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

# Rozdział 4

# Opis problemu

## 4.1 Pojęcie architektury w kontekście systemów komputerowych

### 4.1.1 Architektura systemowa

Systemy komputerowe są kombinacjami sprzętu, oprogramowania i sieci komponentów. Termin *"architektura systemu"* opisuje strukturę, interakcję i technologię komponentów systemu komputerowego, w tym także oprogramowania.

Systemy oprogramowania są skonstruowane tak, aby spełnić cele biznesowe organizacji. Architektura systemowa jest określana jako abstrakcyjny pomost między tymi celami. Choć droga od abstrakcyjnych celów do konkretnych systemów może być złożona, dobrą wiadomością jest to, że architektura oprogramowania może być zaprojektowana, przeanalizowana, udokumentowana i realizowana przy użyciu znanych technik, które przyczynią się do osiągania tych celów biznesowych i misji twórcy.

### 4.1.2 Architektura oprogramowania

Istnieje wiele definicji architektury oprogramowania. Na potrzeby tej pracy autor wybrał przedstawioną w książce Lena Bassa, Paula Clementsa i Ricka Kazmana "Software Architecture in Practice"[11] *Architektura oprogramowania jest zbiorem struktur potrzebnych do uporządkowania systemu, które zawierają elementy oprogramowania i sprzętu oraz opisują relacje między nimi".*

W systemach opartych na modelu *waterfall*, architekturę oprogramowania tworzona jest przed rozpoczęciem kodowania i jest to bardzo poważny proces, który mógł wpłynąć na losy projektu. W modelach projektowych opartych na metodykach zwinnych, do których autor nawiązuje i przekonuje w tej pracy, architektura oprogramowania może być zmieniana również podczas tworzenia systemu. Czasem jest to nawet niezbędne ze względu na dynamiczne zmiany w wymaganiach użytkownika, lub pojawianie się kolejnych *use cases*.

Oto wnioski wynikające z przytoczonej wcześniej definicji:

**Architektura jest zbiorem struktur oprogramowania** Struktura jest zbiorem elementów spojonych ze sobą relacjami. Systemy oprogramowania składają się z wielu struktur, ale żadna pojedyncza struktura nie jest jeszcze architekturą. Istnieją trzy kategorie struktur architektonicznych, które odgrywają ważną rolę w zakresie projektowania, dokumentowania i analizy architektur:

- Niektóre systemy dzielą struktury na mniejsze jednostki wykonawcze, które zwykle nazywane są modułami. Modułom przypisane są konkretne zadania obliczeniowe, które stają się podstawą przypisania pracy dla zespołów programistycznych. W przypadku dużych projektów, elementy te (moduły) mogą być podzielone na jeszcze mniejsze części w celu przypisania do podrzędnych zespołów. Na przykład, zaprojektowanie bazy danych dla systemu zarządzania gospodarką magazynową, produkcją, księgowością i zasobami ludzkimi (ERP[1]) w dużym przedsiębiorstwie może być zbyt skomplikowane dla jednego zespołu i realizacja tego celu musi zostać podzielona na wiele części. Moduł logistyczny może stać się jednym modułem, moduł produkcyjny drugim, itd. Również inne funkcjonalności systemu muszą czasem zostać podzielone i przypisane do osobnych zespołów wdrożeniowych.

- Inne struktury systemowe mogą być dynamiczne, co oznacza, że dotyczą sposobu, w jaki poszczególne elementy współdziałają ze sobą w czasie wykonywania planowanych funkcji systemu. Jeżeli system ma być zbudowany jako zestaw usług, to usługi te współdziałają z infrastrukturą i wymagają synchronizacji.

- Trzeci rodzaj struktury opisuje odwzorowanie elementów softwarowych na elementy systemowe: organizacyjne, instalacyjne, uruchomieniowe i rozwojowe. Na przykład, moduły mogą być przypisane do zaprogramowania przez zespoły oraz przypisane do odpowiednich miejsc w strukturach systemu, tak aby później nie było problemów chociażby z ich testowaniem. Te odwzorowania nazywane są *alokacyjnymi*.

**Architektura jest pojęciem abstrakcyjnym** Ponieważ architektura zawiera struktury, a struktury te zawierają elementy i relacje między nimi, wynika z tego że architektura łączy te elementy systemowe ze sobą wykorzystując relacje. Oznacza to, że architektura pomija pewne informacje, które nie są użyteczne z punktu widzenia systemu, a w szczególności takie, które nie są ze sobą powiązane żadną relacją. Zatem architektura jest przede wszystkim abstrakcją systemu, który wybiera pewne szczegóły i ukrywa inne. We wszystkich nowoczesnych systemach elementy współdziałają ze sobą za pomocą interfejsów, które dzielą dany element na części publiczne i prywatne (wewnętrzne). Architektura zajmuje się stroną publiczną tego podziału, a prywatne interakcje między funkcjami, czy nawet funkcje pełniące tylko zadania pomocnicze, nie są elementem architektonicznym. Taki rodzaj abstrakcji jest niezbędny do opisania systemu w możliwie najprostszy i zrozumiały sposób.

**Każdy system softwarowy posiada architekturę oprogramowania** Każdy system można pokazać w sposób opisany w poprzednich paragrafach, czyli jako zestaw elementów i

[1]ERP - Enterprise Resource Planning (ang.) – planowanie zasobów przedsiębiorstwa

relacji między nimi. W najprostszym przypadku - cały system będzie pojedynczym elementem, ale architektura opisana w ten sposób będzie bezużyteczna.

Nawet jeżeli każdy system posiada architekturę, to nie znaczy, że architektura ta jest znana. Zdarza się, że ludzie, którzy ją tworzyli, już nie pracują, a dokumentacja zaginęła lub, co gorsze, w ogóle nie była tworzona. Jeśli do tego kod źródłowy nie został zachowany, zostaje tylko binarny kod wykonywalny, który niekoniecznie może być pomocny przy analizie systemu. Różnica pomiędzy architekturą systemu a jej reprezentacją polega na tym, że architektura może istnieć niezależnie od specyfikacji systemu. Warunkiem koniecznym do odtworzenia systemu jest jej zapisana dokumentacja.

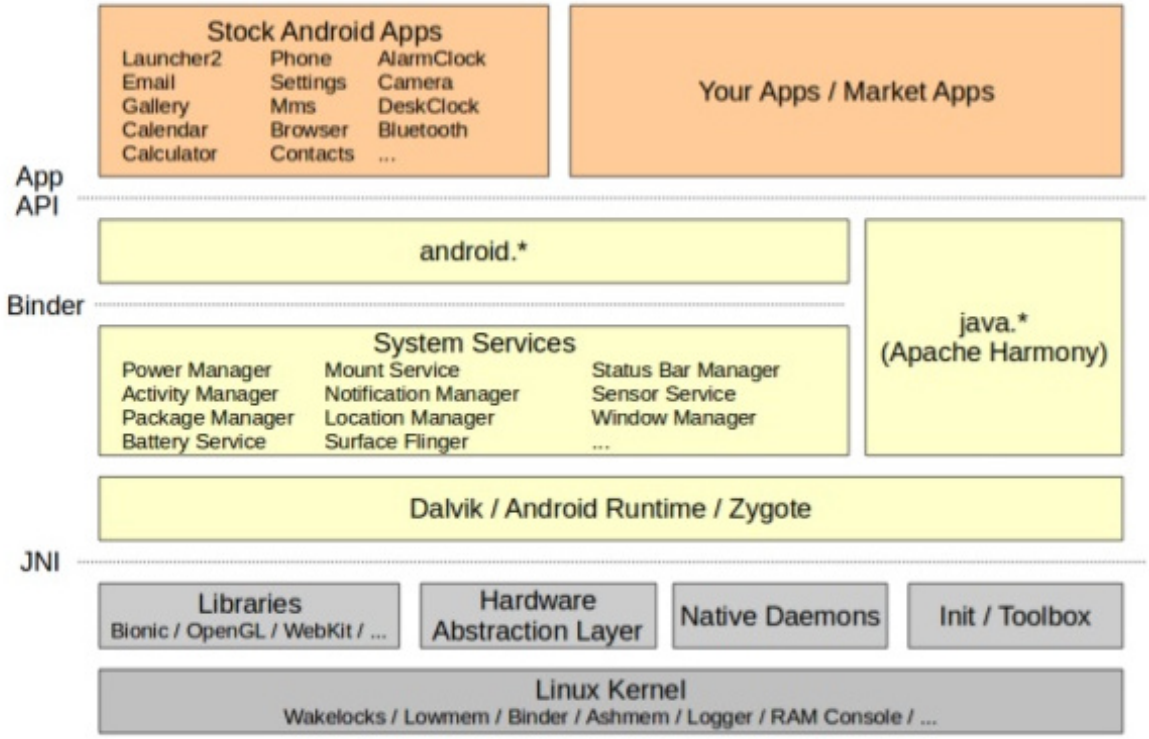
**Architektura obejmuje również zachowanie systemu** Zachowanie się każdego elementu architektury może być również jej częścią, o ile czynność ta wnosi jakieś nowe informacje i wpływa pozytywnie na zrozumienie systemu. Reprezentowanie, jak elementy współdziałają ze sobą, jest częścią definicji architektury, opisanej w poprzednich paragrafach. Rysunki typu *box-and-line*, które często przedstawiane są jako architektura systemu, w gruncie rzeczy nią nie są. Patrząc na nazwy pól (bazy danych, graficzny interfejs użytkownika, etc.), można sobie wyobrazić funkcjonalność i zachowanie odpowiednich elementów, ale wypływają one w większości z wyobraźni obserwatora i nie opierają się na żadnej udokumentowanej informacji. To nie znaczy, że dokładne zachowanie i wydajność każdego elementu muszą być udokumentowane w każdym przypadku - niektóre zachowania systemu nie leżą w grupie zainteresowań architekta. Jednak jeżeli interakcja pomiędzy poszczególnymi elementami jest kluczowa dla działania systemu - powinna ona zostać udokumentowana.

**Nie wszystkie rodzaje architektur są odpowiednie** Definicja architektury nie specyfikuje niestety, która architektura jest dla danego systemu odpowiednia bądź nie - lub - mówiąc bardziej dosadnie, która jest dobra, a która zła. Jak autor wykaże w kolejnych rozdziałach, również w przypadku systemu Android wybór odpowiedniej architektury jest kluczowy dla testowalności i pielęgnowalności oprogramowania tworzonego dla tej platformy.

