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, 25.03.2016

Spis treści

Abstrakt

TESTOWALNOŚĆ APLIKACJI MOBILNYCH NA PLATFORMĘ ANDROID STRESZCZENIE

Jako system operacyjny dostępny bezpłatnie, Android zrzesza ogromną społeczność programistów piszących aplikacje, które poszerzają funkcjonalność urządzeń mobilnych. Okazuje się jednak, że temat automatycznego testowania opracowywanych aplikacji jest często pomijamy w literaturze, a stosowane powszechnie podejście do tworzenia nowych aplikacji nie pozwala na sprawne pisanie testów jednostkowych. W konsekwencji programiści rozpoczynają testowanie od testów integracyjnych. Wychodząc tym problemom naprzeciw, autor w tej pracy dowodzi, że zastosowanie odpowiedniej architektury oraz odpowiedniego podejścia do pisania oprogramowania może ułatwić testowanie aplikacji napisanych dla tego systemu.

Poniższa praca zawiera analizę testowalności aplikacji na platformę Android. Wprowadzone i wyjaśnione zostają podstawowe pojęcia związane z programowaniem na tę platformę oraz testowalnością oprogramowania, a następnie opisany proces projektowy, który znacząco ułatwia utrzymanie i opracowywanie aplikacji Android. Porównane zostają również wybrane rozwiązania architektoniczne w kontekście automatycznego testowania. Otrzymane wyniki pozwalają na wyciągnięcie ciekawych wniosków.

Słowa kluczowe: testowalność, Android, Clean Architecture, Test Driven Development

TESTABILITY OF MOBILE APPLICATIONS FOR ANDROID ABSTRACT

As the operating system available for free, Android joins a huge community of developers writing applications that extend the functionality of mobile devices. However, it turns out that subject of automatic testing of Android applications is often omitted in the literature, and commonly used approach to develop new applications doesn't help for unit tests writing. As a result, developers often use the integration test instead. Going into the problem, the author demonstrates that using of appropriate architecture and approach to writing software can facilitate the testing of applications written for this system.

This publication presents an analysis of applications testability on Android platform. The author introduces and explains the basic concepts related to programming on the platform, as well as software testability. In the second part, the design process, which significantly facilitates the maintenance and development of Android applications, is described. Also selected architectural solutions are compared in the context of automated testing. The results allow to draw many interesting conclusions.

Key words: testability, Android, Clean Architecture, Test Driven Development

Rozdział 1

Wprowadzenie

Pojęcie *android* 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 codziennie przez ludzi na całym świecie. Android to również nazwa firmy, system operacyjny oraz projekty *Open Source*[1]. Jako system operacyjny dostępny bezpłatnie, Android zrzesza ogromną społeczność programistów piszących aplikacje, które poszerzają funkcjonalność tych urządzeń. Dostarcza programistom bogate środowisko użytkownika pozwalając pisać oprogramowanie bez konieczności sięgania do niższych warstw systemu, włączając możliwość dostępu do Internetu, serwerów baz danych, drukarek i innych urządzeń peryferyjnych. W pierwszym kwartale 2016 roku w internetowym sklepie Google Play (wcześniej Android Market)[2], dostępnych było ponad 1,9 miliona aplikacji.

Mimo tak dużej popularności platformy Android oraz licznych publikacji dotyczących programowania na nią, okazuje się, że temat automatycznego testowania opracowywanych aplikacji jest często pomijamy w literaturze. Okazuje się również, że stosowane powszechnie podejście do tworzenia nowych aplikacji nie pozwala na sprawne pisanie testów jednostkowych [6]. W konsekwencji programiści często korzystają z testów integracyjnych, gdzie nakład pracy jest dużo większy. Dodatkowo, pozbawiając się możliwości zastosowania testów jednostkowych na wczesnym etapie projektu z powodu źle przemyślanej struktury aplikacji, projektanci ryzykują utratę jakości, a co za tym idzie - utratę zaufania klientów do oprogramowania. Wychodząc tym problemom naprzeciw, autor w tej pracy próbuje dowieść, że zastosowanie odpowiedniej architektury oraz odpowiedniego podejścia do pisania oprogramowania może ułatwić testowanie aplikacji napisanych dla tego systemu. Stąd tematem pracy jest "Testowalność aplikacji mobilnych na platformę Android", a celem - przebadanie różnych podejść do architektury aplikacji z przeznaczeniem dla systemu Android i sprawdzenie jak one wpływają na testowalność tego systemu.

[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 [26].

[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 [26].

Praca została podzielona na dwie części. W części pierwszej autor prezentuje 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 programowej oraz opisuje aktualnie stosowane podejście przy projektowaniu aplikacji Android. Następnie naświetlone zostają trudności, jakie napotyka się podczas testowania napisanego w ten sposób oprogramowania.

W części drugiej, praktycznej, zostaje przedstawiona 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* autorstwa Roberta Cecila Martina. 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* oraz wspomnianej *The Clean Architecture*.

Rozdział 2

Platforma Android

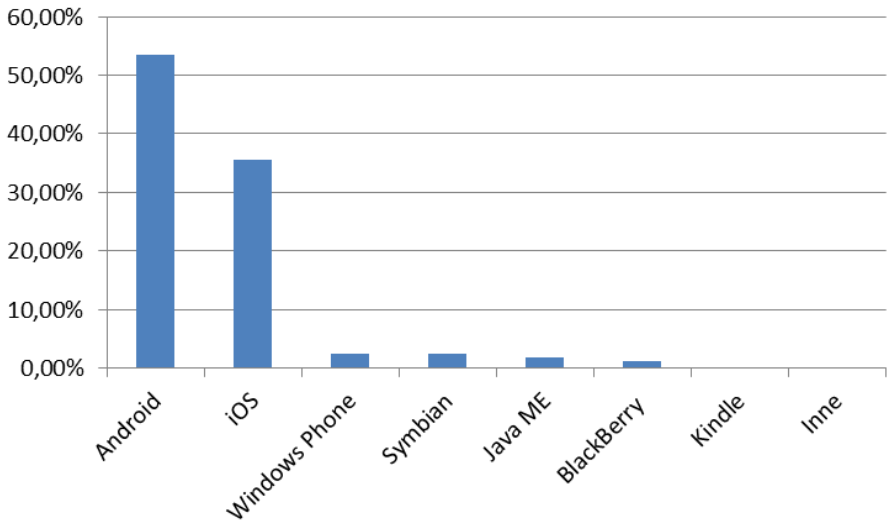
2.1 Początki systemu

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.

Android nazywa kolejne wersje swoich systemów zgodnie z nazwami słodyczy. Dwa premierowe wydania nie miały kryptonimów. Dopiero wersja 1.5, która została wydana 30 kwietnia 2009 roku, otrzymała nazwę *Cupcake*. Najnowsza wersja 6.0 posiada nazwę *Marshmallow*, a w marcu 2016 ukazało się wydanie *beta* systemu Android "N". Oficjalny kryptonim nie jest jeszcze znany.

2.2 Udział w rynku

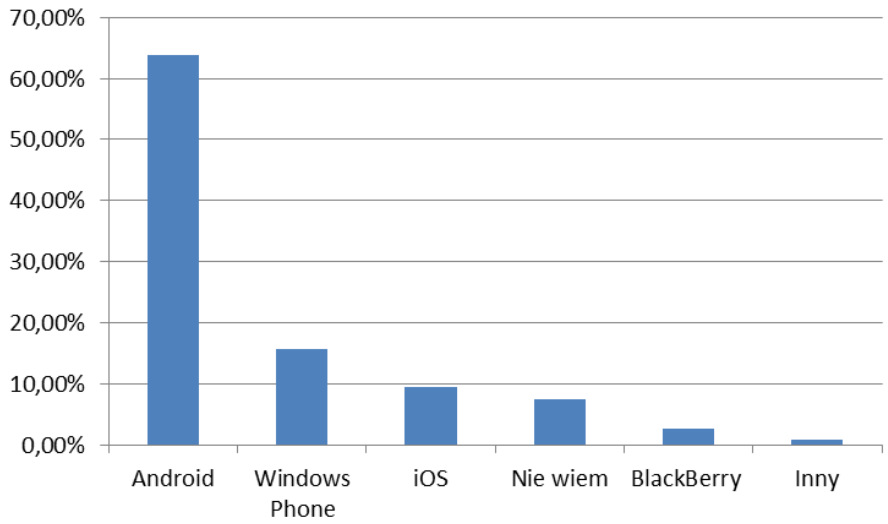
Android, jako system operacyjny przeznaczony dla urządzeń mobilnych, według [3] we wrześniu 2015 roku miał największy udział w rynku urządzeń na świecie, a jego wartość przekroczyła 53% (rysunek 2.1).



