PROJEKT INŻYNIERSKI

„System do obsługi sprzedaży i dostaw produktów i przetworów rolnych”

Szymon Czech
Nr albumu 297923

Kierunek: Informatyka
Specjalność: Grafika komputerowa i oprogramowanie

PROWADZĄCY PRACĘ
Dr inż. Tomasz Moroń
KATEDRA Cybernetyki, Nanotechnologii i przetwarzania danych
Wydział Automatyki, Elektroniki i Informatyki

GLIWICE 2024

Tytuł pracy:

System do obsługi sprzedaży i dostaw produktów i przetworów rolnych.

Streszczenie:

Celem pracy jest przedstawienie procesu projektowania i implementacji oprogramowania służącego do obsługi sprzedaży i dostaw produktów i przetworów rolnych które przeznaczone jest na urządzenia mobilne. Omawia ona narzędzia wykorzystane do opracowania aplikacji takie jak język programowania Java, środowisko programistyczne Android Studio czy technologię bazodanową Room Persistance Library. Praca przedstawia również możliwość wykorzystania wbudowanych funkcji urządzeń mobilnych takie jak wysyłanie i przechwytywanie wiadomości tekstowych (SMS), działanie w tle czy powiadomień w celu przeprowadzenia pełnego procesu obsługi zleceń, począwszy od jego przyjęcia do realizacji. Zawarty jest w niej także opis okien i komunikatów wyświetlanych w aplikacji co służy jako wstępna instrukcja obsługi systemu dla użytkownika oraz niektóre rozwiązania techniczne wykorzystane w celu realizacji określonych funkcjonalności.

Słowa kluczowe:

Programowanie, aplikacja mobilna, krótka wiadomość tekstowa (SMS), zamówienie, klient, dostawa, produkty rolnicze

Thesis title:

System for handling sales and deliveries of agricultural products and preserves.

Abstract:

The aim of the thesis is to present the design and implementation process of a software application for the sale and delivery of agricultural products and preserves, which is designed for mobile devices. It discusses the tools used to develop the application, such as the Java programming language, the Android Studio development environment, and the Room Persistance Library database technology. The work also presents the possibility of using the built-in functions of mobile devices such as sending and capturing text messages (SMS), background tasks or notifications in order to complete the entire process of handling orders from acceptance to fulfilment. It also includes a description of the windows and messages displayed in the application, which serves as a user manual for the system, and certain technical solutions used to implement particular functionalities.

Keywords:

Programming, mobile application, short message service (SMS), order, customer, delivery, agricultural products

Spis treści

1. Wstęp

We współczesnym intensywnie rozwijającym się świecie, elementem towarzyszącym człowiekowi w każdym aspekcie życia są systemy informatyczne. Definiuje się je jako zbiór powiązanych ze sobą elementów, których funkcją jest przetwarzanie danych z wykorzystaniem sprzętu i oprogramowania komputerowego [1]. Aplikacje i systemy dedykowane różnym dziedzinom życia są integralną częścią codziennego funkcjonowania społeczeństwa. Można je spotkać w przemyśle przy projektowaniu i zarządzaniu procesami produkcyjnymi, administracji publicznej przy elektronicznym dostępie do instytucji oraz handlu przy elektronicznym składaniu zamówień czy zarządzaniu magazynem. Mają one na celu automatyzację procesów, poprawę efektywności oraz zwiększenie dostępności do informacji.

Rozwój technologii sprawia, że do możliwości jakie dają te systemy mamy dostęp z poziomu urządzenia mobilnego znajdującego się w naszych kieszeniach – smartfonów. Są to urządzenia rozszerzające funkcjonalności jakie posiadają zwykłe telefony komórkowe. Oprócz możliwości wykonywania połączeń oraz wysyłania wiadomości smartfony oferują między innymi dotykowy ekran, dostęp do Internetu czy możliwość pobierania i instalowania nowego oprogramowania. Mając smartfon zawsze przy sobie coraz więcej osób decyduje się na robienie zakupów używając do tego przeglądarek internetowych lub odpowiedniego oprogramowania. Korzystając z dedykowanych stron internetowych lub aplikacji producentów można w łatwy sposób, nie przerywając codziennych zajęć zamówić odzież, produkty spożywcze czy dania z restauracji. Po drugiej stronie natomiast osoby lub firmy używają oprogramowania, które pomaga im w procesach przyjmowania i realizacji składanych przez klientów zamówień jak również w dostarczaniu ich w docelowe miejsce.

Osoby posiadające gospodarstwa rolne, w których uprawiają ziemię i hodują zwierzęta często sprzedają wytworzone produkty innym. Przy stale wzrastającej liczbie zamówień i klientów, organizacja oraz zarządzanie nimi staje się problematyczna. W poniższej pracy przedstawiono proces projektowania i implementacji oprogramowania na urządzenia mobilne, służącego do obsługi sprzedaży i dostaw produktów i przetworów rolnych.

W rozdziale drugim tej pracy przedstawiono rodzaje występujących aplikacji. Opisano alternatywne rozwiązania poruszanego problemu oraz przeprowadzono analizę tematu.