## 4.2 Architektura Android

Architektura Android przez wielu opisywana jest jako *Java on Linux*, czyli programowanie w Javie pod Linuksem. Jednakże jest to stwierdzenie zbyt ogólne, biorąc pod uwagę złożoność całej platformy. Całość systemu zawiera bowiem w sobie komponenty, które układają się w pięć głównych warstw: *Android applications*, *Android Framework*, *Dalvik virtual machine*, *user-space native code* oraz *Linux kernel*[10].

Aplikacje Android pozwalają developerom rozszerzać i ulepszać funkcjonalność urządzeń, bez konieczności sięgania do niższych warstw systemu. W zamian framework Androida dostarcza developerom bogate środowisko użytkownika, które pozwala na dostęp do różnych udogodnień, jakie urządzenie z tym systemem jest w stanie programiście zaoferować. Jest to swoiste połączenie pomiędzy aplikacjami a wirtualną maszyną Dalvika, które zezwala




Rysunek 4.1: Przegląd architektury Android. Źródło: Karim Yaghmour of Opersys Inc., `http://http://www.slideshare.net/opersys/inside-androids-ui`

na konfigurowanie interfejsu użytkownika (UI[2]), dostęp do baz danych oraz przekazywanie informacji pomiędzy poszczególnymi komponentami aplikacji.

Zarówno aplikacje jak i opisywany *framework* napisane są w Javie i wykonywane na wirtualnej maszynie Dalvika (*DalvikVM*). Jest to specjalnie zaprojektowana wirtualna maszyna z własnym kodem bajtowym, zoptymalizowana pod jądro Linuxa i pozwalająca na uruchamianie aplikacji androidowych. Niestety, nie wszystkie biblioteki wykorzystane przy jej tworzeniu udostępnione są na zasadzie wolnych licencji.

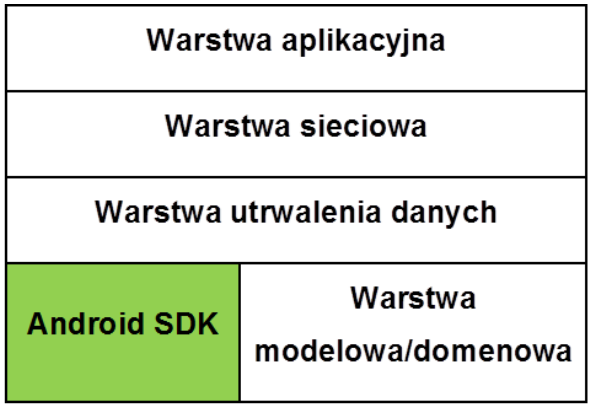
Natywne elementy kodu Androida zawierają usługi systemowe, usługi sieciowe (takie jak *DHCP* i *wpa_supplicant*) oraz różnego rodzaju biblioteki, między innymi *WebKit* i *OpenSSL*.

Najniższą warstwą, a zarazem podstawą systemu Android jest jądro Linuxa. Android dokonał licznych uzupełnień i zmian w jego źródłach, z których niektóre mają swoje negatywne konsekwencje dla bezpieczeństwa (ale nie będziemy zajmować się tym w ramach tej publikacji). Sterowniki znajdujące się w jądrze systemu zapewniają również dodatkowe funkcje, takie jak dostęp do kamery, sieci Wi-Fi oraz dostęp do innych urządzeń sieciowych. Szczególnie godny uwagi jest sterownik *Binder*, który realizuje komunikację między procesami[10] (IPC[3]).

[2]User Interface
[3]Inter-Process Communication

## 4.3 Standardowe podejście przy tworzeniu aplikacji

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




Rysunek 4.2: Podejście „standardowe" przy tworzeniu aplikacji dla systemu Android

Czyli, na najniższej warstwie położony jest Android SDK i na niej budowane są kolejne warstwy. Każda z kolejnych warstw oprogramowania korzysta z warstwy poniżej, a co za tym idzie, dziedziczy również zależności z warstwy Android SDK. Analizując taką strukturę aplikacji dostępnych pod Androidem można 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*[4] możemy znaleźć w całym kodzie,
- 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 ktoś będzie szukał zestawu testów jednostkowych – można się rozczarować.

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

[4]Wywołanie zwrotne (ang. callback) jest to technika programowania będąca odwrotnością wywołania funkcji. Zwykle korzystanie z właściwości konkretnej biblioteki polega na wywołaniu funkcji (podprogramów) dostarczanych przez tę bibliotekę. W tym przypadku jest odwrotnie: użytkownik jedynie rejestruje funkcję do późniejszego wywołania, natomiast funkcje biblioteki wywołają ją w stosownym dla siebie czasie. Źródło: Wikipedia

## 4.4 Trudności w testowaniu aktualnej struktury aplikacji

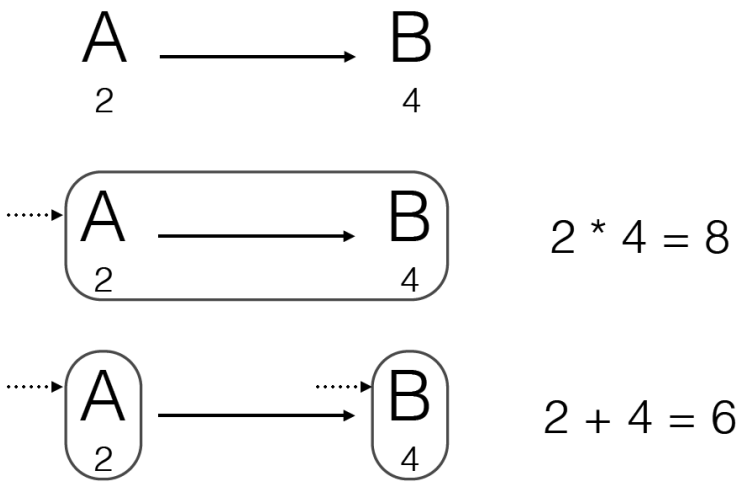
Weźmy dwie funkcjonalności: funkcjonalność **A** opisaną za pomocą kodu z jedną instrukcją warunkową *„if”*, oraz funkcjonalność **B**, w której mamy dwie zależne od siebie instrukcje *„if”*, czyli cztery możliwe decyzje programowe. Testując te funkcjonalności razem (z powodu sprzężenia nie mamy innego wyjścia) należy wykonać łącznie

$$2 * 4 = 8$$

testów. Testy te równocześnie przestają być testami jednostkowymi, gdyż łączą w sobie kilka funkcjonalności i stają się przez to testami integracyjnymi. Testując natomiast te funkcjonalności osobno, trzeba wykonać

$$2 + 4 = 6$$

testów jednostkowych. W tym przykładzie oczywiście nie widać zbyt wielkiej optymalizacji, ale w aplikacjach z kodem, który dostarcza setki czy tysiące decyzji, różnica będzie znacząca.



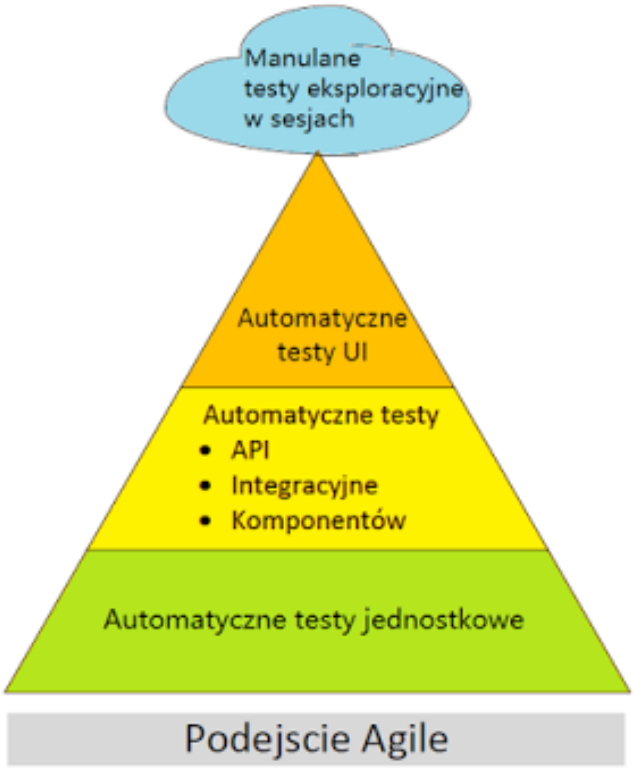

Rysunek 4.3: Różnice w testowaniu klas osobno i razem. *"Integrated Tests are a Scam"* - schemat autorstwa J.B. Rainsbergera. Źródło: `https://vimeo.com/80533536)`

Również w przypadku architektury aplikacji Android, jeżeli zachodzi konieczność testowania każdej klasy lub pojedynczej funkcji w powiązaniu z *Android SDK*, liczba testów jednostkowych zauważalnie wzrośnie.

## 4.5 Idealna i odwrócona piramida testowania

Z doświadczenia własnego autora oraz innych testerów można wywnioskować, że jeżeli aplikacja ma być przetestowana zaczynając od testów integracyjnych zamiast od testów jednostkowych, nakład pracy będzie zdecydowanie większy, niż gdy zastosowany zostanie schemat standardowy, czyli zaczynając od *Unit Tests*, kontynuując poprzez testy integracyjne, następnie systemowe, a kończąc na akceptacyjnych (etapów może być więcej).

Pozbawiając się możliwości zastosowania testów jednostkowych na 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 [5] w książce „Succeeding with Agile" [7], przedstawiona została na rysunku 4.4.