Rysunek 2.1: Android – udział w rynku urządzeń mobilnych na świecie[3].

Drugie miejsce na świecie według tego samego portalu zajmuje system iOS (38%), a trzecie - Windows z niespełna 3% udziału.

W Polsce proporcje są nieco inne. Prowadzi również Android, ale według [5] jego udział na rynku urządzeń mobilnych wynosi aż 63.8%. Na drugim miejscu plasuje się system Windows (15.7%), a trzeci iOS zajmuje około 10% rynku urządzeń mobilnych (rysunek 2.2).

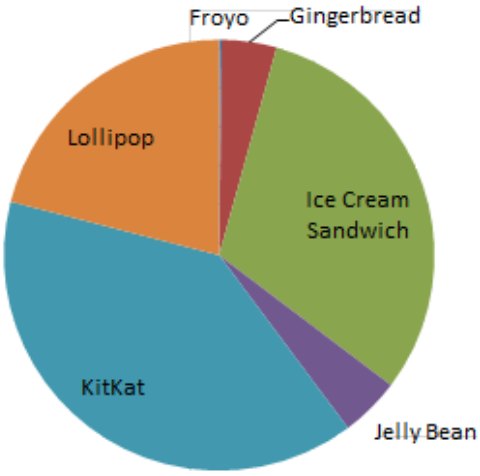


Rysunek 2.2: Android – udział w rynku urządzeń mobilnych w Polsce [5].

2.3 Rozwój systemu

Niezwykle cenną z punktu widzenia programistów 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 przedstawia rysunek 2.3.

Wersja	Kryptonim	API	Dystrybucja
2.2	Froyo	8	0,20%
2.3.3-2.3.7	Gingerbread	10	4,10%
4.0.3-4.0.4		15	3,70%
4.1.x	Ice Cream Sandwich	16	12,10%
4.2.x		17	15,20%
4.3	Jelly Bean	18	4,50%
4.4	KitKat	19	39,20%
5.0	Lollipop	21	15,90%
5.1		22	5,10%



Rysunek 2.3: Statystyki dotyczące używania poszczególnych wersji systemu Android przez użytkowników [14].

Z urządzeń, na których instalowany jest Android, większość stanowią 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 *head unitach* swoich samochodów, 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, 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 osób próbujących wykorzystać je do pisania własnych programów. Mimo wszystko, wielu programistów, nie pracując dla Google bezpośrednio, zaangażowanych jest w tworzenie kodu nowych wersji systemu Android. Nie jest to prostym zadaniem, bo z niezrozumiałych przyczyn firma utrzymuje większość informacji dotyczących rozwoju tej platformy w tajemnicy.

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

2.4 Programowanie w systemie w Android

Jako system operacyjny dostępny na licencji *Open Source*, Android zrzesza również ogromną społeczność programistów piszących aplikacje poszerzające funkcjonalność urządzeń. Najpopularniejszymi językami programowania, które wykorzystuje się do pisania aplikacji na Androida, są Java i C++ ze środowiskiem *Android NDK (Native Development Kit)*.

Programując aplikacje dla Androida w języku Java, programiści wykorzystują Android SDK (Software Development Kit). Ten zestaw narzędzi dla programistów składa się z dwóch części: SDK Tools – wymaganej do pisania aplikacji niezależnie od wersji Androida, oraz Platform Tools – czyli narzędzi zmodyfikowanych pod kątem konkretnych wersji systemu. W skład środowiska programistycznego wchodzi dokumentacja, przykładowe programy, biblioteki, emulator oparty na QEMU[1] oraz debugger. SDK dostępne jest zarówno dla Windowsa, Linuksa jak i dla MacOSX.

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 [26].

Dużym ułatwieniem dla programistów Javy jest również Android Studio - zintegrowane środowisko programistyczne oparte na IntelliJ IDEA[2]. Jego główne zalety to zarządzanie wieloma projektami, możliwość konfiguracji budowy programu w kilku wariantach dla jednego projektu, możliwość automatycznego uzupełniania kodu oraz wykonywania testów jednostkowych i integracyjnych bezpośrednio z narzędzia.

Jako narzędzie służące do budowania projektów w Javie wykorzystywany jest *Gradle*. 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.

O ile język Java wydawać się może najrozsądniejszym wyborem, o tyle wielu programistów używa środowiska NDK. Jest to zestaw narzędzi, który pozwala realizować części aplikacji za pomocą kodu w językach programowania C i C++. Zazwyczaj środowisko to wykorzystuje się w celu pisania programów, które intensywnie wykorzystują procesor, takich jak silniki gier, przetwarzania sygnału czy symulacji fizyki. Jednak programiści muszą wziąć pod uwagę również wady takiego rozwiązania, które mogą nie do końca zbilansować korzyści. Natywny kod Androida na ogół nie powoduje zauważalnej poprawy wydajności [15]. Zwiększa za to znacznie złożoność aplikacji, który to problem i tak już jest dużym wyzwaniem dla programistów Java. Podsumowując, z NDK należy korzystać tylko wtedy, jeśli jest to

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

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

niezbędne dla wytwarzanej aplikacji. Zanim programista zdecyduje się na to rozwiązanie, najpierw powinien sprawdzić, czy środowisko programistyczne Androida nie zapewnia już funkcjonalności, jakiej potrzebuje.

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 w języku Kotlin. Ten statycznie typowany język programowania działający na maszynie wirtualnej Javy jest zaprojektowany z myślą o pełnej interoperacyjności z Javą i polega na jej bibliotekach [26].

Rozdział 3

Testowalność oprogramowania

3.1 Pojęcie testowalności i pielęgnowalności oprogramowania

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?

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.

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

[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

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 Rodzaje testów

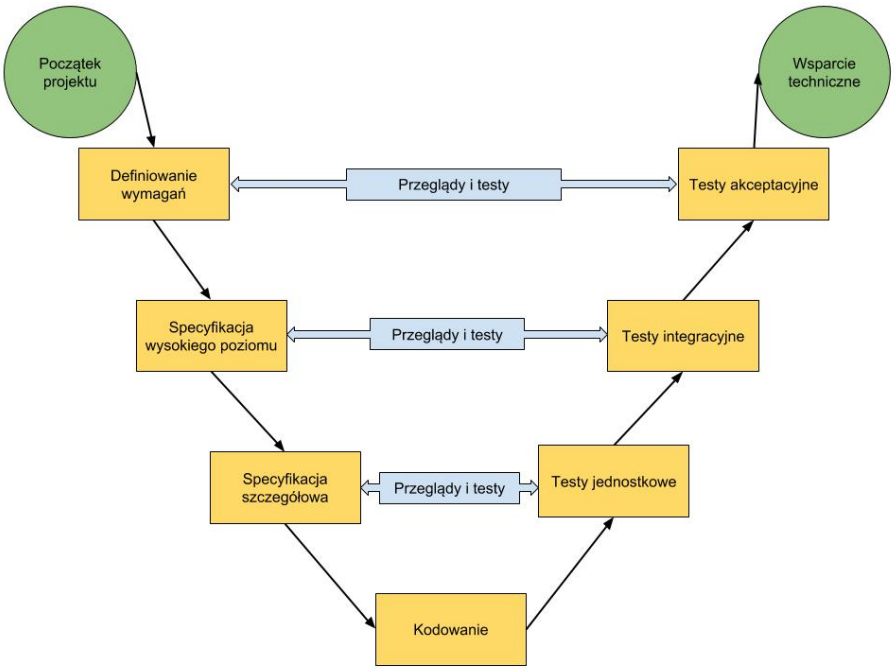
Proces testowania może zostać wprowadzony na każdym etapie tworzenia aplikacji, w zależności od wybranej strategii testu. Jednakże, jak już zostało wspomniane w podrozdziale 3.2, najlepiej jest rozpoczynać testowanie tak szybko jak to jest możliwe, w możliwie najwcześniejszym stadium projektu. Nawet 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 programowym, powinien zostać poddany testowaniu. Pozwala to już na początku uniknąć błędów mogących wpływać na rozwój projektu przez długi czas.

Rozpoznaje się kilka poziomów testowania, w zależności od modelu zarządzania projektem. Rozważając model V przedstawiony na schemacie 3.1, wyróżnić można cztery główne z punktu widzenia testowalności obszary:

- testy jednostkowe;
- testy integracyjne;
- testy systemowe;
- testy akceptacyjne.