Rozdział trzeci skupia się na wymaganiach i narzędziach pracy. Przedstawiono w nim zarówno wymagania funkcjonalne jak i niefunkcjonalne oraz opis przypadków użycia systemu. Opisano tam również narzędzia oraz metody wykorzystane przy projektowaniu i tworzeniu aplikacji.

Rozdział czwarty to specyfikacja zewnętrzna systemu. Zostały w nim przedstawione szczegółowe przykłady działania systemu, wyświetlane okna oraz komunikaty.

W rozdziale piątym opisano specyfikację wewnętrzną. Opisano tam architekturę systemu i struktury danych w nim występujące. Przedstawiony został również schemat i organizacja bazy danych.

Rozdział szósty skoncentrowany jest na weryfikacji i walidacji systemu. Opisano w nim platformy testowe oraz sposoby testowania aplikacji.

W rozdziale siódmym przeprowadzono podsumowanie pracy nad systemem. Znajdą się tam wnioski z osiągniętych rezultatów oraz przedstawiono możliwości dalszych prac rozwojowych nad aplikacją.

1.1. Cel pracy

Celem niniejszej pracy jest zaprojektowanie oraz zaimplementowanie aplikacji do obsługi sprzedaży i dostaw produktów rolnych, która będzie pomagała użytkownikowi w przyjmowaniu, zarządzaniu oraz realizacji zleceń.

1.2. Zakres pracy

Zakres pracy obejmuję: analizę problemu w celu wyodrębnienia wymagań oraz przypadków użycia, zaprojektowanie relacyjnej bazy danych, implementację wszystkich funkcjonalności oraz zaprojektowania przejrzystego i intuicyjnego interfejsu użytkownika.

2. Analiza tematu

System opracowywany w poniższej pracy należy do grupy oprogramowania użytkowego zwanego inaczej aplikacją. Jest to program umożliwiający kontakt użytkownika z komputerem przy pomocy interfejsu graficznego lub wiersza poleceń. Zwykle aplikacje projektowane są do realizacji konkretnych zadań, dostarczania usług jego użytkownikom lub organizacjom oraz rozwiązywania określonych problemów. Ich głównym celem jest automatyzacja i poprawa efektywności wykonywanych zadań lub zapewnienie rozrywki.

2.1. Typy aplikacji

Wyróżnia się kilka typów aplikacji. Każdy z nich charakteryzuje się unikalnymi zastosowaniami oraz wymaganiami sprzętowymi i technologicznymi. Aplikacje możemy podzielić na:

- Aplikacje desktopowe – to oprogramowanie instalowane bezpośrednio na dysku komputera stacjonarnego bądź laptopa. Do jej sprawnego działania należy spełnić wymagania systemowe takie jak odpowiedni system operacyjny zainstalowany na komputerze (np. Windows lub Linux) lub sprzętowymi np. wymagane miejsce na dysku.
- Aplikacje webowe – są oprogramowaniem realizującym swoje funkcjonalności wykorzystując do tego przeglądarkę internetową. Aplikacje te do działania potrzebują dostępu do Internetu. Zaletą tego typu aplikacji jest dostępność. Można z niej korzystać w przypadku dostępności połączenia internetowego, niezależnie od wykorzystywanego urządzenia – komputera lub smartfonu – bądź systemu operacyjnego.
- Aplikacje mobilne – to oprogramowanie dedykowane na urządzenia przenośne takie jak smartfon czy tablet. Podobnie jak w przypadku aplikacji desktopowych, do ich działania znaczące są parametry techniczne, ale również system operacyjny urządzenia. Dodatkowo na sposób realizacji technicznej aplikacje mobilne można podzielić na:

- Aplikacje natywne – są budowane z przeznaczeniem na tylko jeden system operacyjny urządzenia. Wiąże się to z tym, że dla każdej platformy (różnych systemów operacyjnych urządzeń mobilnych np. Android, iOS) aplikacja musi być zaimplementowana w odpowiednim języku programowania. Cechują się bezpośrednim dostępem do funkcji urządzenia i domyślnych elementów interfejsu użytkownika [2].
- Aplikacje hybrydowe – łączą w sobie elementy zarówno aplikacji natywnych i webowych. Są implementowane przy użyciu technologii webowych a następnie opakowywane w formę aplikacji natywnej. Dzięki temu aplikacja taka może być instalowana na urządzeniu mobilnym tak jak natywna a jednocześnie zapewniona jest wieloplatformowość i możliwość działania na różnych systemach operacyjnych [2].

2.2. Motyw stworzenia pracy

Sektor rolniczy gospodarki jest niezwykle istotny dla życia ludzkiego. To dzięki niemu produkowana są artykuły, które następnie zostają sprzedane, przetworzone i ustawiane na sklepowych półkach by ostatecznie trafić w ręce klientów. Jednak nie wszystkie gospodarstwa produkują żywność na dużą skalę by sprzedawać artykuły hurtowo. Niektóre osoby prowadzące swoje hodowle rozpoczynają sprzedaż pojedynczych produktów do osób ze swojego otoczenia to jest rodziny, znajomych bądź sąsiadów. Najczęściej oferowanymi artykułami są owoce i warzywa jak np. ziemniaki, pomidory czy marchewki jak również produkty odzwierzęce takie jak jajka czy mleko.