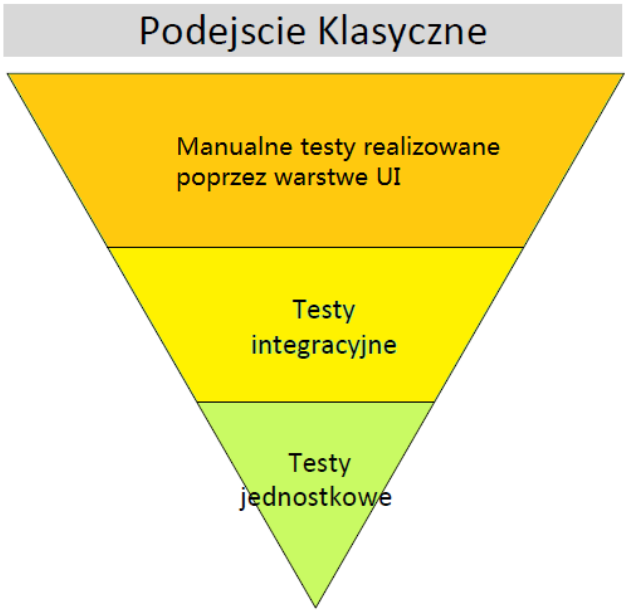



Rysunek 4.4: Idealna piramida testowania według Mike'a Cohna[7]. Źródło: `http://www.scrumdo.pl/2015/08/piramida-testow-agile.html`

[5]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.

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 przypadków testowych 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.

Analizując aktualną strukturę większości aplikacji androidowych, schemat ten wygląda jednak tak jak na rysunku 4.5.




Rysunek 4.5: Podejście klasyczne do testów, czyli ddwrócona piramida testowania (*Ice Cream AntiPatern*). Źródło: `http://www.scrumdo.pl/2015/08/piramida-testow-agile.html`

Na rysunku 4.5 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.

Autor nie zaprzecza, że tym sposobem nie da się dobrze przetestować aplikacji, szczególnie jeżeli zastosujemy taktyki testowania opisane w rozdziale 3. Jak powszechnie wiadomo, 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.

Okazuje się, że strukturę aplikacji Android da się jednak przeprojektować tak, aby ułatwić pracę zarówno programistom jak i testerom. Propozycja zmiany struktury aplikacji Android zaproponowana została w rozdziale 5.1.1

## 4.6 Przyczyny rezygnacji z testów jednostkowych

Trudna w testowaniu struktura aplikacji to jednak nie jedyna przyczyna niedostatecznej ilości testów jednostkowych, a nawet ich braku w procesie tworzenia aplikacji. Przyczyn takiej sytuacji może być wiele, a najważniejsze z nich, zdaniem autora, to:

- Niedostateczna znajomość języka programowania wśród programistów

  Jeżeli programista, nawet dobry, został zmuszony przez sytuację do poznawania nowego języka programowania, to w pierwszej kolejności chciałby, aby jego kod się kompilował, a program zaczął działać. Jeżeli zaczyna pisać testy, to znaczy, że ma na to czas i nie musi zagłębiać się w techniki konfiguracji kompilatora, czy środowiska programistycznego.

- Słaba znajomość narzędzi testowych

  Jeżeli programista nie doskonali się w temacie projektowania testów, nie poznaje narzędzi testowych, które mogą spowodować, że testowanie stanie się łatwe i przyjemne, szybko zniechęci się już przy pierwszej próbie napisania trudniejszego testu, czy próbie *zamockowania* jednego z interfejsów.

- Niska jakość kodu źródłowego

  Kiepsko zaprojektowany kod i niezbyt optymalnie zaprojektowana architektura systemowa powoduje, że nie ma możliwości w rozsądnym czasie przygotować zestawu testów. To powoduje, że firmy rezygnują z testów jednostkowych na rzecz testów integracyjnych i systemowych, co pokazane zostało na rysunku 4.5.

- Brak czasu na testy

  W dzisiejszych czasach koszty projektów informatycznych są zwykle ograniczone. W związku z tym programiści starają się wykorzystywać cały swój czas aż do *deadline'ów* na "ulepszanie" kodu, co skutkuje brakiem czasu na testy jednostkowe.

- Przekonanie programistów o własnej nieomylności

  Człowiek jest omylny i popełnia błędy, do czego już autor nawiązał w rozdziale 3. Niektórzy programiści nie chcą jednak przyjąć tego do wiadomości i upierają się przy stwierdzeniach, że testy do ich kawałka kodu źródłowego nie są potrzebne.

Należy zastanowić się, czy zastosowanie jednej z taktyk zwinnych, w tym przypadku TDD, pozwoliłoby usystematyzować pracę nad projektem i usunąć wszystkie powyższe przyczyny.

## 4.7 Pielęgnowalność aplikacji Android

W realnym świecie niestety bardzo rzadko mamy możliwość tworzenia aplikacji od początku. W większości przypadków programiści muszą borykać się z kodem, który ktoś już kiedyś napisał, a dotyczy to w zasadzie wszystkich większych projektów informatycznych. Weźmy na przykład taką aplikację jak *Gmail*. Wychodzą ciągle nowe wersje, ale trudno wyobrazić sobie sytuację, że któraś z nich została po prostu napisana od nowa. W takich przypadkach mamy do czynienia z tzw. **kodem zastanym**.

O ile kod pisany był w sposób przejrzysty, dobrzy programiści są w stanie dobudować nowe części aplikacji, nawet nie mając dokumentacji do starszej części. Wiele firm programistycznych stosuje podejście, że sam kod programu jest zarówno jego dokumentacją. Rozsądne nadawanie nazw zmiennym oraz umieszczanie rzeczowych komentarzy pozwala programistom zrozumieć swoich poprzedników, a automatycznym narzędziom w stylu *JavaDoc*[6] wygenerować całkiem wyczerpującą dokumentację.

Gorsza sytuacja jest z testowaniem. Jeżeli produkt nie był tworzony z zastosowaniem TDD, praca jaką należałoby wykonać przy pisaniu unit testów do gotowego kodu może być ogromna. Ponadto istnieje ryzyko, że tester chcąc sprostać wymaganiom osób zarządzających projektem będzie tak pisał testy jednostkowe, aby pokrywały jak największą część kodu, niekoniecznie przy tym wnosząc jakoś wartość w sprawdzenie niezawodności aplikacji.

Jedno z możliwych rozwiązań zaproponowane zostało w rozdziale 5.3

[6] Javadoc – narzędzie automatycznie generujące dokumentację na podstawie zamieszczonych w kodzie źródłowym znaczników w komentarzach. Javadoc został stworzony specjalnie na potrzeby języka programowania Java przez firmę Sun Microsystems. Źródło: Wikipedia

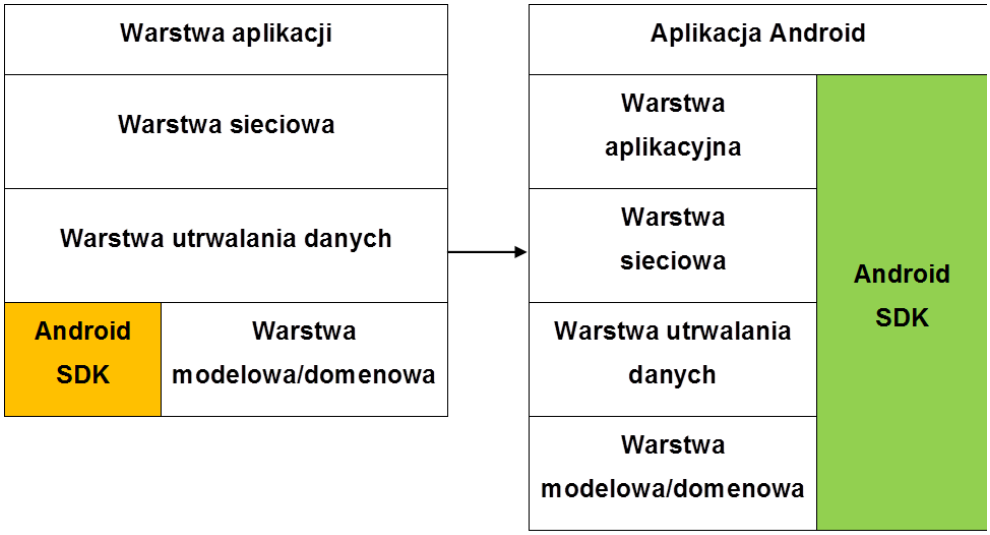
# Rozdział 5

# Propozycja rozwiązania

## 5.1 Tworzenie aplikacji od podstaw

### 5.1.1 Nowe podejście do architektury systemu

Okazuje się, że największą przeszkodą w testowaniu aplikacji androidowych jest Android sam w sobie. 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:



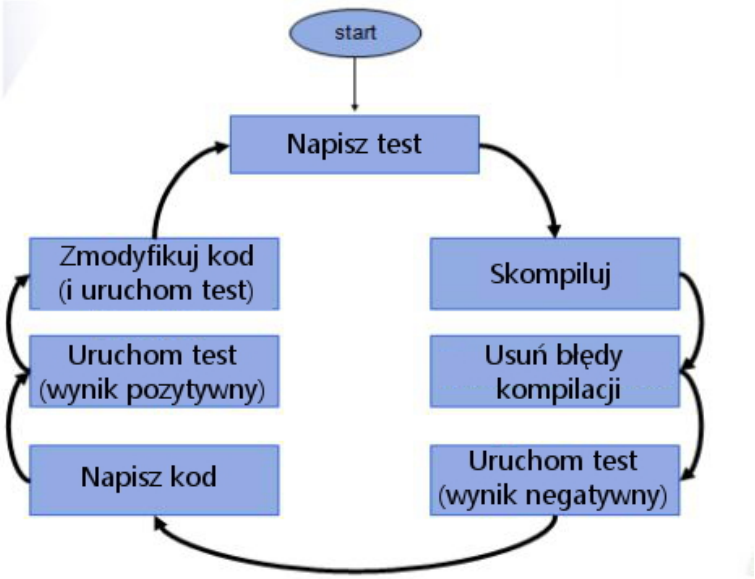

Rysunek 5.1: Zmodyfikowana struktura aplikacji Android: Clean Architecture

Nowy model aplikacji oddziela (przynajmniej w teorii) warstwy Application, Networking, Persistance oraz Model/Domain od środowiska Android SDK, a co za tym idzie nie trzeba umieszczać odwołania do tego środowiska praktycznie w każdym pliku (wyrażenie `import android.*`). Dopiero najwyższą warstwą jest warstwa Aplikacja Android. W takim podejściu należy użyć Androida jako pewnego rodzaju *pluginu* do tworzonej aplikacji i uniezależnić warstwę logiczną od reszty warstw. Czyli najpierw napisać aplikację, która będzie oddzielona od Android SDK (na przykład w czystej Javie), potem dołączyć do tego android SDK i

złączyć to wszystko w Aplikację Android. W ten sposób można być przekonanym, że kod, który stanowi podstawę naszej aplikacji, czyli jej logikę działania, zostanie przetestowany tak jak należy, czyli w oddzieleniu od reszty warstw.