3.3.1 Testy jednostkowe (modułowe)

Według [10], testy modułowe polegają na wyszukiwaniu błędów i weryfikacji funkcjonalności oprogramowania (np. modułów, klas), które można testować oddzielnie. Testowanie może być wykonywane w izolacji od reszty systemu, w zależności od kontekstu cyklu rozwoju



Rysunek 3.1: Model V - najpopularniejszy model zarządzania projektem informatycznym (*Vmodel*) [25].

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 3.6.1). 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.

3.3.2 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 [10].

3.3.3 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 użytkowników urządzeniach [10].

3.3.4 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.

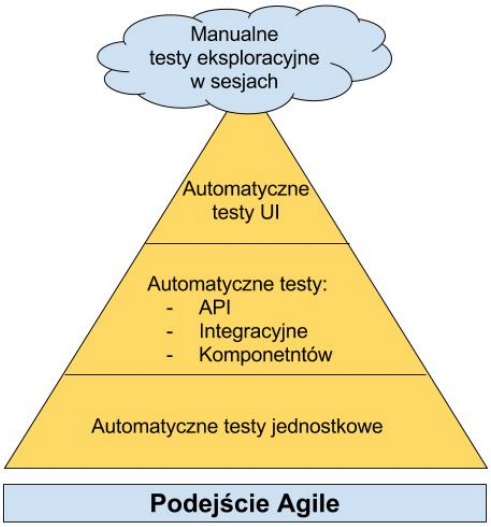
Odpowiedzialność za testy akceptacyjne, w przeciwieństwie do testów opisanych w poprzednich paragrafach, leży po stronie klientów. Zwykle testy akceptacyjne dzieli się na dwa etapy: testy *alfa* oraz *beta*. Testy *alfa* przeprowadzane są przez przyszłych użytkowników w siedzibie producenta, natomiast *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 [10].

3.4 Idealna i odwrócona piramida testowania

Bazując na [23] 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 oprogramowania. Idealna piramida testowania, spopularyzowana przez Mike'a Cohna [4] [8], przedstawiona została na rysunku 3.2.

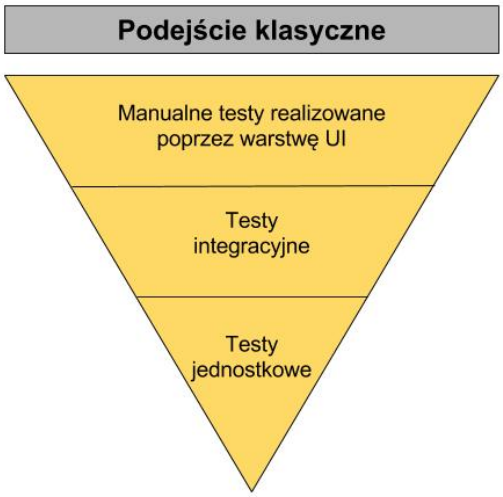
[4]Mike Cohn jest jednym twórców metodologii tworzenia oprogramowania Scrum. Jest jednym z założycieli Scrum Alliance oraz właścicielem Mountain Goat Software, firmy, która oferuje szkolenia na Scrum i technik Agile [26].



Rysunek 3.2: Idealna piramida testowania według Mike'a Cohna [8].

Wynika z tego, 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 przedstawia się tak, jak na rysunku 3.3.



Rysunek 3.3: Podejście klasyczne do testów, czyli odwrócona piramida testowania (*Ice Cream AntiPatern*) [24].

Na rysunku 3.3 widać, że z powodu zbyt dużego *couplingu* unit testy zastąpione zostają testami integracyjnymi, a najwięcej przypadków testowych wykonywanych jest na interfejsie użytkownika (w czym znacznie pomagają frameworki testowe pozwalające na zautomatyzowanie pewnych czynności, nagranie makr według specyfikacji lub *„user stories”*) oraz testy manualne.

Autor nie zaprzecza, że tym sposobem nie da się dobrze przetestować aplikacji, szczególnie jeżeli zastosujemy taktyki testowania opisane w podrozdziale 3.6. Jak powszechnie wiadomo, testowanie gruntowne nie jest możliwe, jakiejkolwiek metody by nie użyto. 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

3.5 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 do najważniejszych z nich można zaliczyć:

- Niedostateczną znajomość języka programowania wśród programistów;

  Jeżeli programista 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.

- Słabą znajomość narzędzi testowych;

  Jeżeli programista nie doskonali się w temacie projektowania testów, nie poznaje narzędzi testowych, które mogą spowodować, że testowanie stanie się łatwe, szybko zniechęci się już przy pierwszej próbie napisania trudniejszego testu.

- Niską jakość kodu źródłowego;

  Kiepsko zaprojektowany kod i nieoptymalnie 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 3.3.

- 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 terminu oddania swojej części programu na "ulepszanie" kodu, co skutkuje brakiem czasu na testy jednostkowe.

- Przekonanie programistów o własnej nieomylności.

  Człowiek jest omylny i popełnia błędy, do czego nawiązano już w rozdziale 3.2. Niektórzy programiści nie chcą jednak przyjąć tego do wiadomości i upierają się przy stwierdzeniach, że testy do ich kawałka kodu źródłowego nie są potrzebne.

3.6 Zwinne podejście do testowania

3.6.1 Wytwarzanie sterowane testowaniem

Technika wytwarzania sterowanego zarządzaniem (*Test Driven Development*) zyskuje coraz więcej popularności. Związane to jest pośrednio z nowymi modelami zarządzania projektami informatycznymi opartymi na metodykach zwinnych, w tym Agile[5]. W założeniu podejście takie pozwala:

- wyszukać więcej błędów na wcześniejszym etapie procesu programowania;
- świadomie zaimplementować zmiany wtedy, kiedy są potrzebne;
- wykonać testy regresywne bazując na wcześniej napisanych procedurach;
- przedłużyć żywotność kodu.

Jeżeli istnieją testy jednostkowe i pokrywają znaczącą część kodu źródłowego (70 - 80%), 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 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 jak zmiana serwera baz danych, zmiana modelu, czy interfejsu użytkownika. 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.

[5]Agile software development – grupa metodyk wytwarzania oprogramowania opartego na programowaniu iteracyjno-przyrostowym, powstałe jako alternatywa do tradycyjnych metod typu *waterfall*[6]. 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 [26].

3.6.2 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 3.6.3.

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*[7] (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 [2].

3.6.3 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

[7]GWT - forma tworzenia specyfikacji na podstawie przykładów

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

Rozdział 4

Testowalność aplikacji Android

4.1 Pojęcie architektury w kontekście systemów komputerowych

4.1.1 Architektura systemowa

Systemy komputerowe są kombinacjami sprzętu komputerowego i oprogramowania działającymi coraz częściej również w ramach sieci komputerowej. Termin *"architektura systemu"* opisuje strukturę oraz interakcję 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" [16]:

> "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*[1], architektura oprogramowania tworzona jest przed rozpoczęciem pisania kodu źródłowego i jest to bardzo poważny proces, który mógł wpłynąć na całość projektu. W modelach projektowych opartych na metodykach zwinnych,

[1]Iteracyjny model kaskadowy (ang. waterfall model) – jeden z kilku rodzajów procesów tworzenia oprogramowania zdefiniowany w inżynierii oprogramowania [26].

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*.

Analizując przytoczoną powyżej definicję architektury, można wyciągnąć następujące wnioski:

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[2]) 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, a moduł wspomagania księgowości - trzecim. 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 programowych 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

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

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 komputerowy posiada architekturę oprogramowania

Każdy system można pokazać w sposób opisany w poprzednich paragrafach, czyli jako zestaw elementów i 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 w danym przedsiębiorstwie, 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