W przypadku niedużej liczby zamówień zarządzanie nimi nie stanowi problemu. Jednym z rozwiązań, które pozwala na ich organizacje bez ingerencji programów komputerowych jest prowadzenie notatek w papierowym zeszycie. Może ono polegać na przetrzymywaniu listy klientów wraz z ich danymi takimi jak numer telefonu i adres zamieszkania na jednej ze stron, listy produktów wraz z cenami i ilością na magazynie na drugiej oraz listą zleceń na innej stronie. W przypadku gdy zamówienie zostaje zrealizowane następuje jego oznaczenie lub przekreślenie. Mimo iż rozwiązanie to pozwala na przechowywanie wszystkie potrzebne informacje to jest ono problematyczne. Osoba prowadząca notatki zmuszona jest do prowadzenia kalkulacji cen poszczególnych zamówień co w ostateczności może skutkować pomyłkami w obliczeniach. Kolejnym istotnym problemem jest przechowywanie informacji o stanie magazynowym poszczególnych produktów. W takim przypadku każda realizacja zamówienia lub

uzupełnienie magazynu skutkuje zmianą ilości artykułów a co za tym idzie zmianą na jednej ze stron w zeszycie.

Częściowym rozwiązaniem problemu może być wykorzystanie programów komputerowych jakimi są arkusze kalkulacyjne. Przykładem takiego programu może być Microsoft Excel produkowany przez firmę Microsoft dla systemów operacyjnych Windows, macOS, iOS czy Android. Zasada działania takiego rozwiązania jest analogiczna do notatek prowadzonych w papierowym zeszycie, ale tym razem z aspektem technologicznym. Arkusze programu Microsoft Excel mogą być wykorzystywane jak kartki papieru i przetrzymywać w tabelach listy klientów, artykułów i zamówień co pozwala na uproszczenie procesu organizacji i zarządzania sprzedażą produktów. Dodatkową zaletą jest możliwość edytowania komórek arkusza dzięki czemu w prosty sposób można wprowadzać konieczne zmiany co umożliwia utrzymanie porządku w zapiskach.

Oba te rozwiązania mają swoje zalety i częściowo pomagają w prowadzeniu i zarządzaniu sprzedażą. Kolejnym ważnym aspektem poruszanego problemu jest jednak sam proces przyjmowania zamówień. Praca w gospodarstwie wymaga wkładu dużej ilości zarówno czasu jak i uwagi. Odbieranie połączeń telefonicznych lub wiadomości i notowanie zamówień poszczególnych klientów może pochłonąć znaczącą część dnia, którą można poświęcić pracy. Dodatkowo osoby chcące złożyć zamówienie osobiście mogłyby nie być w stanie tego zrobić ze względu na nieobecność rolnika w domu. Zatem kluczowym elementem jest wybór takiego typu oprogramowania, które pozwoli użytkownikowi na przyjmowanie nowych zamówień bez ingerencji w jego codzienne zajęcia. W poniższej pracy został zaprojektowany system zarządzania sprzedażą produktów przeznaczony na urządzenia mobilne. Dzięki takiemu rozwiązaniu, system może korzystać z wbudowanych funkcjonalności jakie oferują telefony i smartfony. Jednym z nich jest możliwość składania zamówień przez krótkie wiadomości tekstowe (ang. Short Message Service - SMS). System przechwytując odpowiednio oznaczone wiadomości pozwala rolnikowi na wykonywanie swoich prac bez zbędnych przerw z możliwością późniejszego ich obejrzenia. To na ich podstawie użytkownik tworzy nowe zamówienia wybierając konkretne produkty oraz ich ilość. Kolejną wbudowaną funkcją, którą znajduje zastosowanie w systemie są powiadomienia. Urządzenia mobilne są istotnym elementem życia człowieka i mimo iż nie używamy ich cały czas to znajdują się one w naszych kieszeniach. Powiadomienia wysyłane przez system na ekran użytkownika w konkretnych sytuacjach np. przechwycenie nowej wiadomości, dostarczą taką informacje bez konieczności otwierania aplikacji mobilnej.

Ostatecznie funkcjonalności, które oferuje poprzednie rozwiązanie wykorzystujące arkusz kalkulacyjny zostaną zebrane w systemie. Wszystkie informacje na temat klientów, aktualnych jak i już zrealizowanych zamówień oraz stanu magazynowego będą

przechowywane w pamięci urządzenia mobilnego. Przeliczanie wartości zamówień oraz aktualizacja zapasów pozwoli na znaczne uproszczenie oraz automatyzacje, organizację oraz zarządzanie sprzedażą produktów dla klientów.

3. Wymagania i narzędzia

Fundamentalnym aspektem pracy jest określenie narzędzi wykorzystywanych w trakcie realizacji projektu jak i wyszczególnienie jego wymagań. Rozdział ten skupia się na przedstawieniu tych dwóch obszarów. Zostały w nim omówione elementy takie jak wykorzystywane środowisko programistyczne, język programowania, technologie bazodanowe oraz programy usprawniające proces projektowania systemu. Dalsza część rozdziału dotyczy szczegółowego określenia wymagań funkcjonalnych i niefunkcjonalnych stawianych przed projektem.