### 5.1.2 Zastosowanie techniki *Test Driven Development* przy tworzeniu oprogramowania

Jak już autor wspomniał w rozdziale 3.5.1 dotyczącym metodologii TDD, wytwarzanie oprogramowania zaczynając od testowania powoduje, że zostanie napisanie tylko tyle aplikacji, ile to jest ujęte w wymaganiach. A co za tym idzie jej stopień komplikacji zostanie ograniczony do niezbędnego minimum. Zastosowanie tej techniki w przypadku aplikacji tworzonych pod Android z wykorzystaniem uporządkowanej architektury pozwoli na zwiększenie testowalności aplikacji.




Rysunek 5.2: Schemat kodowania w technice TDD

Korzyści, których można się spodziewać po zastosowaniu tej techniki są następujące:

- Wczesne wykrywanie błędów. Odporność na błędy regresyjne. Programista poprawia swoje pomyłki na bieżąco.
- Łatwiejszy *refactoring* kodu.
- Dobrze napisane testy stają się dokumentacją kodu.
- Lepiej zaprojektowane interfejsy. Technika projektowania testów za pomocą *user stories* zmusza do lepszego przemyślenia rozwiązań i dokładnego określenia zadań.
- Uproszczona integracja, łatwiejsze łączenie różnych fragmentów kodu poddanych wcześniej testom. Mniej błędów przedostaje się do etapu testów systemowych.

- Automatyzacja i powtarzalność *(contignous integration)*: testy można uruchamiać regularnie o określonych porach lub na pewnych etapach produkcji.
- Możliwość przetestowania funkcjonalności bez uruchamiania oprogramowania.

Wady *Test Driven Development* dotyczą głównie wykorzystania czasu w projekcie. Wymagany jest:

- Dodatkowy czas na stworzenie testów jednostkowych - deweloper potrzebuje czasem nawet o 40 procent więcej czasu na wykonanie tych samych zadań.
- Dodatkowy czas na utrzymanie testów - przy wprowadzaniu zmian w istniejącej funkcjonalności należy pamiętać, że potrzebny jest również czas na modyfikację istniejących testów jednostkowych.

### 5.1.3 Automatyzacja testów jednostkowych

Testy automatyczne powinny przyczyniać się do poprawy jakości dostarczając szybkiej - dużo szybszej niż w przypadku testowania manualnego - informacji zwrotnej na temat działania programu. Dodatkowo, szybka informacja zwrotna jest potwierdzeniem, że programista niczego nie uszkodził podczas modyfikowania gotowych funkcji. Automatyzacja testów nie ma sensu, jeżeli jej koszt jest większy niż korzyści, które przynosi. Z analizy rysunku 4.4 z rozdziału 4.5 można wnioskować, że największą korzyść przynosi automatyzacja testów jednostkowych. Są relatywnie łatwe do napisania, wykonują się w ułamkach sekund i są również łatwe do modyfikacji. Unit testy są solidną podstawą automatycznych testów regresyjnych.

### 5.1.4 Kod jako dokumentacja programu

Martin Robert Cecil[1] - popularny w środowisku programistycznym *Uncle Bob* - w swojej książce "Clean Code"[12], scharakteryzował dobrze napisany kod następująco:

- Czysty kod można odczytać.
- Czysty kod posiada przypisane testy jednostkowe.
- Czysty kod posiada znaczące i rozpoznawalne nazwy zmiennych, klas i funkcji.
- W czystym kodzie poszczególne klasy i podporządkowane im funkcje robią tylko jedną rzecz.
- Czysty kod posiada minimalne zależności, które są wyraźnie zdefiniowane i zapewniają czyste i minimalne API.
- Czysty kod nie powinien się duplikować.

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

- Czysty kod powinien być "literacki", ponieważ w zależności od języka nie wszystkie informacje mogą być rozumiane tak samo.
- Czysty kod może być łatwo rozbudowany przez inne osoby.
- Czysty kod powinien być złożony w takiej formie, aby był czytelny nie tylko dla maszyny, ale również dla człowieka.

Okazuje się jednak, że nie wystarczy raz napisać dobrze kod programu. Jeszcze ciężej jest go utrzymać w czystej formie podczas rozbudowy aplikacji. Warto więc, zdaniem autora, rozpatrzyć jedno z nowoczesnych podejść do uporządkowania architektury aplikacji, które może zwiększyć testowalność aplikacji pisanych dla systemu Android.

## 5.2 Różne podejścia do uporządkowania architektury

### 5.2.1 Architektura cebulowa - *The Onion Architecture*

Jeffrey Pallermo w połowie 2008 roku opublikował na swoim blogu serię artykułów[14], w których zaproponował inne od klasycznego podejście do warstw w aplikacji. Zauważył, że uzależnienie warstwy biznesowej od warstwy danych niekorzystnie wpływa na stabilność warstwy biznesowej, która z założenia powinna być niezależna od wprowadzania nowych technologii do warstwy bazodanowej. Stare podejście tego nie zapewniało. W większości przypadków warstwa biznesowa musiała być modyfikowana, a przynajmniej rekompilowana podczas modyfikacji warstwy dostępu do danych.

W zaproponowanym przez Palermo modelu, im warstwy położone są w architekturze głębiej, tym mniej podatne są na zmiany. Zależności mogą być zwrócone tylko w jedną stronę - do wnętrza "cebuli". W ten sposób w centrum znajdzie się model domeny, a na zewnątrz - warstwa bazodanowa. Jest to podejście nazywane *odwróceniem zależności*[2] i wymaga od programistów zastosowania odpowiednich technik projektowych, między innymi *wstrzykiwania zależności*[3].

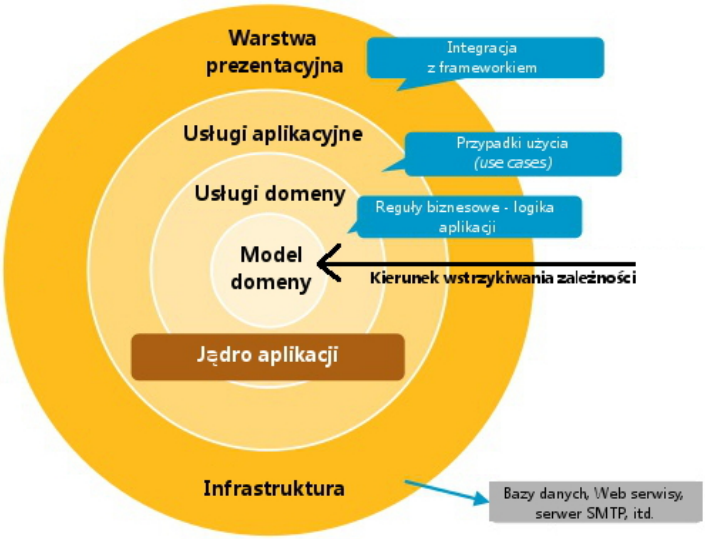
Warstwy wewnętrzne definiują interfejsy, za pomocą których mogą komunikować się z warstwami zewnętrznymi. Dzięki temu programiści nie muszą implementować na przykład dostępu z warstwy biznesowej do bazy danych w warstwie danych, zaimplementują tylko odpowiedni interfejs.

Kluczowe założenia architektury cebulowej:

- Aplikacja zbudowana jest według niezależnego modelu obiektowego.
- Warstwy wewnętrzne definiują interfejsy. Warstwy zewnętrzne te interfejsy implementują.

[2]Dependency Inversion Principle (DIP) - zasada mówiąca, że moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy powinny zależeć od abstrakcji. Abstrakcje natomiast nie powinny zależeć od szczegółowych rozwiązań, tylko odwrotnie.

[3]Dependency Injection (DI) – wzorzec projektowy i wzorzec architektury oprogramowania polegający na usuwaniu bezpośrednich zależności pomiędzy komponentami na rzecz architektury typu plug-in. Źródło: Wikipedia




Rysunek 5.3: *Onion architecture* według Jeffrey'a Palermo. Źródło: `http://jeffreypalermo.com/blog/the-onion-architecture-part-1`

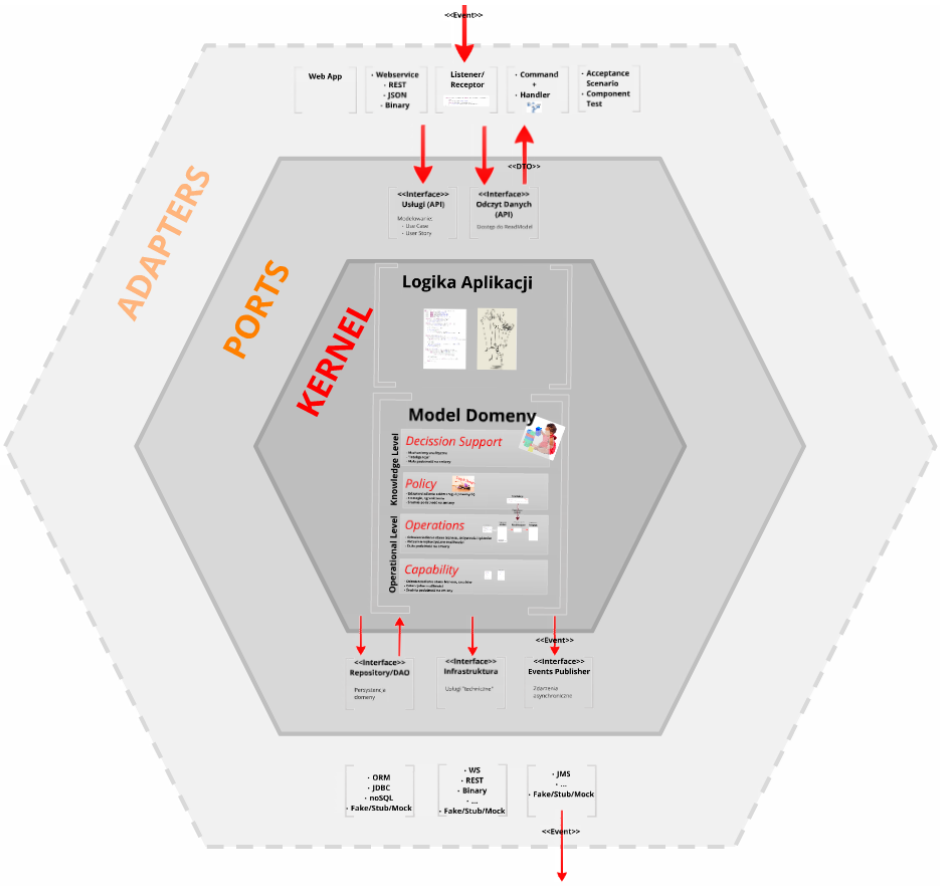
- Kierunek *sprzężenia* jest od warstwy zewnętrznej do wewnętrznej.
- Cały kod rdzenia aplikacji (*Object Model, Object services, Application Services*) może zostać skompilowany i wykonany bez kontaktu z infrastrukturą i interfejsem użytkownika.