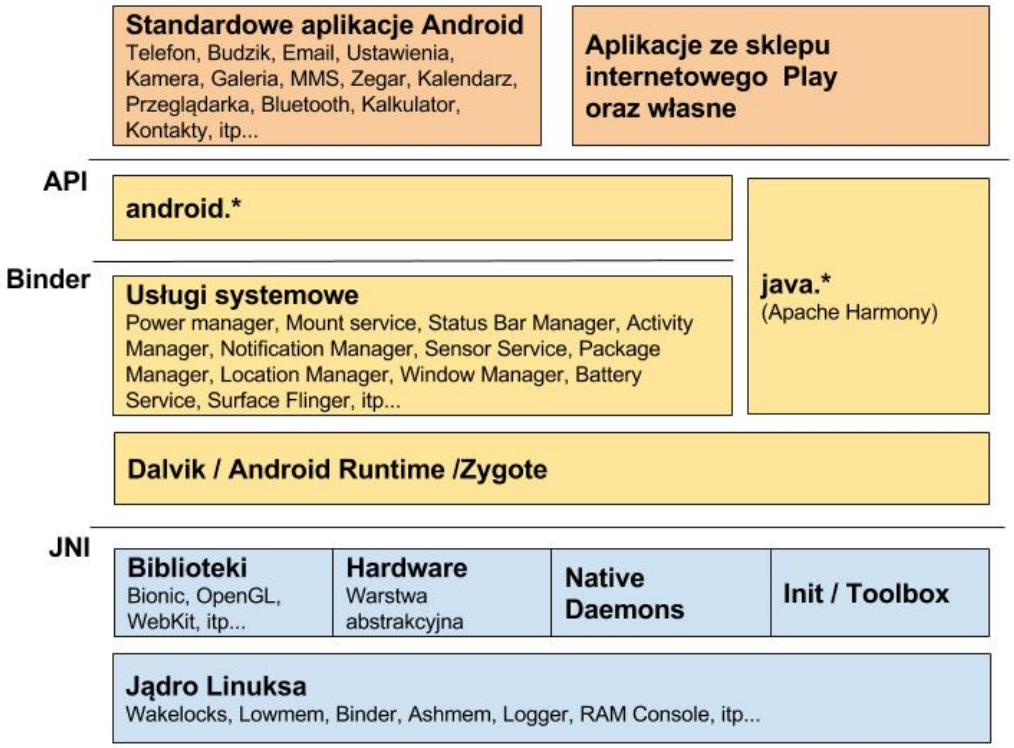
Zachowanie się każdego elementu architektury może być również jej częścią, o ile czynność ta wnosi jakieś nowe informacje i wpływa pozytywnie na zrozumienie systemu. Reprezentowanie, jak elementy współdziałają ze sobą, jest częścią definicji architektury, opisanej w poprzednich paragrafach. Schematy blokowe, 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* [12] (rysunek 4.1).



Rysunek 4.1: Przegląd architektury Android [27].

Aplikacje Android pozwalają programistom 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 na konfigurowanie interfejsu użytkownika (UI[3]), dostęp do baz danych oraz przekazywanie informacji pomiędzy poszczególnymi komponentami aplikacji.

Zarówno aplikacje jak i opisywany *framework* napisane są w języku Java 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.

[3]User Interface

Natywne elementy kodu Androida zawierają usługi systemowe, usługi sieciowe 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. 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 [12] (IPC[4]).

4.3 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*[5], czy funkcji należących do standardowych bibliotek, gdyż zapewne zostały przetestowane już podczas procesu ich tworzenia.

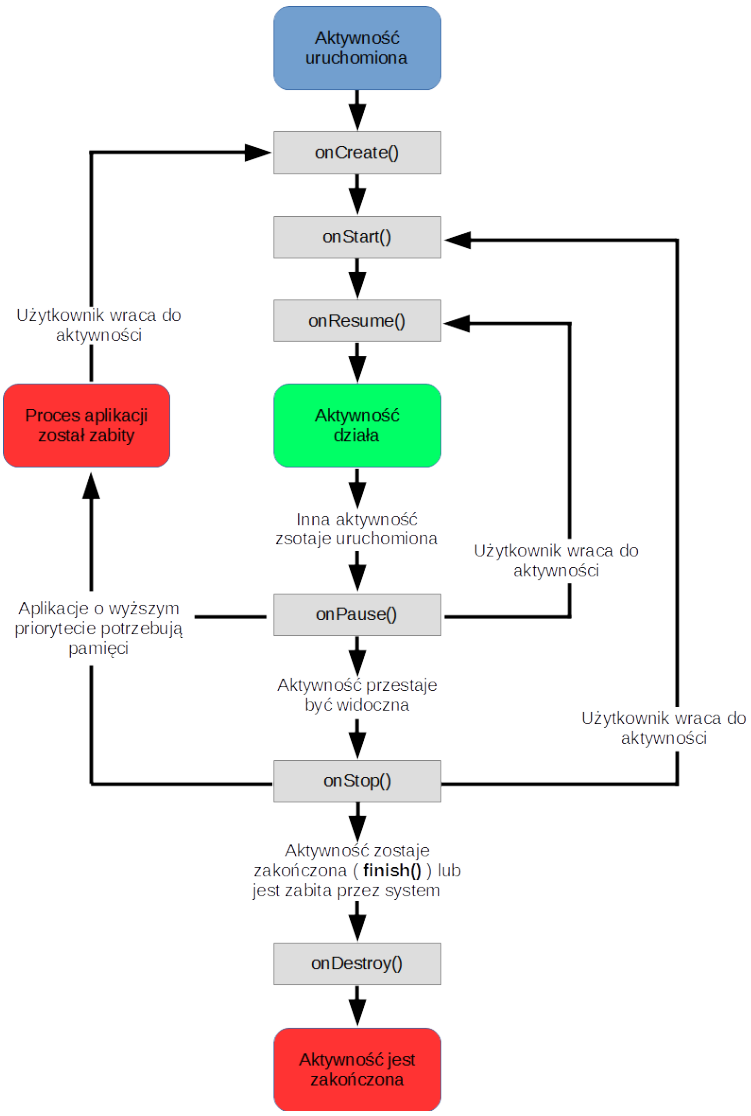
Według [23], można wyróżnić trzy główne obszary testowania:

- Cykle życia aktywności;

  Należy przetestować, czy aktywność przechodzi prawidłowo przez swoje cykle życia. Trzy główne pętle do przetestowania znajdują się na rysunku 4.2:

[4]Inter-Process Communication

[5]*Settery* i *gettery* - w programowaniu metody służące do kontrolowania zmian wartości zmiennych



Rysunek 4.2: Cykl życia *Activity* [4]

- 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: 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*[6]

  Według dokumentacji Android [9] możliwe są następujące opcje przechowywania danych:

[6]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]

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

- pamięć wewnętrzna urządzenia - do zachowywania danych niepublicznych;

- zewnętrzna karta pamięci - do zachowywania danych publicznych;

- 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[7], 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. Programiści 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 innego producenta. 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;

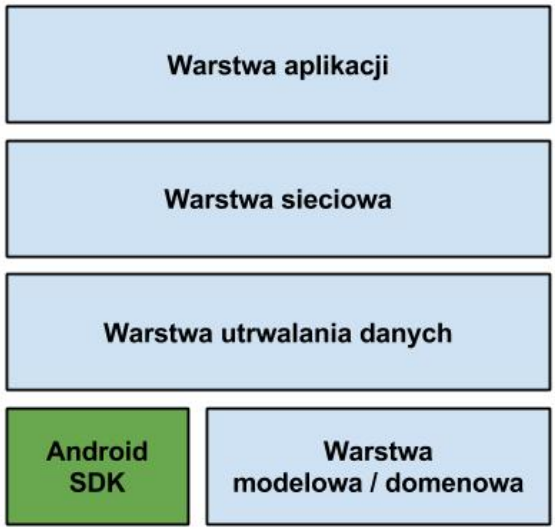
  - zewnętrzne karty pamięci.

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

4.4 Standardowe podejście przy tworzeniu aplikacji

W większości przypadków aplikacje z przeznaczeniem dla systemu Android pisane są według schematu widocznego na rysunku 4.3:

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



Rysunek 4.3: Podejście „standardowe” przy tworzeniu aplikacji dla systemu Android.

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:

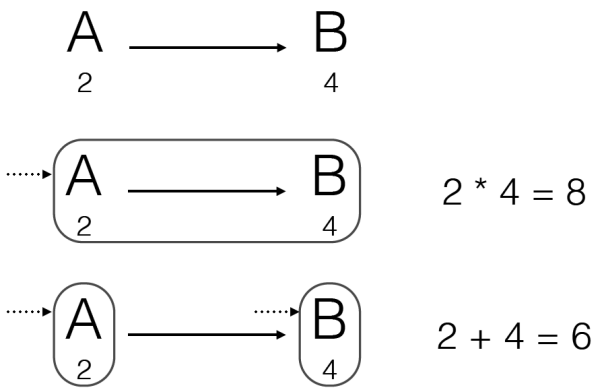
- nie jest zachowana zasada pojedynczej odpowiedzialności;
- warstwa odpowiedzialna za logikę domenową jest pomieszana z warstwą UI (User Interface);
- logika UI jest pomieszana z asynchronicznym pobieraniem danych;
- funkcje *callback*[8] można znaleźć w całym kodzie;
- elementy warstwy UI: Activity i Fragmenty potrafią mieć tysiące linii kodu;
- w większości plików aplikacji na każdej warstwie istnieją odwołania do środowiska Android;
- i ostatnie, ale najważniejsze z punktu widzenia tej pracy: problemy z wprowadzeniem testów jednostkowych.