3.1. Środowisko programistyczne

Istnieje szeroka gama narzędzi wykorzystywanych do implementacji aplikacji mobilnych. Aby zawęzić zakres prac zdecydowano, że oprogramowanie projektowane w poniższej pracy będzie natywną aplikacją mobilną dedykowaną na system operacyjny Android. Wybór jest także podyktowany możliwością testowania programu na rzeczywistym urządzeniu mobilnym w posiadaniu autora pracy.

W trakcie pracy wykorzystano środowisko programistyczne Android Studio, które jest oficjalnym, wspieranym i rozwijanym przez firmę Google narzędziem do programowania aplikacji mobilnych na ten system operacyjny. Obsługuje ono głównie dwa języki: **Java** oraz **Kotlin**, umożliwiając programistom wybór w zależności od preferencji bądź wymagań projektu. Zestaw narzędzi deweloperskich takich jak debuger (ang. **Debugger**) czy edytor widoków (ang. Layout Editor) oferowanych przez środowisko znacznie ułatwia tworzenie, debugowanie, wizualizację i optymalizację aplikacji. Dodatkowo wbudowane narzędzia emulacyjne pozwalają na testowanie aplikacji na różnych wersjach systemu Android i różnych urządzeniach bez konieczności posiadania fizycznego urządzenia. To zdecydowanie ułatwia proces testowania i dostosowywania aplikacji do różnych ekranów i rozdzielczości.

3.2. Język programowania

Jako główny język programowania służący do implementacji systemu wybrano język Java. Oprócz zastosowania przy projektowaniu aplikacji mobilnych opartych o system Android jest szeroko stosowany w różnych dziedzinach jak rozwój oprogramowania korporacyjnego, grach komputerowych czy aplikacjach webowych. Charakteryzuje się obiektowością co oznacza, że pisane w Javie programy oparte są o koncepcje klas i obiektów, którą łączą dane i metody na nich operujące a także automatycznym zarządzaniem pamięcią. Dzięki temu programista nie musi ręcznie alokować i zwalniać pamięci a **Garbage Collector** zajmie się usunięciem nieużywanych obiektów co ułatwia unikanie wycieków pamięci. Jest także językiem silnie typowanym co oznacza, że typ zmiennych musi być znany podczas kompilacji a także nie zmieniany dynamicznie w takcie działania programu [3]. Dodatkowo obszerne **API** (ang. **application programming interface**) upraszcza prace programistów dzięki możliwości korzystania z przygotowanych, gotowych bibliotek i komponentów.

W maju 2017 roku na konferencji Google I/O oficjalnie ogłoszono wsparcie dla drugiego języka programowania aplikacji mobilnych na platformę Android jakim jest Kotlin [4]. Motywacją twórców języka do jego stworzenia była chęć eliminacji błędów powstałych podczas projektowania języka Java a także ułatwienie procesu implementacji aplikacji mobilnych [5]. Jak podają autorzy języka w dokumentacji, aplikacje napisane w kotlinie charakteryzują się krótszym czasem wykonania. Dodatkowym aspektem jest łatwa możliwość migracji kodu źródłowego z Javy do Kotlina a ponadto oba te języki są do siebie podobne oraz generują zbliżony kod bajtowy [6].

Mimo wszystkich atutów jakie daje Kotlin zdecydowano pozostać przy starszym języku programowania jakim jest Java. Motywacją do takiego wyboru była lepsza znajomość języka Java przez autora pracy oraz doświadczenie w pracy z językiem. Wcześniejsza realizacja projektów programistycznych w Javie wpływa zarówno na jakość kodu źródłowego jak i na szybki postęp prac implementacyjnych.

3.3. Baza danych

W prezentowanym projekcie zdecydowano, aby zastosować lokalną bazę danych. Jest to rodzaj bazy, która jest przechowywana i zarządzana bezpośrednio na urządzeniu mobilnym na którym działa aplikacja. W przeciwieństwie do baz danych znajdujących się na zewnętrznych serwerach, do lokalnej bazy dostęp posiadają tylko aplikacje na danym urządzeniu. Takie rozwiązanie gwarantuje wysoką wydajność poprzez szybki dostęp do

przechowywanych danych i możliwość korzystania z nich w przypadku braku połączenia z Internetem.

W aplikacji wykorzystano system do zarządzania relacyjnymi bazami danych **SQLite,** natomiast do jej obsługi użyto **Room Persistance Library** z pakietu **Android Jetpack**. Jest to biblioteka dostarczająca warstwę abstrakcji na **SQLite** co umożliwia płynny i prosty dostęp do bazy danych. Jej głównymi cechami charakterystycznymi jest weryfikacja zapytań **SQL** w czasie kompilacji oraz możliwość wykorzystania adnotacji minimalizujących powtarzalność kodu. [7]

3.4. Wymagania funkcjonalne

Po przeprowadzeniu analizy tematu wyszczególniono następujące wymagania funkcjonalne dla opracowywanego systemu:

- Przyjmowanie nowych zleceń – system ma przechwytywać wiadomości SMS z zamówieniami klientów, które będą oznaczone odpowiednimi sygnaturami a następnie zapisywać ich treść oraz numer telefonu nadawcy w relacyjnej bazie danych.
- Stan magazynowy – system ma przechowywać stan magazynowy uwzględniający dostępną liczbę danego produktu oraz jego aktualną cenę, powinien również umożliwiać uzupełnianie zapasów oraz aktualizacje cen produktów.
- Tworzenie zamówień na podstawie wiadomości – system ma umożliwiać na postawie przechwyconych wiadomości tekstowych tworzenie nowych zamówień. Użytkownik aplikacji ma możliwość wyboru z listy danego produktu wraz z jego określoną liczbą oraz dodania go do koszyka.
- Dodawanie nowych klientów – system ma dawać możliwość zapisu danych klientów w relacyjnej bazie danych. Klienci będą identyfikowani na podstawie numeru telefonu. Dodatkowo w bazie danych przechowywane będą informacje o kontrahentach, takie jak imię, nazwisko oraz adres zamieszkania. Ponadto system powinien udostępniać możliwość modyfikacji danych personalnych klientów.
- Przeglądanie danych – użytkownik ma możliwość wyświetlania informacji przechowywanych w relacyjnej bazie danych, w tym:

  - Listy przechwyconych wiadomości – użytkownik widzi wszystkie wiadomości, które zostały przechwycone przez system i na ich podstawie tworzy nowe zamówienia.
  - Listy aktualnych oraz zrealizowanych zleceń – system udostępnia użytkownikowi możliwość przeglądania utworzonych zamówień, także tych które zostały już zrealizowane oraz wyświetlania podsumowania w formie listy produktów razem z ich ilością oraz całkowitą wartością zamówienia.
  - Lista klientów – użytkownik widzi klientów, którzy zostali dodani w systemie oraz ma podgląd w ich danych osobowych (imię i nazwisko, numer telefonu, adres).
  - Lista produktów na magazynie – użytkownik ma możliwość przeglądania dostępnych produktów, ich dostępnej liczby oraz aktualną cenę.
- Realizacja zamówień – system daje możliwość zakończenia procedowanych zleceń. Użytkownik po dostarczeniu produktów do klienta lub odebraniu ich, ma możliwość oznaczenia zamówienia jako zrealizowane lub jeśli transakcja nie doszła do skutku anulowane.
- Raporty sprzedażowe – system powinien dawać możliwość generowania raportów na temat wyników sprzedaży w określonym czasie.