### 5.2.2 Architektura portów i adapterów - *Ports and Adapters Architecture*

Opracowana w 2005 roku przez Alistaira Cockburna[4] architektura niegdyś znana była pod nazwą architektury heksagonalnej[6]. Nazwa *"heksagonalna"* przyjęła się, ponieważ kiedyś znane były tylko trzy porty wejściowe i trzy porty wyjściowe, stąd wszystkie schematy rysowane były jako foremny sześciokąt. Wraz ze wzrostem liczby portów wejściowych i wyjściowych zmieniono nazwę na *Ports & Adapters*, nie zmieniono natomiast zasady przedstawiania architektury na schematach, co przedstawia rysunek 5.4.

Idea jest bardzo podobna do architektury cebulowej: w centrum znajduje się jądro aplikacji, czyli model domeny i logika biznesowa, odpowiedzialna za kluczowe działania programu. Jądro otoczone jest przez warstwę portów, zawierającą porty zarówno wejściowe, jak i wyjściowe. Zewnętrzną warstwą jest warstwa adapterów. W szczegółach przedstawia się to następująco:

[4]Alistair Cockburn - jeden z inicjatorów ruchu Agile. Jest współautorem wydanego w 2001 roku manifestu Agile. W roku 2005 pomagał współtworzyć deklarację 'PM Declaration of Interdependence'. Propagator przypadków użycia jako dokumentacji procesów biznesowych oraz wymagań co do zachowania oprogramowania. Źródło: Wikipedia




Rysunek 5.4: Architektura heksagonalna. Opracowanie schematu: Sławomir Sobótka. Źródło: `https://prezi.com/p0psif9qixgz/ports-adapters`

**Jądro - *Kernel*** Jądro oferuje model domeny i logikę biznesową: ogólne możliwości aplikacji, operacje, logikę operacji oraz mechanizmy wspierania decyzji, co już w wymienionej kolejności stanowi architekturę warstwową. Warstwa również może być podzielona na kilka poziomów logiki.

**Porty - *Ports*** Porty są to interfejsy usług: API, odczyt danych (wejściowe), oraz interfejsy repozytoriów czy DAO[5], sterowniki do różnego rodzaju maszyn, urządzeń, odbiorników GPS czy radia (wyjściowe). Za pomocą portów wyjściowych warstwa jądra jest w stanie emitować zdarzenia. Zdarzenia są techniką odwracania kontroli, czyli działają z zachowaniem wspomnianej już zasady, że warstwa wewnętrzna nie posiada kontroli nad warstwą zewnętrzną.

**Adaptery - *Adapters*** Szczególnym przykładem adaptera jest aplikacja webowa, która z jednej strony komunikuje się z serwerem baz danych, a z drugiej z cienkim klientem,

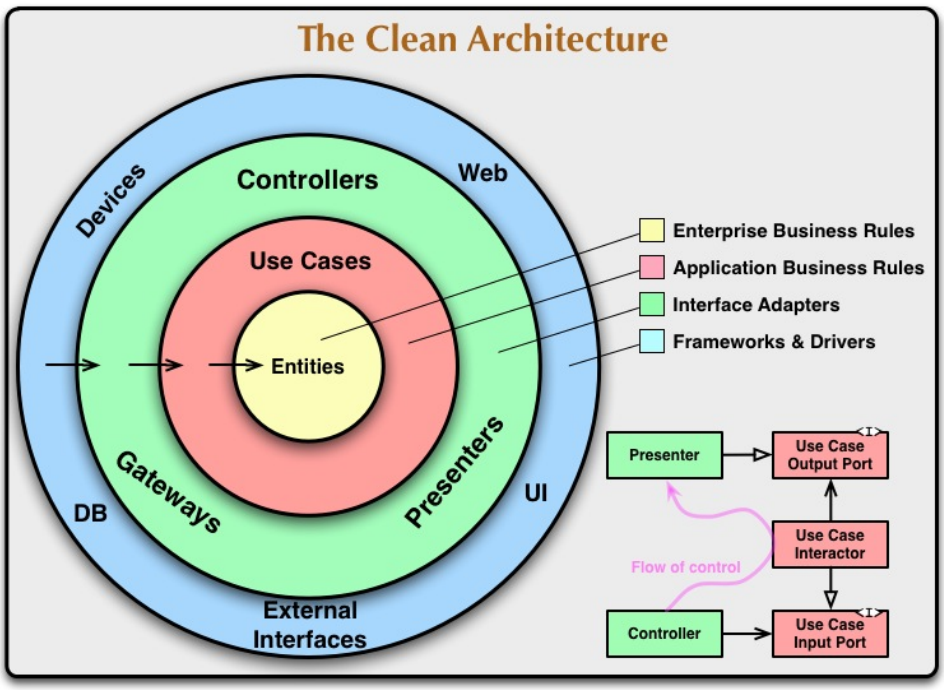
[5]Data Access Object – komponent dostarczający jednolity interfejs do komunikacji między aplikacją a źródłem danych.

jakim jest przeglądarka internetowa po stronie użytkownika. Może odbierać również zdarzenia od innych modułów systemowych, jeżeli architektura systemowa jest modułowa (Listener/Receptor zdarzeń). Głównym zadaniem adapterów jest delegowanie zdarzeń do portów na warstwie położonej wewnątrz i powinno się w miarę możliwości unikać przypisywania adapterom większej ilości zadań.

Z punku widzenia systemu napisanego w architekturze cebulowej, testowalność oprogramowania wygląda następująco: testy jednostkowe penetrują jądro, testy integracyjne penetrują porty, a testy systemowe penetrują kilka jąder na raz.

### 5.2.3 Architektura uporządkowana - *The Clean Architecture*

Cytowany już w rozdziale 5.1.4 Uncle Bob opublikował w 2012 roku na swoim blogu propozycję usystematyzowania kodu Android, która zainteresowała również autora tej pracy. Właśnie ten typ architektury zostanie wykorzystany przez autora w części 6 do wykazania wzrostu testowalności aplikacji napisanej dla Androida w "czystej architekturze" w porównaniu do aplikacji napisanej w sposób "tradycyjny", przedstawiony w rozdziale 4. Schemat przedstawiony jest na rysunku 5.5




Rysunek 5.5: 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 bardziej adekwatną jest nazwa *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 ma powiązania z frameworkiem Android.

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

Warstwy na schemacie połączone są strzałkami, które informują o kierunku przekazywania informacji. Ze schematu wynika, że warstwa *Entities* nie posiada informacji 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*, a przypadki użycia mogą korzystać z wiedzy o całej warstwie logicznej.

Przekładając powyższe na język programistów, jeżeli używamy instrukcji `import android.*`, to tylko w stosunku do wyższej warstwy. Nie wolno tego robić w stosunku do warstwy niższej, bo wtedy cała koncepcja może ulec dezintegracji.

W ten sposób teoretycznie można stworzyć architekturę, która:

- jest niezależna od frameworka (tutaj Android SDK) i zachowana zostaje zasada zależności,

- jest niezależna od textitUI, 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 do wykonania.

Jeżeli struktura tworzonej aplikacji zostanie ułożona w powyższy sposób – testowanie (przynajmniej jednostkowe) nie powinno być bardziej skomplikowane niż w przypadku kodu napisanego w czystej Javie czy C++.

## 5.3 Praca z kodem zastanym

W rozdziale 4.7 autor nawiązał do problemu pracy z kodem zastanym, tak zwanym *Legacy Code*. Co więc zrobić, aby zaimplementować testy do już napisanego kodu? Godfrey Nolan[6] w swojej książce "Agile Android"[13] proponuje następujące rozwiązanie:

[6]Godfrey Nolan - założyciel i prezes RIIS LLC, firmy zajmującej się rozwojem oprogramowania dla platform przenośnych. Autor książek o Androidzie: "Agile Android", "Booletproof android", "Android Best Practices" i "Decompiling Android"

- Wprowadzić metodę ciągłej integracji w procesie budowania kodu - (*Continuous Integration (CI)*).

  Jest to należąca do metodologii zwinnych praktyka polegająca na regularniej integracji zmian w kodzie do bieżącego repozytorium. Warunkiem koniecznym umieszczenia kodu w repozytorium jest upewnienie się, że dany kod działa.

- Przy projektowaniu nowych funkcjonalności używać metody TDD.

- Skorzystać z serwera CI (dobrym przykładem jest tutaj *Jenkins*[7]), który będzie wykonywał unit testy, do których mamy przekonanie, że ich implementacja nie powinna się zmieniać. Oczywiście tylko na tym etapie, bo testy automatyczne muszą również podlegać przeglądom.

- Uświadomić zespół programistów na czym polegają testy jednostkowe i przekonać ich do stosowania TDD.

- Dodać metryki mierzenia kodu do CI. Ustawić poziom minimalny na 10-15%.

- Użyć frameworka do podstawowych testów GUI już istniejącej aplikacji (najlepiej *Espresso*). Stworzyć testy opierając się na przypadkach użycia (ang. *use cases).*

- Bezwzględnie pisać testy jednostkowe do nowych części oprogramowania zgodnie z TDD, *mockując* jeśli to możliwe wykorzystywane obiekty z zastanego kodu.

- Wyizolować zastany kod, tak aby nikt z programistów nie miał do niego dostępu, jeżeli naprawdę nie jest to niezbędne.