Spowodowane jest to dwoma czynnikami: pierwszy to trudność w wyodrębnieniu obszarów testowych wymienionych w części 4.3 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.

[8]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 [26].

4.5 Trudności w testowaniu aktualnej struktury aplikacji

Weźmy dwie funkcjonalności przedstawione na rysunku 4.4: 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.



Rysunek 4.4: Różnice w testowaniu klas osobno i razem. *"Integrated Tests are a Scam"* - schemat autorstwa J.B. Rainsbergera [11].

Testując te funkcjonalności razem (z powodu sprzężenia nie ma innego wyjścia) należy wykonać łącznie 8 testów, co zostało przedstawione na równaniu 4.1:

$$2 * 4 = 8 \tag{4.1}$$

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 łącznie 6 testów jednostkowych (równanie 4.2):

$$2 + 4 = 6 \tag{4.2}$$

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

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.

4.6 Pielęgnowalność aplikacji Android

W realnym świecie niestety bardzo rzadko jest 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. Przeanalizujmy jako przykład aplikację *Gmail*, największy program pocztowy wydawany przez firmę

Google[9]. 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*[10] 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.

[9]Gmail – bezpłatny serwis webmail posługujący się technologią AJAX, stworzony i rozwijany przez przedsiębiorstwo Google. W lutym 2016 roku liczba jego użytkowników przekroczyła miliard [26].

[10]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 [26].

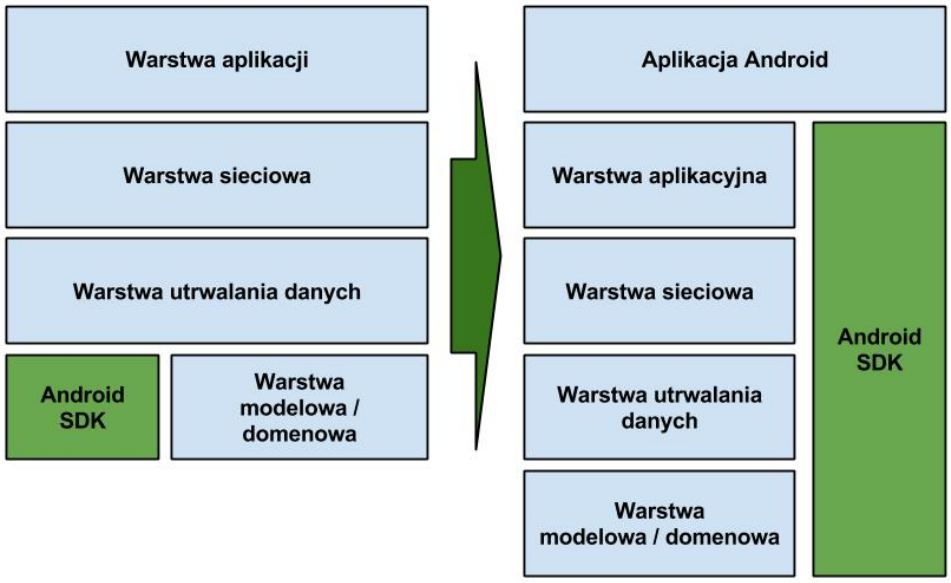
Rozdział 5

Alternatywne podejście do pisania aplikacji na platformę Adroid

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 w rozdziale 4.4 tak jak to widać na rysunku 5.1.



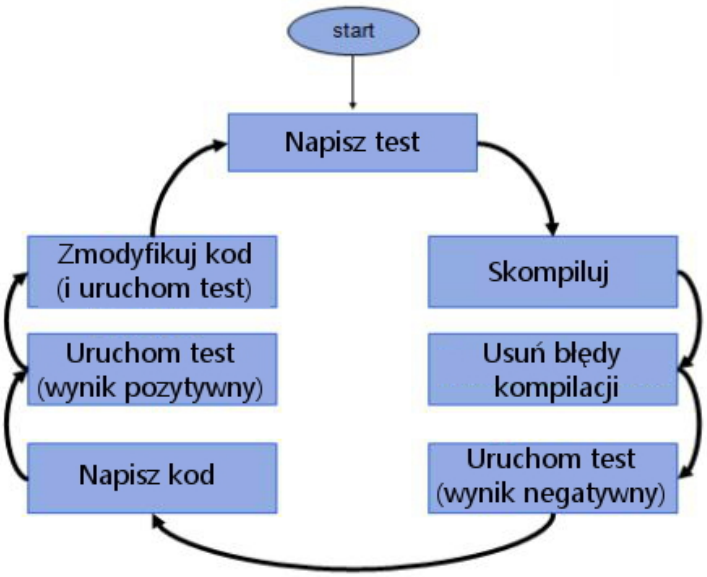
Rysunek 5.1: Propozycja modyfikacji struktury aplikacji Android.

Nowy model aplikacji oddziela warstwy *Application*, *Networking*, *Persistance* oraz *Model/Domain* od środowiska Android SDK, a co za tym idzie nie ma potrzeby umieszczać odwołania do tego środowiska praktycznie w każdym pliku. Dopiero najwyższą warstwą

w hierarchii jest warstwa *Aplikacja Android*. W takim podejściu należy użyć Androida jako pewnego rodzaju *pluginu* do tworzonej aplikacji i uniezależnić warstwę logiczną od reszty warstw. Najpierw napisać aplikację, która będzie oddzielona od Android SDK (na przykład w czystej Javie), następnie dodać zależności do Android SDK i skompilować całość w aplikację androidową. w ten sposób można być przekonanym, że kod, który stanowi podstawę naszej aplikacji, czyli jej logikę działania, zostanie przetestowany w oddzieleniu od reszty warstw.

5.1.2 Zastosowanie techniki *Test Driven Development* przy tworzeniu oprogramowania

Jak już zauważono w rozdziale 3.6.1 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 pisania programu w technice *Test Driven Development*.

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 refaktoring[1] kodu;

[1]Refaktoryzacja (czasem też refaktoring, ang. refactoring) – proces wprowadzania zmian w projekcie/programie, w wyniku których zasadniczo nie zmienia się funkcjonalność. Celem refaktoryzacji jest więc nie wytwarzanie nowej funkcjonalności, ale utrzymywanie odpowiedniej, wysokiej jakości organizacji systemu [26].

- 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 - programista 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 3.2 z rozdziału 3.4 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. Testy jednostkowe są solidną podstawą automatycznych testów regresyjnych.

5.1.4 Kod jako dokumentacja programu

Martin Robert Cecil[2] - popularny w środowisku programistycznym *Uncle Bob* - w [18], 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;

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

- 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;
- 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 Praca z kodem zastanym

W rozdziale 4.6 autor nawiązał do problemu pracy z kodem zastanym, tak zwanym *Legacy Code*. Co więc zrobić, aby zaimplementować testy do już napisanego kodu? Godfrey Nolan[3] w swojej książce "Agile Android" [21] proponuje następujące rozwiązanie:

- 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*[4]), który będzie wykonywał testy jednostkowe, do których istnieje przekonanie, że ich implementacja nie powinna się zmieniać. Należy pamiętać, że testy automatyczne muszą również podlegać przeglądom;
- Uświadomić zespół programistów na czym polegają testy jednostkowe i przekonać ich do stosowania TDD;

[3]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"

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

- Dodać metryki mierzenia kodu do CI. Ustawić poziom minimalny na 10-15%;
- Użyć frameworka do podstawowych testów GUI już istniejącej aplikacji. 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[5].

5.3 Różne podejścia do uporządkowania architektury

5.3.1 Architektura cebulowa - *The Onion Architecture*

Jeffrey Pallermo w połowie 2008 roku opublikował na swoim blogu serię artykułów [22], 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.