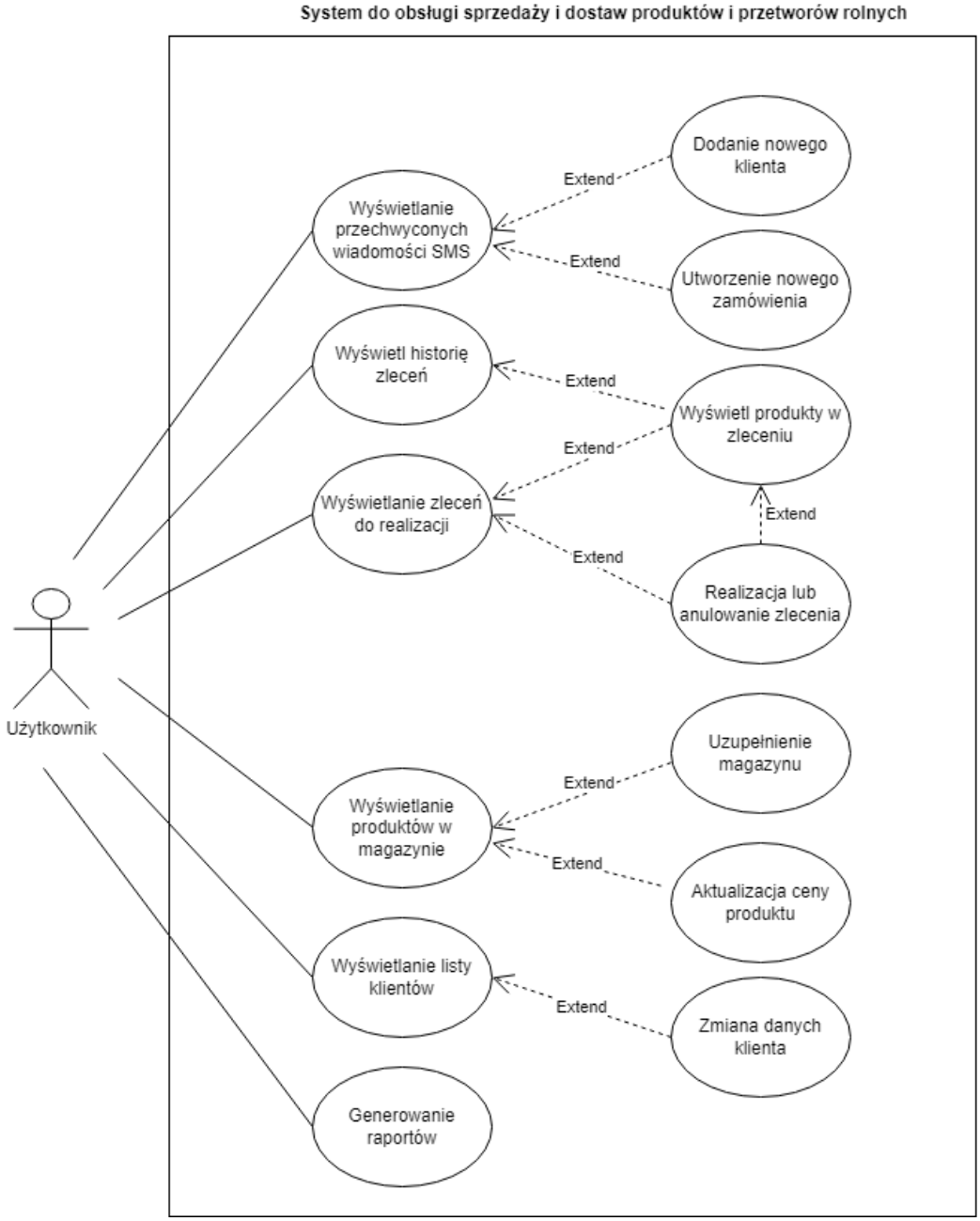
3.5. Wymagania niefunkcjonalne

Równie istotne dla prawidłowego działania systemu są wymagania niefunkcjonalne- mają one wpływ na skuteczność, bezpieczeństwo i wydajność zaprojektowanego rozwiązania. Dla systemu obsługi sprzedaży opracowywanego w tej pracy wyróżniono następujące wymagania niefunkcjonalne:

- Interfejs użytkownika – system powinien być estetycznie wykonany pod względem wizualnym. Wszystkie funkcje dostępne w systemie powinny być czytelne a przemieszczanie się po różnych dostępnych oknach ma być proste i intuicyjne.
- Rozszerzalność – system ma być zaprojektowany z myślą o łatwym dodawaniu nowych funkcjonalności bez względnie dużych ingerencji w dotychczasowy kod źródłowy.

- Bezpieczeństwo – w trakcie przechwytywania wiadomości SMS system powinien przechwytywać odpowiednie wiadomości. Do relacyjnej bazy danych powinny być zapisywane tylko wiadomości oznaczone konkretnymi sygnaturami, aby system nie przechowywał wiadomości do tego nie przeznaczonych.

3.6. Diagram przypadków użycia



Rysunek 3.1 Diagram przypadków użycia

4. Specyfikacja zewnętrzna

Aby używać oprogramowania w sposób jak został do tego przeznaczony, przeciętny użytkownik powinien posiadać odpowiednią wiedzę w celu jego prawidłowego użytkowania. Rozdział ten zawiera szczegółowe informacje na temat wymagań jakie stawia opracowywany system oraz sposób jego instalacji na urządzeniu mobilnym. Opisano w nim również formę wizualną w jakiej prezentuje się system oraz sposób jej obsługi przez użytkownika, w tym umiejscowienie najważniejszych funkcjonalności oferowanych przez system w interfejsie aplikacji. Dodatkowo w rozdziale przedstawiono działanie systemu w konkretnych przypadkach.

4.1. Wymagania systemowe

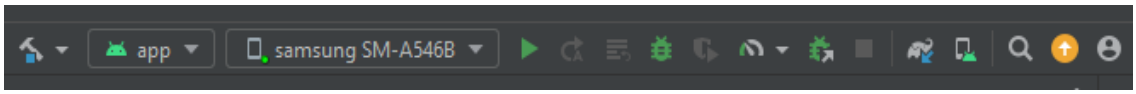
Aby system mógł być poprawnie zainstalowany wymagane jest do tego urządzenie mobilne z systemem operacyjnym Android. Jednak istnieją ograniczenia odnoszące się do wersji systemu. Opracowane w tej pracy rozwiązanie zostało zaimplementowane w oparciu o narzędzia **SDK** (ang. **Software development kit**) w wersji API 33. Android cechuje się natomiast szeroką kompatybilnością wsteczną, czyli zdolnością aplikacji do działania na starszych wersjach systemu operacyjnego. Minimalną możliwą wersją SDK na której możliwe będzie uruchomienie systemu jest wersja API 28. Oznacza to, że najniższą wersją systemu operacyjnego na jakiej można uruchomić opisywany w tej pracy system jest wersja 9.0 systemu Android – „Pie”.

4.2. Instalacja aplikacji

Firma Google oferuje możliwość opublikowania autorskiej aplikacji w serwisie Google Play. W tym celu należy jednak posiadać konto deweloperskie, zakładane przez uiszczanie opłaty w kwocie wymaganej przez firmę Google. Z uwagi na brak konta deweloperskiego przez autora, system można zainstalować korzystając z dołączonego do tej pracy pliku z rozszerzeniem **.apk**. W trakcie pracy nad oprogramowaniem, system instalowany był

bezpośrednio ze środowiska Android Studio. W tym celu należy wykonać następujące kroki:

- Podłączyć urządzanie mobilne z komputerem za pomocą kabla USB
- Włączyć tryb programisty na smartfonie – w tym celu należy w ustawieniach telefonu znaleźć zakładkę „Informacje o telefonie". Następnie należy nacisnąć kilka razy w opcję „Numer wersji" (opcjonalnie „Numer kompilacji" w zależności od urządzenia) aż do momentu włączenia trybu programisty
- W sekcji „Opcje programistyczne" włączyć możliwość debugowania poprzez kabel USB
- W środowisku programistycznym należy upewnić się, że urządzenie jest połączone i wybrane jako docelowe do instalacji. Następnie na pasku narzędziowym wciśnięcie przycisku „Uruchom" (zielona strzałka Rysunek 4.1) skompiluje aplikacje oraz zainstaluje ją na urządzeniu mobilnym.