- Usunąć niewykorzystywane i nieużyteczne części kodu.

- Przepisać i przetestować wyizolowany kod, aby zwiększyć metryki pokrycia do około 60-70%.

Najważniejsze więc jest wyizolować stary kod, przetestować aplikację za pomocą frameworka do testów GUI, usunąć ewentualne błędy nie pozwalające na poprawną pracę programu i nie modyfikować wyizolowanego kodu podczas pisania nowych funkcjonalności. Nowe funkcje należy dodawać stosując już taktykę *Test Driven Development*. Dopiero na końcu, jeżeli tworzone środowisko jest już stabilne, należy rozpocząć przebudowę starych części aplikacji, tak aby metryki pokrycia kodu rosły wraz z upływem czasu. Przy wykonywaniu *refaktoringu* Godfrey Nolan proponuje użyć narzędzia SonarQube[8].

[7] http://jenkins-ci.org/

[8] SonarQube - platforma do prowadzenia ciągłej inspekcji jakości kodu źródłowego, dostarczana na licencji *open source*

# Rozdział 6

# Porównanie testowalności na przykładzie aplikacji

## 6.1 Opis doświadczenia

Doświadczenie polega na przeanalizowaniu przykładowej aplikacji dla Systemu Android zaprojektowanej na dwa sposoby. Wersja pierwsza to aplikacja napisana w standardowej architekturze (nazywana dalej *wersją pierwotną*), wersja druga to ten sam program napisany przy wykorzystaniu *Clean Architecture* (nazywana dalej *wersją poprawioną*). Do tego celu zdecydowano się wykorzystać aplikację *JSON Web Token Authentication for Android* napisaną przez Victora Albertosa [**?**] , a której źródła udostępnione są w serwisie GitHub na licencji *open source*.

W części doświadczalnej badania ograniczono do analizy testowalności w zakresie testów jednostkowych i wczesnych testów integracyjnych pomijając przegląd na etapie testów systemowych czy akceptacyjnych z powodu braku dostępu do wymagań systemowych i wymagań klienta. Jednakże już na podstawie analizy testów jednostkowych i wstępnego testu integracyjnego można z dużym prawdopodobieństwem ocenić, czy zastosowanie TDD i architektury *Clean Architecture* poprawi testowalność programu i spowoduje ułatwienie dalszego procesu testowego.

## 6.2 Opis aplikacji

*JSON Web Token Authentication for Android* poświadcza prawdziwość użytkowników Androida i iOS korzystając z *REST API* serwera *Parse* oraz textitJSON[1] Web Tokens (JWT). JWT to otwarta, według standardu przemysłowego RFC 7519[3], metoda do uwierzytelniania stron w środowisku aplikacji internetowych. Wykorzystywana jest do przekazywania tożsamości użytkowników między dostawcą a odbiorcą usług internetowych, lub innego typu uwierzytelnień zgodnie z logiką biznesową.

[1]JSON, JavaScript Object Notation – lekki format wymiany danych komputerowych. JSON jest formatem tekstowym, bazującym na podzbiorze języka JavaScript. Źródło: Wikipedia.

Serwer *Parse* to wspólna platforma do przechowywania danych i interakcji z usługami internetowymi dla aplikacji mobilnych. Wielu programistów decyduje się na wykorzystanie tej platformy sprawdzonej, zamiast pisać własne rozwiązania w tym zakresie.

Victor Albertos zdecydował się użyć wyżej wymienionego rozwiązania, aby zwiększyć pielęgnowalność swojej aplikacji: móc modyfikować logikę biznesową nie modyfikując oprogramowania po stronie klienta. Serwer Parse, jako rozwiązanie uniwersalne, zapewniał takie podejście i wpisywał się doskonale w koncepcję autora *JSON Web Token Authentication for Android*.

## 6.3 Zasada działania

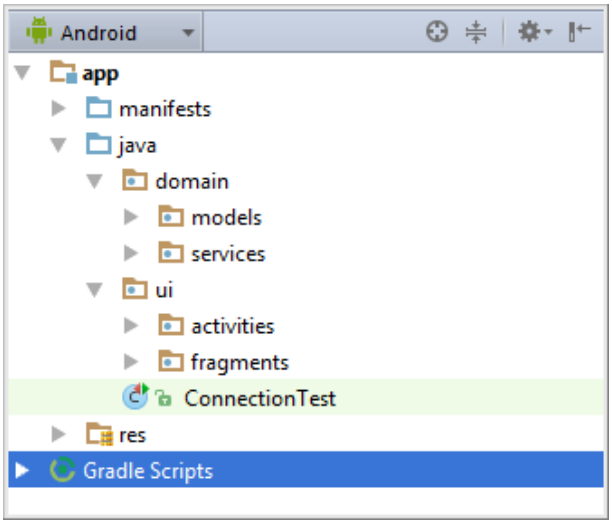
Użytkownik loguje się z urządzenia z zainstalowanym systemem Android za pomocą swojej nazwy użytkownika i hasła do serwera *Parse*. Jeżeli nie posiada jeszcze konta na serwerze, może założyć je bezpośrednio z używanego programu. Jeżeli użytkownik istnieje, od chwili zalogowania może korzystać z dostępnych mu usług serwera, a logując się z wielu urządzeń korzysta z wielosesyjności systemu. Użytkownik może również akualizować swoje dane na serwerze.

## 6.4 Analiza aplikacji pod względem testowalności

W tej pracy autor skupia się na budowie aplikacji tylko z punktu widzenia testowania aplikacji, nie będzie więc analizowany szczegółowo kod programu. Przeprowadzone natomiast zostaną testy, na podstawie których można w wystarczającej części ocenić, czy zmiana struktury oprogramowania na *Clean Architecture* oraz zastosowanie *Test Driven Development* wpłynie pozytywnie na testowalność oprogramowania, w tym przypadku tej wybranej aplikacji.

### 6.4.1 Budowa analizowanej aplikacji w wersji pierwotnej

Schemat budowy aplikacji *JSON Web Token Authentication for Android* w wersji pierwotnej przedstawia się tak, jak to pokazano na rysunku 6.1.




Rysunek 6.1: Schemat budowy badanej aplikacji wykorzystującej architekturę standardową

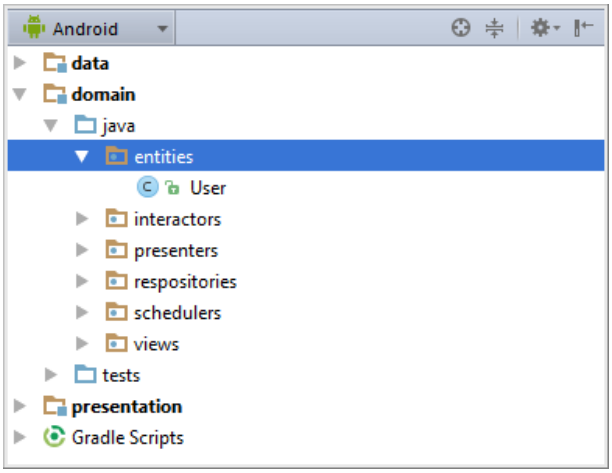
Aplikacja podzielona jest na moduł domeny i moduł interfejsu użytkownika. W celu przetestowania aplikacji, utworzono jeden test całkowicie zależny od środowiska Android, który w zależności od kontekstu można nazwać integracyjnym, systemowym, bądź nawet końcowym. Dla potrzeb pracy traktowany jest jako test integracyjny, gdyż interesuje autora tylko wynik weryfikacji połączenia, bez zwracania uwagi na aspekty graficzne czy użytkowe wiązane z docelowym urządzeniem. Test polega na zalogowaniu się do programu za pomocą przykładowych danych i otrzymaniu wyniku, czy weryfikacja przebiegła pomyślnie. Poza tym przypadkiem nie stwierdzono żadnych innych testów, w tym jednostkowych.

### 6.4.2 Budowa analizowanej aplikacji w wersji poprawionej

Każde z wymienionych w rozdziale 5 podejść do uporządkowania architektury systemowej można wykorzystać do polepszenia testowalności aplikacji tworzonych dla systemu Android. Analizując opisywane rozwiązania krok po kroku można dojść do wniosku, że ich idea jest taka sama, a różnią się jedynie szczegółami. W tym wypadku autor *JSON Web Token Authentication for Android* zdecydował się na *The Clean Architecture*, zaprezentowaną w tej pracy w rozdziale 5.2.3.

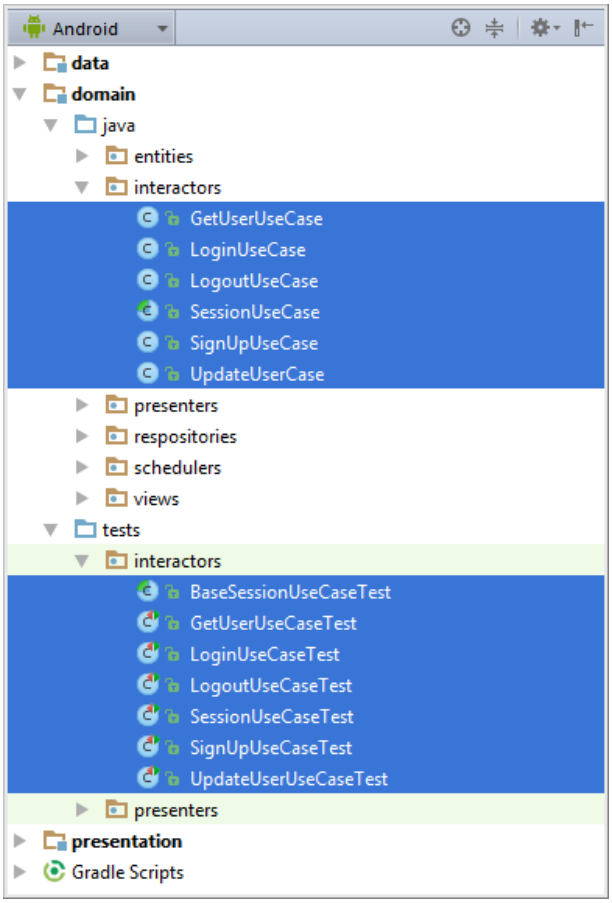
Aplikacja w wersji poprawionej nadal ma budowę modułową - tym razem moduły są trzy: moduł domeny, moduł danych i moduł prezentacyjny. Zgodnie z nowym podejściem (patrz rysunek 5.5 można rozpoznać poszczególne warstwy uporządkowanej architektury:

- Warstwę *entities*, w której znajduje się definicja klasy *User* oraz klasy *Credentials*. Należą one do logiki biznesowej aplikacji i to na tych klasach oparte jest działanie programu.