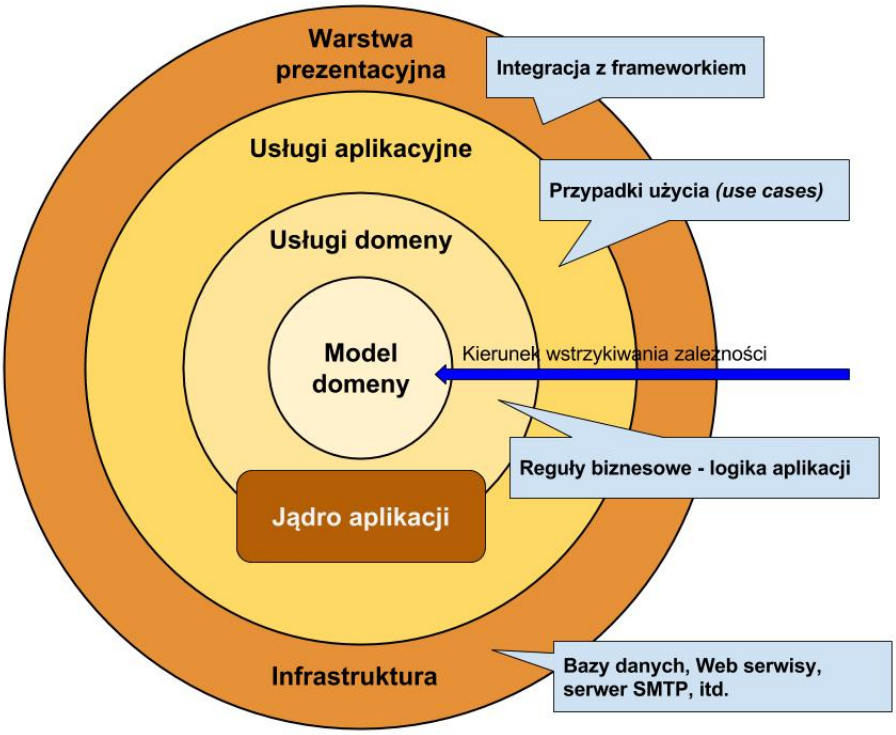
W zaproponowanym przez Palermo modelu, im warstwy położone są w architekturze głębiej, tym mniej podatne są na zmiany. Zależności mogą być zwrócone tylko w jedną stronę - do wnętrza "cebuli". w ten sposób w centrum znajdzie się model domeny, a na zewnątrz - warstwa bazodanowa. Jest to podejście nazywane *odwróceniem zależności*[6] i wymaga od programistów zastosowania odpowiednich technik projektowych, między innymi *wstrzykiwania zależności*[7].

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

[6]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.

[7]Dependency Injection (DI) – wzorzec projektowy i wzorzec architektury oprogramowania polegający na usuwaniu bezpośrednich zależności pomiędzy komponentami na rzecz architektury typu plug-in [26].

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.



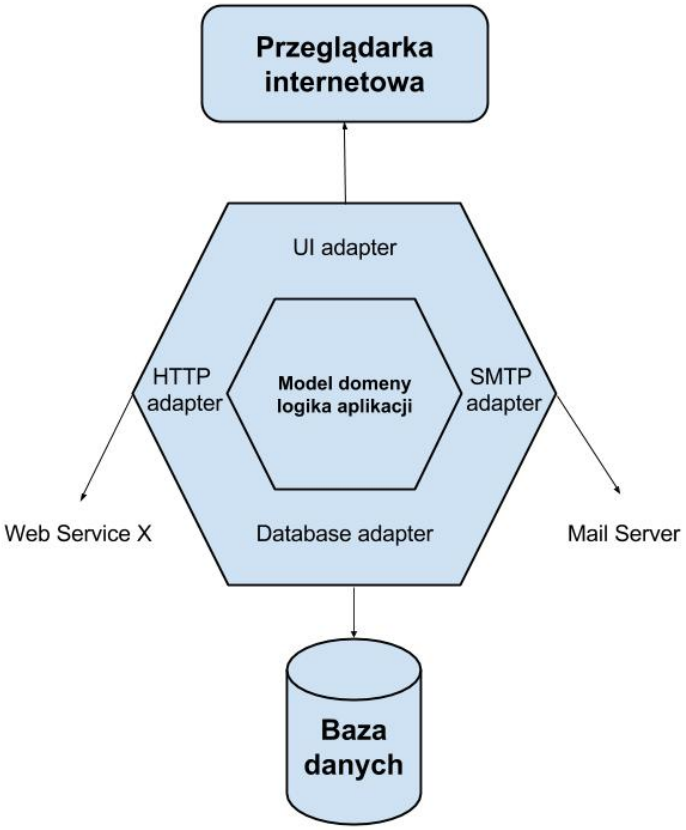
Rysunek 5.3: *Onion architecture* według Jeffrey'a Palermo [22].

Kluczowe założenia architektury cebulowej (rysunek 5.3):

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

5.3.2 Architektura portów i adapterów - *Ports and Adapters Architecture*

Opracowana w 2005 roku przez Alistaira Cockburna[8] architektura niegdyś znana była pod nazwą architektury heksagonalnej [7]. 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.



Rysunek 5.4: Architektura heksagonalna według Alistaira Cockburna.

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.

Jądro - *Kernel* Jądro oferuje model domeny i logikę biznesową: ogólne możliwości aplikacji, operacje, logikę operacji oraz mechanizmy wspierania decyzji, co już w wymienionej

[8] 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 [26].

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[9], 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[10], jakim jest na przykład 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.

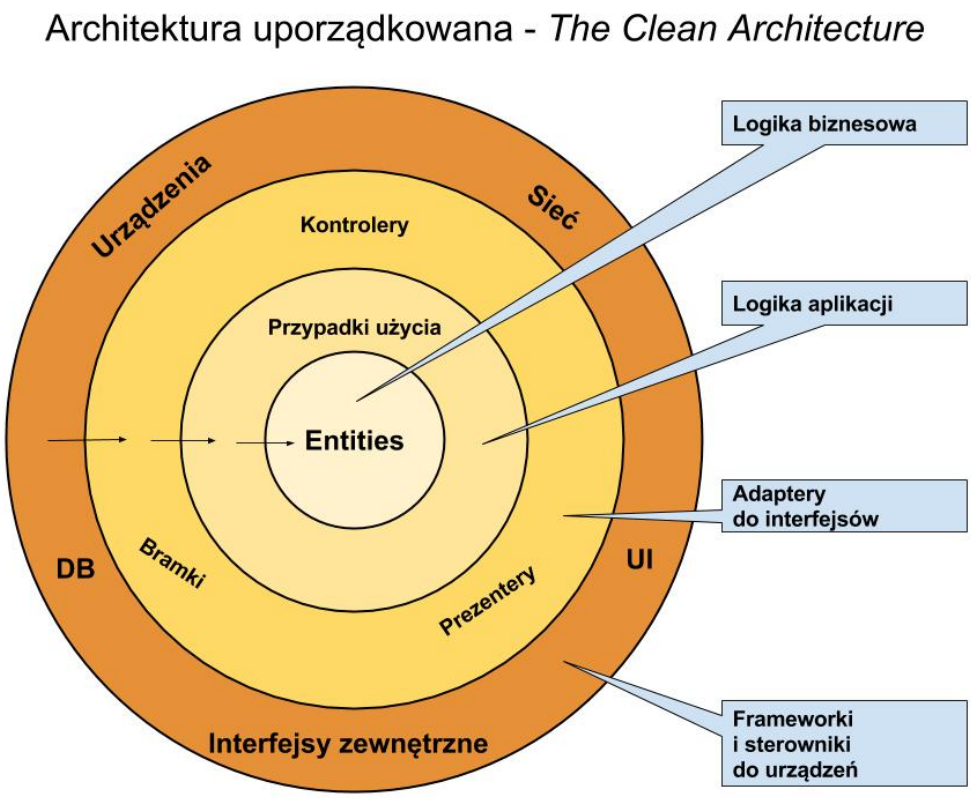
Z punktu widzenia systemu napisanego w architekturze cebulowej, testowalność oprogramowania wygląda następująco: testy jednostkowe eksplorują warstwę *Kernel* zawierającą model domeny i logikę biznesową, testy integracyjne sprawdzają porty, a testy systemowe badają kilka warstw *Kernel* na raz.

5.3.3 Architektura uporządkowana - *The Clean Architecture*

Cytowany już w rozdziale 5.1.4 Uncle Bob opublikował w 2012 roku na swoim blogu propozycję usystematyzowania architektury systemowej [20], która zainteresowała również autora tej pracy. Schemat architektury uporządkowanej przedstawiony jest na rysunku 5.5.

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

[10]Cienki klient (ang. thin client) – komputer bądź specjalizowane urządzenie (terminal komputerowy) wraz z odpowiednim oprogramowaniem typu klient, umożliwiające obsługę aplikacji stworzonej w architekturze klient-serwer. Cechą szczególną cienkiego klienta jest niezależność od obsługiwanej aplikacji serwerowej (jej zmiana nie pociąga za sobą konieczności wymiany oprogramowania klienta). Dodatkowym atutem jest niewielkie zapotrzebowanie na moc przetwarzania [26].



Rysunek 5.5: Clean architecture of Android według Uncle Bob [20].