Rysunek 4.1 Pasek narzędziowy w środowisku Android Studio z podłączonym urządzeniem Samsung A54.

4.3. Użytkownicy w systemie

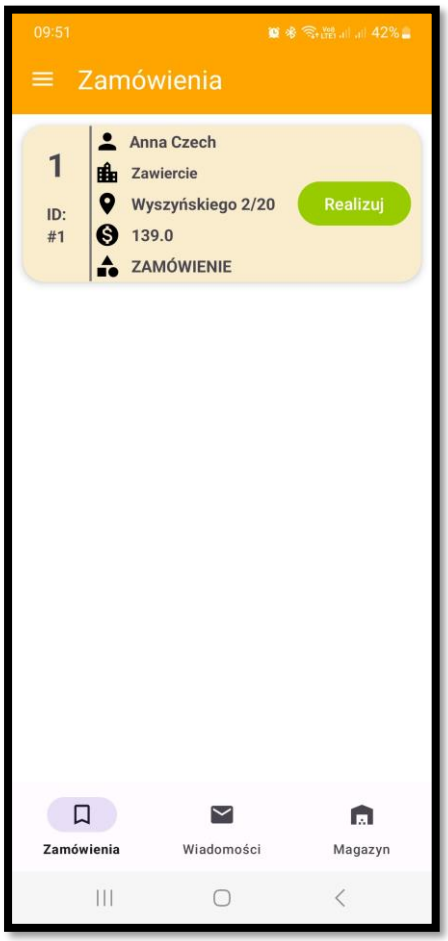
W systemie został wyróżniony tylko jeden typ użytkownika. Za użytkownika przyjmuje się rolnika lub przedsiębiorcę, który zajmuje się sprzedażą produktów i przetworów rolnych jak i ich odbiorem (dostawą). Osoba ta, posiada w systemie możliwości tworzenia nowych zamówień na podstawie wiadomości SMS, realizacji i zarządzaniem zleceniami oraz obsługą magazynu.

4.4. Interfejs użytkownika

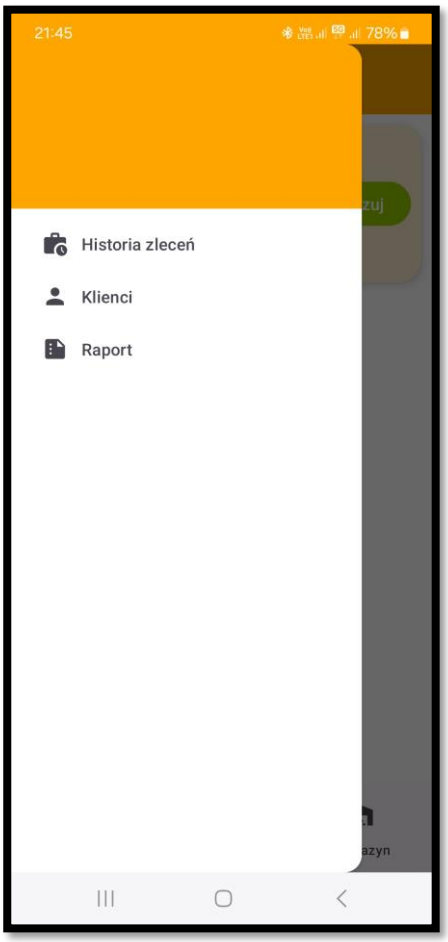
Graficzny interfejs użytkownika w systemie został zaprojektowany w staranny sposób. Jednym z jego głównych założeń była możliwość prostej i intuicyjnej nawigacji pomiędzy poszczególnymi funkcjonalnościami oraz oknami. Dodatkowym celem było zapewnienie bardzo dobrej widoczności wszystkich kontrolek oraz możliwość przechodzenia do różnych okien z jednego miejsca w systemie.

Po instalacji systemu na urządzaniu mobilnym otwierane jest okno startowe (Rysunek 4.2). Z tego miejsca użytkownik dzięki dwóm panelom ma możliwość dostania się do wszystkich funkcjonalności jakie oferuje omawiany program. W dolnej części znajduje się panel nawigacyjny, który pozwala na proste przełączanie się między docelowymi oknami. W tym celu zastosowano komponent *BottomNavigationView*, który udostępnia środowisko programistyczne. Pozwala on na wykorzystanie od trzech do pięciu przycisków przekierowujących użytkownika do kolejnych okien. Dodatkowo daje on możliwość konfiguracji pola o ikonę dzięki czemu przemieszczanie się po oknach systemu staje się bardziej intuicyjne.

Drugim elementem pozwalającym na dotarcie do innych funkcjonalności jest panel boczny wysuwany z lewej strony ekranu za pomocą przycisku na górnym pasku narzędziowym systemu (przycisk oznaczony trzema poziomymi liniami na rysunku 4.2). Po naciśnięciu przycisku na ekran, wysuwany jest panel (Rysunek 4.3) dzięki któremu użytkownik może korzystać z pozostałych możliwości programu.



Rysunek 4.2 Główny widok systemu – okno zamówień.



Rysunek 4.3 Panel boczny - Navigation View.