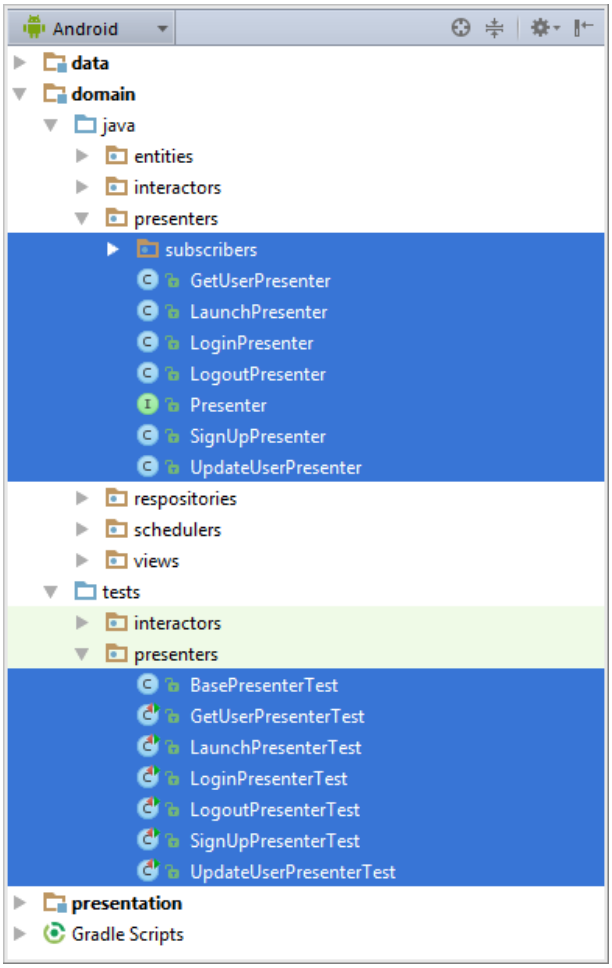

Rysunek 6.2: Schemat budowy aplikacji w wersji poprawionej: warstwa *entities*.

- Warstwę *Use Cases*, należącą do modułu domeny, do której na podstawie przypadków użycia zostały napisane testy jednostkowe (patrz rysunek 6.3)




Rysunek 6.3: Schemat budowy aplikacji w wersji poprawionej: warstwa *use cases*.

- Warstwę *Presenters*, również należącą do modułu domenowego i odpowiadające jej testy jednostkowe (rysunek 6.4).




Rysunek 6.4: Schemat budowy aplikacji w wersji poprawionej: warstwa *presenters*.

- Moduły *data* oraz *presentation*, to najbardziej zewnętrzna warstwa, zależna od wszystkich wyżej opisanych.

Zgodnie z ideą uporządkowanej architektury, warstwa *entities* nie posiada informacji o warstwach *use cases*, przypadki użytkownika nie wiedzą nic o warstwie prezentacyjnej, a ta nie ma danych na temat warstw zewnętrznych, takich jak wspomniane bazy danych. Kierunek wstrzykiwania zależności jest zawsze od warstwy zewnętrznej do warstwy wewnętrznej, co pokazuje rysunek 5.5.

## 6.5 Przebieg doświadczenia

Podczas doświadczenia wykonano zaprojektowane przez autora aplikacji testy dla obu wersji programu, wykorzystując to samo środowisko testowe [2]: Android Studio w wersji 1.5.1, ten sam sprzęt i ten sam emulator: Nexus_5_API_23.

## 6.6 Wyniki doświadczenia

### 6.6.1 Wyniki dla testów jednostkowych

Wyniki doświadczenia dla testów jednostkowych przedstawiają się następująco:

W przypadku aplikacji w wersji poprawionej testy jednostkowe istnieją dla warstw *use cases* oraz *presenters* Rezultat wykonania przedstawiają tabele 6.1, 6.2 i 6.3.

Tabela 6.1: Zestawienie testów jednostkowych: podsumowanie dla warstw *Use Cases* oraz *Presenters*.

| **Warstwa** | **Przetestowane klasy** | **Przetestowane funkcje** | **Pokrycie linii kodu** |
|---|---|---|---|
| *Use Cases* | 100% (6/6) | 100% (16/16) | 100% (36/36) |
| *Presenters* | 100% (15/15) | 81% (35/43) | 85% (85/99) |

Tabela 6.2: Zestawienie testów jednostkowych: podsumowanie szczegółowe dla warstwy *Use Cases*.

| **Warstwa** | **Przetestowane klasy** | **Przetestowane funkcje** | **Pokrycie linii kodu** |
|---|---|---|---|
| *GetUserUseCase* | 100% (1/1) | 100% (2/2) | 100% (3/3) |
| *LoginUseCase* | 100% (1/1) | 100% (3/3) | 100% (6/6) |
| *LogoutUseCase* | 100% (1/1) | 100% (2/2) | 100% (3/3) |
| *SessionUseCase* | 100% (1/1) | 100% (3/3) | 100% (14/14) |
| *SignUpUseCase* | 100% (1/1) | 100% (3/3) | 100% (5/5) |
| *UpdateUserCase* | 100% (1/1) | 100% (3/3) | 100% (5/5) |

W aplikacji w wersji pierwotnej nie zaprojektowano testów jednostkowych w ogóle, a testując funkcjonalność programu zdano się w całości na automatyczny test integracyjny.

[2] Środowisko testowe (ang. test environment) - środowisko, w skład którego wchodzi sprzęt, wyposażenie, symulatory, oprogramowanie oraz inne elementy wspierające, potrzebne do wykonania testu. [wg IEEE 610]

Tabela 6.3: Zestawienie testów jednostkowych: podsumowanie szczegółowe dla warstwy *Presenters*.

| **Warstwa** | **Przetestowane klasy** | **Przetestowane funkcje** | **Pokrycie linii kodu** |
|---|---|---|---|
| *subscribers* | 100% (3/3) | 80% (8/10) | 91% (22/24) |
| *GetUserPresenter* | 100% (2/2) | 80% (4/5) | 81% (9/11) |
| *LaunchPresenter* | 100% (2/2) | 83% (5/6) | 85% (12/14) |
| *LoginPresenter* | 100% (2/2) | 80% (4/5) | 81% (9/11) |
| *LogoutPresenter* | 100% (2/2) | 83% (5/6) | 85% (12/14) |
| *SignUpPresenter* | 100% (2/2) | 80% (4/5) | 81% (9/11) |
| *UpdateUserPresenter* | 100% (2/2) | 83% (5/6) | 85% (12/14) |

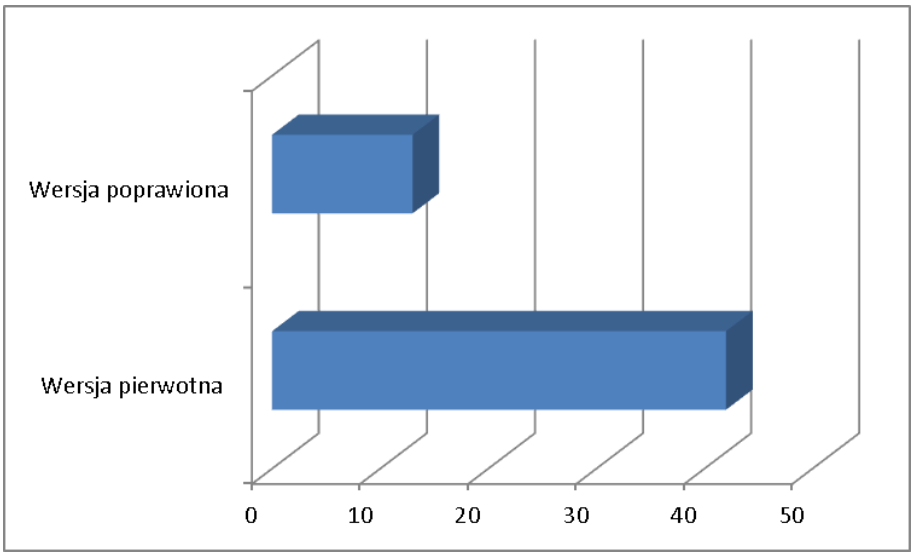
W przypadku aplikacji w wersji poprawionej, jak przedstawiają tabele 6.2 i 6.3, zaprojektowano 6 testów opartych na przypadkach użycia oraz 7 testów dla prezenterów, co daje łącznie

$$6 + 7 = 13$$

testów jednostkowych. W przypadku aplikacji w wersji pierwotnej - zgodnie z wyjaśnieniem, które autor umieścił w rozdziale 4.4 - aby pokryć ten sam obszar funkcjonalności należałoby zaprojektować

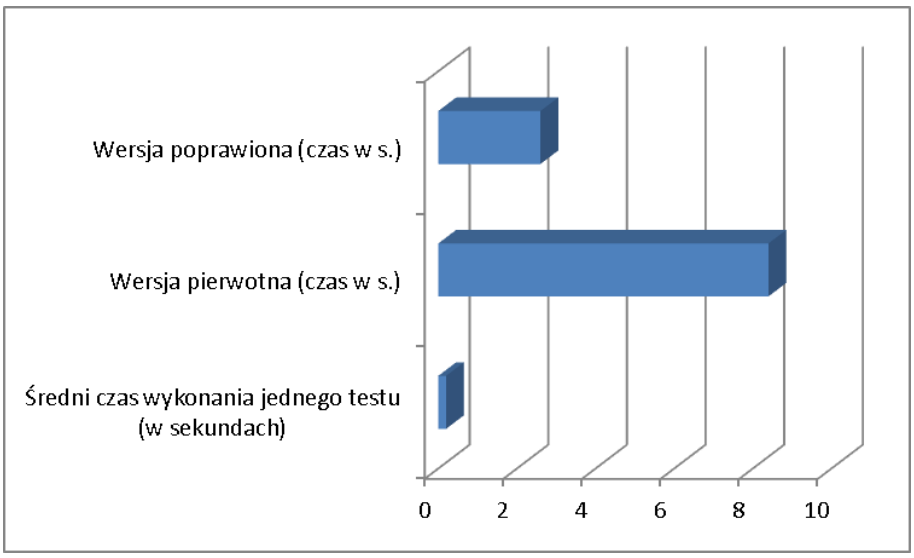
$$6 * 7 = 42$$

testy jednostkowe. Różnicę pokazuje wykres 6.5.