Idea tego rozwiązania jest następująca:

- Najważniejszym elementem jest środek, czyli warstwa *Entities*. w warstwie tej znajdują się kluczowe dla tworzonej aplikacji reguły biznesowe gwarantujące poprawność działania programu - logika działania aplikacji;
- Drugą warstwą jest *Use Cases* - *przypadki użycia*, ale bardziej adekwatną jest nazwa *intencje biznesowe*. Jest to zbiór wszystkich zachowań, jakich oczekuje się od tworzonego systemu. W przypadku aplikacji bankowej, może być to na przykład funkcja wykonywania przelewu, jako jeden przypadek użycia, a innym takim przypadkiem byłoby sprawdzenie stanu konta;
- Wyższa, otaczająca warstwa dotyczy wszystkich kontrolerów i prezenterów;
- Dopiero na ostatniej, najbardziej zewnętrznej warstwie pojawia się powiązanie z Android SDK. Wykorzystywane jest, 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 przpływu 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.

Podsumowując, jeżeli programista używa powiązania z Android SDK, to tylko z poziomu najwyższej warstwy. Nie należy 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 *UI*, bazy danych lub innych urządzeń zewnętrznych;
- jest testowalna, w oderwaniu od warstwy zawierającej interfejsy wymienione powyżej.

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++.

Rozdział 6

Porównanie testowalności na przykładzie wybranej aplikacji

6.1 Wybór rozwiązania

O ile wybór *Test Driven Development* jako techniki wspomagającej programowanie aplikacji okazał się dla autora naturalnym z punktu widzenia metodologii zwinnych (patrz sekcja 5.1.2), o tyle wybór architektury systemowej sprawił więcej kłopotu.

W poprzednim rozdziale przeanalizowano trzy próby usystematyzowania architektury systemu: architekturę cebulową *(The Onion Architecture)*, architekturę portów i adapterów *(Ports and Adapters Architecture)* oraz architekturę uporządkowaną *(The Clean Architecture)*. Wszystkie z nich proponują uporządkowanie systemu w podobny sposób, różniąc się jedynie szczegółami. Autor zdecydował się na wybór ostatniej z wymienionych - *The Clean Architecture* - ze względu na większą ilość dostępnych materiałów i dokumentacji na jej temat. Robert Cecil Martin, autor tej propozycji, oprócz artykułu umieszczonego na swoim blogu nawiązał do tego tematu również w swoich książkach [18], [19, **?**] [17], wyjaśniając kluczowe pojęcia w przyjaznym dla programistów języku.

Drugim kryterium wyboru była data publikacji propozycji - w porównaniu do pozostałych dwóch, *The Clean Architecture* okazało się podejściem najświeższym.

6.2 Opis doświadczenia

Doświadczenie polegało 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 [1], a której źródła udostępnione są w serwisie GitHub na licencji *Open Source*.

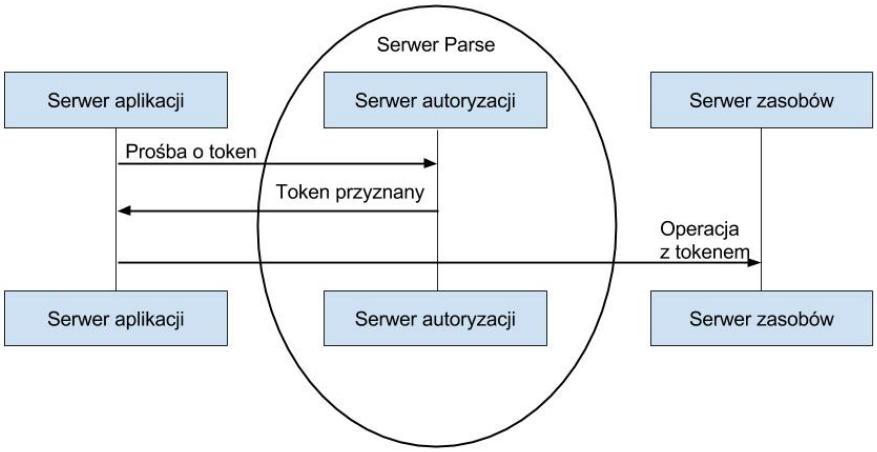
W części doświadczalnej badania ograniczono do analizy testowalności w zakresie testów jednostkowych i wczesnych testów integracyjnych. Pominięto przegląd na etapie testów

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 *Test Driven Development* i podejścia *The Clean Architecture* poprawi testowalność programu i spowoduje ułatwienie dalszego procesu testowego.

6.3 Opis aplikacji

JSON Web Token Authentication for Android poświadcza prawdziwość użytkowników Androida i iOS korzystając z *REST API* serwera *Parse* oraz *JSON*[1] *Web Tokens (JWT)*. JWT to otwarta, według standardu przemysłowego RFC 7519 [13], 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.

Serwer *Parse* to wspólna platforma do przechowywania danych i interakcji z usługami internetowymi dla aplikacji mobilnych. Wielu programistów decyduje się na wykorzystanie tej sprawdzonej platformy, zamiast wymyślać własne rozwiązania w tym zakresie. Zasadę działania autentykacji przedstawia rysunek 6.1.



Rysunek 6.1: Schemat działania JWT.

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*.

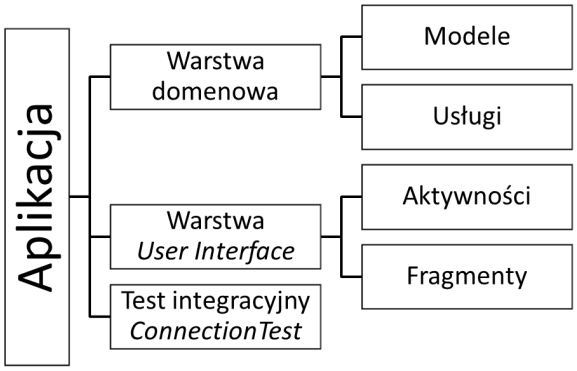
[1] JSON, JavaScript Object Notation – lekki format wymiany danych komputerowych. JSON jest formatem tekstowym, bazującym na podzbiorze języka JavaScript [26].

6.4 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.

6.4.1 Budowa analizowanej aplikacji w wersji pierwotnej

Schemat budowy aplikacji *JSON Web Token Authentication for Android* w wersji pierwotnej przedstawia rysunek 6.2.



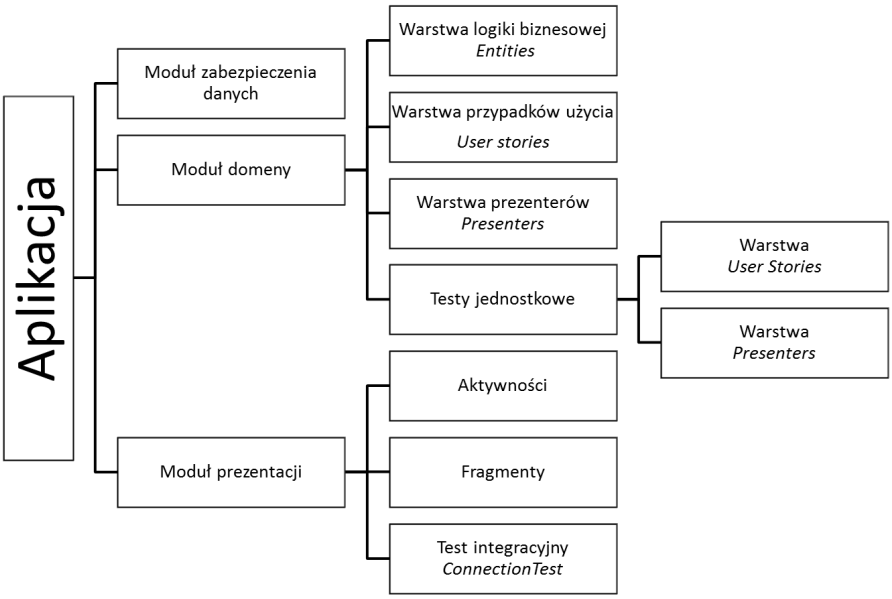
Rysunek 6.2: Schemat budowy badanej aplikacji wykorzystującej architekturę standardową

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 związane 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 szczegółowo opisywane rozwiązania można dojść do wniosku, że ich idea jest taka sama, a różnią się jedynie szczegółami. W tym przypadku autor *JSON Web Token Authentication for Android* zdecydował się na *The Clean Architecture*, zaprezentowaną w tej pracy w rozdziale 5.3.3.

Aplikacja w wersji poprawionej nadal ma budowę modułową - tym razem moduły są trzy: moduł domeny, moduł danych i moduł prezentacyjny. Zgodnie z nowym podejściem przedstawionym na rysunku 5.5, można rozpoznać poszczególne warstwy uporządkowanej architektury (rysunek 6.3):



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

- 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;
- Warstwę *Use Cases*, należącą do modułu domeny, do której na podstawie przypadków użycia zostały napisane testy jednostkowe;
- Warstwę *Presenters*, również należącą do modułu domenowego i odpowiadające jej testy jednostkowe;
- 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 komputerowy i ten sam emulator: Nexus_5_API_23.