Rysunek 6.5: Porównanie ilości testów jednostkowych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej.

Taka liczba testów musiała się również odbić na ich czasie wykonania, co w przypadku badanej aplikacji przedstawia się jak na wykresie 6.6:



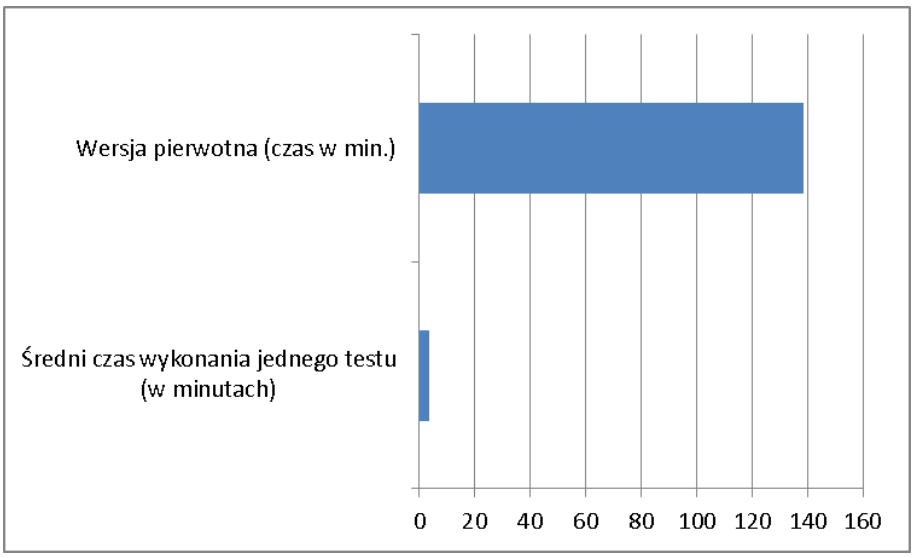

Rysunek 6.6: Porównanie czasu wykonania testów jednostkowych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej.

### 6.6.2 Testy integracyjne

Jak już wspomniano wcześniej, w przypadkach obu wersji oprogramowania zaprojektowano test integracyjny *ConnectionTest*. Dla obu opisywanych przypadków przewidziano taki sam zakres testowania: generowany jest zestaw kilku użytkowników dla których przeprowadzano próbę połączenia. Czas trwania całego testu od uruchomienia do otrzymania wyniku pozytywnego bądź negatywnego (wraz z uruchomieniem emulatora systemu Android w wybranym do badań środowisku testowym) wyliczony w doświadczeniu wyniósł około 200 sekund.

W przypadku gdy dla aplikacji w wersji pierwszej nie stwierdzono testów jednostkowych, to

- zakładając, że każdy z sześciu testów jednostkowych opartych na przypadkach użycia dla aplikacji w wersji poprawionej znalazłby jeden błąd, liczba powtórzeń testu *ConnectionTest* w najgorszym razie mogłaby wzrosnąć sześciokrotnie;
- zakładając, że dodatkowo każdy z siedmiu testów jednostkowych przeznaczonych dla prezenterów również znalazłby błąd, ilość powtórzeń testu *ConnectionTest* w najgorszym przypadku mogłaby wzrosnąć 42-krotnie do momentu uzyskania wyniku pozytywnego (patrz wykres 6.7).




Rysunek 6.7: Porównanie czasu wykonania testów integracyjnych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej.

## 6.7 Wnioski końcowe

Powyższe doświadczenie pokazuje, że wykorzystanie architektury *Clean Architecture* oraz podejścia *Test Driven Development* wydaje się właściwe dla polepszenia testowalności badanej aplikacji zarówno w przypadku, gdy w wersji pierwotnej nie napisano w ogóle testów jednostkowych, jak i gdyby je zaprojektowano dla takiego samego obszaru testowania, co w przypadku programu w wersji poprawionej. Korzyści z zastosowania danych rozwiązań pokazują wykresy 6.5 i 6.6.

Udowodniono, że nawet jeżeli testy jednostkowe w aplikacji pierwotnej zostałyby napisane, do przetestowania tego samego obszaru funkcjonalności ich liczba musiałaby być zdecydowanie większa od liczby ich odpowiedników w przypadku aplikacji poprawionej. Wymagałoby to odpowiednio większego nakładu pracy pielęgnacyjnej przy ewentualnej modyfikacji programu. Zmiana kodu programu w zakresie rozszerzenia funkcjonalności również wymagałaby napisania większej ilości dodatkowych testów jednostkowych.

Gdyby testów jednostkowych, tak jak w badanym przypadku, nie było, wtedy potrzeba zwiększenia czasu przeznaczonego na testy integracyjne i wyższego szczebla jest jeszcze większa, co pokazuje wykres 6.7.

Przeprowadzone doświadczenie dowodzi, że warto wykorzystać opisywane w rozdziale 5 metody do zwiększenia testowalności, a także - jak się okazuje - pielęgnowalności aplikacji przeznaczonych dla systemu Android.

# Rozdział 7

# Wnioski

# Bibliografia


[1] Gnu general public license, version 1, 1989.

[2] Android sdk - wyjaśnienie z polskiej wikipedii, 2015.

[3] Standard rfc 7519, May 2015.

[4] Agile Aliance. Guide to agile practices, 2013.

[5] Antyweb. Antyweb.pl, 2015.

[6] Alistair Cockburn. Hexagonal architecture, April 2005.

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

[8] Fermentas Inc. Introduction to android, February 2016.

[9] Grupa Robocza ISTQB. Certyfikowany tester, plan poziomu podstawowego, 2011.

[10] Collin Mulliner Joshua J. Drake, Zach Lanier. *Android Hacker's Handbook*. Wiley, Indianapolis, wydanie pierwsze, 2014.

[11] Rick Kazman Len Bass, Paul Clements. *Software Architecture in Practice*. Addison-Wesley, New York, wydanie trzecie, 2013.

[12] Robert Cecil Martin. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, New York, wydanie pierwsze, 2009.

[13] Godfrey Nolan. *Agile Android*. Apress, New York, wydanie pierwsze, 2015.

[14] Jeffrey Palermo. The onion architecture: part 1-4, July 2008.

[15] Diego Torres Milano Paul Blundell. *Learning Android Application Testing*. Packt Publishing, Birmingham, wydanie drugie, 2015.

[16] Cameron Watson. V-model opisany przez cameron watson, 2015.

# Spis rysunków


2.1 Android – udział w rynku urządzeń mobilnych w Polsce. (Źródło: portal AntyWeb.pl, 05/2015) . . . . . 6
2.2 Android – udział w rynku urządzeń mobilnych na świecie. (Źródło: portal android.com.pl, 09/2015) . . . . . 7
2.3 Statystyki dotyczące używania poszczególnych wersji systemu Android przez użytkowników (Źródło: portal androidnow.pl, 09/2015). Ostatnio doszła wersja 6.0, ale jej udział na rynku urządzeń w obecnej chwili jest znikomy. . . . 8

3.1 Cykl życia *Activity*. Źródło: Dokumentacja Android . . . . . 12
3.2 Model V - najpopularniejszy model zarządzania projektem informatycznym (*Vmodel*). Źródło: [16] . . . . . 14

4.1 Przegląd architektury Android. Źródło: Karim Yaghmour of Opersys Inc., `http://http://www.slideshare.net/opersys/inside-androids-ui` . . . 22
4.2 Podejście „standardowe" przy tworzeniu aplikacji dla systemu Android . . . 23
4.3 Różnice w testowaniu klas osobno i razem. *"Integrated Tests are a Scam"* - schemat autorstwa J.B. Rainsbergera. Źródło: `https://vimeo.com/80533536)` 24
4.4 Idealna piramida testowania według Mike'a Cohna[7]. Źródło: `http://www.scrumdo.pl/2015/08/piramida-testow-agile.html` . . . . . 25
4.5 Podejście klasyczne do testów, czyli ddwrócona piramida testowania (*Ice Cream AntiPatern*). Źródło: `http://www.scrumdo.pl/2015/08/piramida-testow-agile.html` . . . . . 26

5.1 Zmodyfikowana struktura aplikacji Android: Clean Architecture . . . . . 29
5.2 Schemat kodowania w technice TDD . . . . . 30
5.3 *Onion architecture* według Jeffrey'a Palermo. Źródło: `http://jeffreypalermo.com/blog/the-onion-architecture-part-1` . . . . . 33
5.4 Architektura heksagonalna. Opracowanie schematu: Sławomir Sobótka. Źródło: `https://prezi.com/p0psif9qixgz/ports-adapters` . . . . . 34
5.5 Clean architecture of Android według Uncle Ben. Źródło: `http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html` . . . . . 35

6.1 Schemat budowy badanej aplikacji wykorzystującej architekturę standardową 40
6.2 Schemat budowy aplikacji w wersji poprawionej: warstwa *entities*. . . . . 41
6.3 Schemat budowy aplikacji w wersji poprawionej: warstwa *use cases*. . . . . 41
6.4 Schemat budowy aplikacji w wersji poprawionej: warstwa *presenters*. . . . . 42

6.5 Porównanie ilości testów jednostkowych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej. . . . . 44
6.6 Porównanie czasu wykonania testów jednostkowych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej. . . . 45
6.7 Porównanie czasu wykonania testów integracyjnych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej. . . . 46

# Spis tabel

3.1 Koszty znalezienia błędu na poszczególnych etapach projektu . . . . . 11

6.1 Zestawienie testów jednostkowych: podsumowanie dla warstw *Use Cases* oraz *Presenters*. . . . . 43
6.2 Zestawienie testów jednostkowych: podsumowanie szczegółowe dla warstwy *Use Cases*. . . . . 43
6.3 Zestawienie testów jednostkowych: podsumowanie szczegółowe dla warstwy *Presenters*. . . . . 44