6.6 Wyniki doświadczenia

6.6.1 Wyniki dla testów jednostkowych

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)

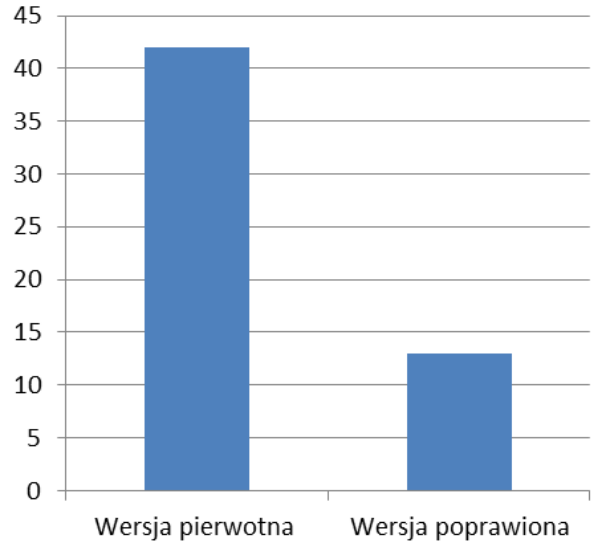
W przypadku aplikacji w wersji poprawionej, jak przedstawiają tabele 6.2 i 6.3, zaprojektowano 6 testów opartych na przypadkach użycia oraz 7 testów dla prezenterów, czyli łącznie 13 testów jednostkowych (równanie 6.1).

$$6 + 7 = 13 \tag{6.1}$$

W przypadku aplikacji w wersji pierwotnej - zgodnie z wyjaśnieniem, które autor umieścił w rozdziale 4.5 - aby pokryć ten sam obszar funkcjonalności należałoby zaprojektować zgodnie z równaniem 6.2 42 testy jednostkowe.

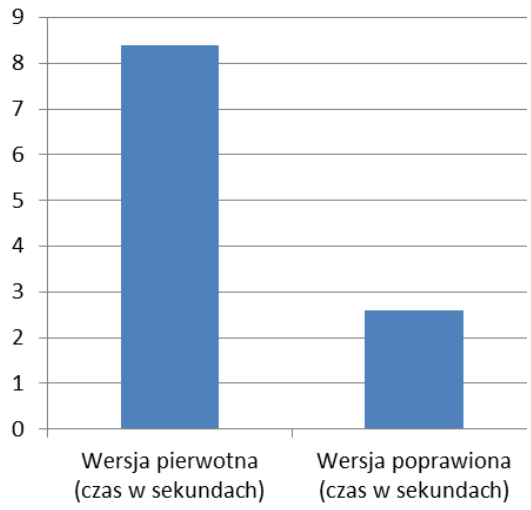
$$6 * 7 = 42 \tag{6.2}$$

Różnicę tę uwidoczniono na wykresie 6.4.



Rysunek 6.4: 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 wykres 6.5:



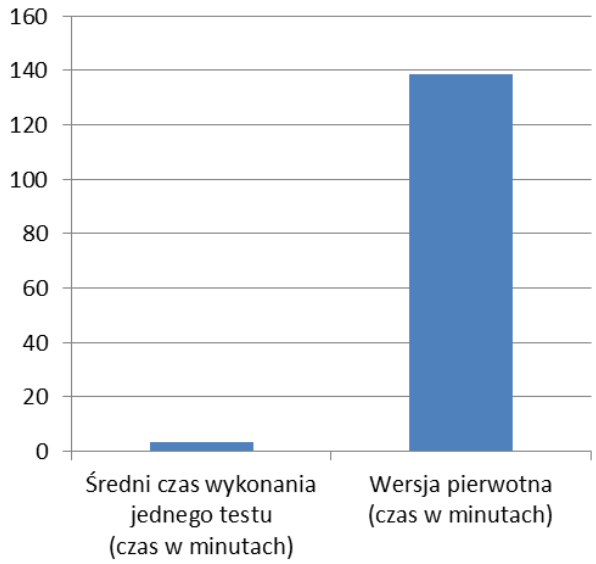
Rysunek 6.5: Porównanie czasu wykonania testów jednostkowych potrzebnych do pokrycia tej samej funkcjonalności w przypadku wersji pierwotnej i poprawionej.

6.6.2 Testy integracyjne

Dla obu opisywanych wersji zaprojektowano test integracyjny *ConnectionTest*, dla którego 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 (zostało to zaprezentowane na wykresie 6.6).



Rysunek 6.6: 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 6.4 i 6.5.

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 aplikacji w wersji pierwotnej nie było, wtedy potrzeba zwiększenia czasu przeznaczonego na testy integracyjne i wyższego szczebla jest jeszcze większa, co przedstawia wykres 6.6.

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.

Zakończenie

Poniższa praca zawiera analizę testowalności aplikacji na platformę Android. Praca została podzielona na dwie części. W pierwszej autor wprowadza i wyjaśnia podstawowe pojęcia związane z programowaniem na platformę Android oraz testowalnością oprogramowania. Okazuje się, że powszechnie stosowane podejście projektowe przy realizacji aplikacji mobilnych na platformę Android znacznie utrudnia wydajne pisanie testów jednostkowych, a cały ciężar testowania przeniesiony jest na testy integracyjne, systemowe, a nawet akceptacyjne. W części drugiej autor szczegółowo opisuje proces projektowy, który znacząco ułatwia utrzymanie i opracowywanie aplikacji Andorid [20]. Następnie porównuje wybrane rozwiązania architektoniczne w kontekście automatycznego testowania. Otrzymane wyniki są bardzo obiecujące i pozwalają na wyciągnięcie wielu ciekawych wniosków.

Testy automatyczne powinny przyczyniać się do poprawy jakości dostarczając szybkiej informacji zwrotnej na temat działania programu. Z przeprowadzonej analizy 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. Okazuje się, że zastosowanie architektury *The Clean Architecture* oraz techniki *Test Driven Development* w przypadku badanej aplikacji pozwala na napisanie 6 razy mniej testów jednostkowych niż w przypadku tej samej aplikacji napisanej w sposób standardowy, co zostało zaprezentowane na wykresie 6.4. Przekłada się to zarówno na czas wykonywania tych testów (wykres 6.5), co ma niebagatelne znaczenie przy automatyzacji zestawów testowych i wyborze zakresu dla testów regresywnych, jak i ich pielęgnowalność.

W przypadku testów integracyjnych analiza wykazała, że zysk może być jeszcze większy. W granicznym przypadku czas ich wykonania, jak pokazuje wykres 6.6, może zostać skrócony nawet czterdziestokrotnie.

Cel pracy został osiągnięty. Wykonane eksperymenty potwierdzają wszystkie postawione przez autora tezy. W przyszłości planuje się rozszerzenie badań na testy systemowe i akceptacyjne. Teraz nie było to możliwe, ponieważ w badanej aplikacji nie są one zdefiniowane, a autor nie miał dostępu zarówno do wymagań systemowych, jak i do wymogów na poziomie użytkownika.

Bibliografia

[1] Victor Albertos. Authentication for android and ios, July 2015.

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

[3] Android. Android.com.pl, October 2015.

[4] Android.com. Cykl życia aktywności, March 2016.

[5] Antyweb. Antyweb.pl, 2015.

[6] Michał Charmas. Przejrzysty i testowalny kod na androidzie? spróbujmy z clean architecture. *Programista*, 28(9):66–70, 2014.

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

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

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

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

[11] Rainsberger J.B. Integrated tests are a scam, 2014.

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

[13] JWT. Standard rfc 7519, May 2015.

[14] Kamil Komisarz. Android now, October 2015.

[15] Jakub Konieczny. Czy java faktycznie jest taka wolna?, September 2012.

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

[17] Robert Cecil Martin. *Agile software development: principles, patterns, and practices.* Pearson Education, New York, wydanie pierwsze, 2002.

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

[19] Robert Cecil Martin. *The clean coder: a code of conduct for professional programmers.* Prentice Hall, New York, wydanie pierwsze, 2011.

[20] Robert Cecil Martin. The clean architecture, August 2012.

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

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

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

[24] Scrumdo.pl. Piramida testów agile, August 2015.

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

[26] Wikipedia.org. Wikipedia - wolna encklopedia, 2016.

[27] Karim Yaghmour. Przegląd architektury android, 2012.

Spis rysunków

Spis tabel