Rozwiązanie takie jest szeroko stosowane przez programistów i można je spotkać w wielu popularnych aplikacjach mobilnych. Pozwala to na efektywną nawigację pomiędzy poszczególnymi oknami oraz oferuje proste możliwości rozwoju poprzez dodanie kolejnych przycisków przekierowujących.

Dolny panel nawigacyjny pozwala na dotarcie do trzech kluczowych okien w systemie. Są to widoki związane bezpośrednio z procesem przyjmowania nowych zleceń (Rysunek 4.4), ich realizacją (Rysunek 4.2) oraz zarządzaniem stanem magazynowym (Rysunek 4.5). Panel ten oferuje możliwość dostania się do okien:

Rysunek 4.4 Okno wiadomości.

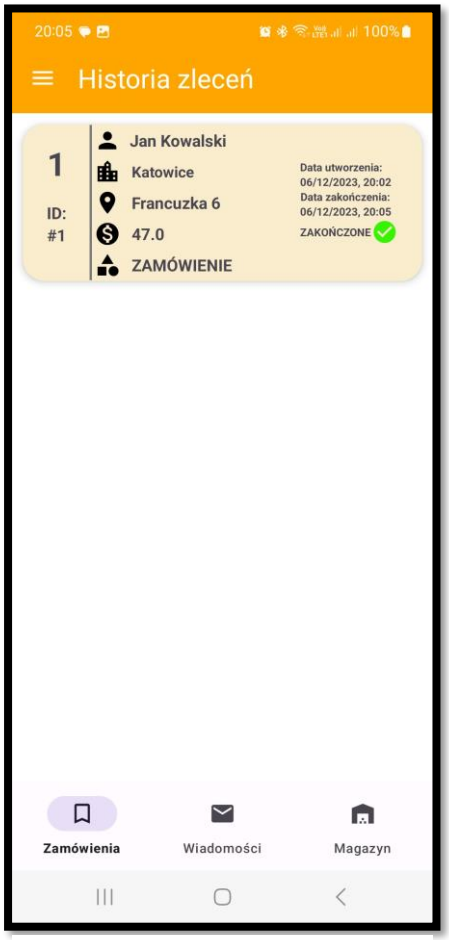
Rysunek 4.5 Okno magazynu.

- Zamówienia (Rysunek 4.2) – znajdują się tam wszystkie zlecenia, które zostały stworzone w systemie. Na jednym elemencie listy zawarte są podstawowe informacje o zamówieniu, w tym imię i nazwisko klienta, adres dostawy, całkowitą kwotę oraz typ zamówienia.
- Wiadomości (Rysunek 4.4) – przechowywane są tam wiadomości, które zostały przechwycone przez system. W elemencie listy wyświetlany jest typ zlecenia (zamówienie lub dostawa), treść wiadomości oraz numer telefonu klienta.

- Magazyn (Rysunek 4.5) – znajduje się tam lista z produktami oferowanymi przez sprzedającego. Na każdym elemencie listy pokazana jest jego nazwa razem z wizualizującym produkt obrazem dostępnym w sieci [8] i dostępna na magazynie liczba danego artykułu oraz cena za sztukę lub kilogram.

Wszystkie z tych trzech okien umożliwiają korzystanie z głównych funkcjonalności jakie oferuję system. Okna te i możliwości jakie dają użytkownikowi zostaną szczegółowo omówione w rozdziałach **4.4.1 Magazyn**, **4.4.2 Wiadomości** oraz **4.4.3 Zamówienia**.

Wysuwany z lewej strony panel boczny (Rysunek 4.3) pozwala na uruchomienie następnych funkcjonalności programu. Znajdują się w nim przyciski otwierające okna związane z klientami zapisanymi w systemie, zleceniami, które zostały już zakończone oraz okna generowania raportu. Przekierowują one do okien:

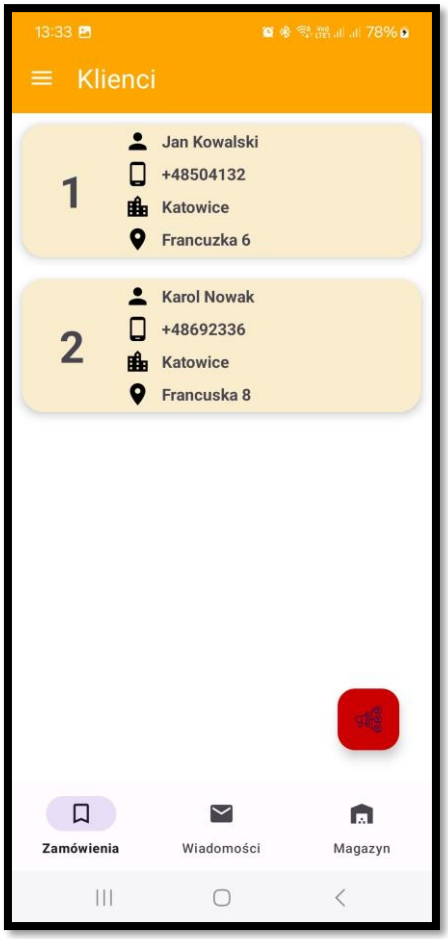


Rysunek 4.6 Okno historii zleceń.

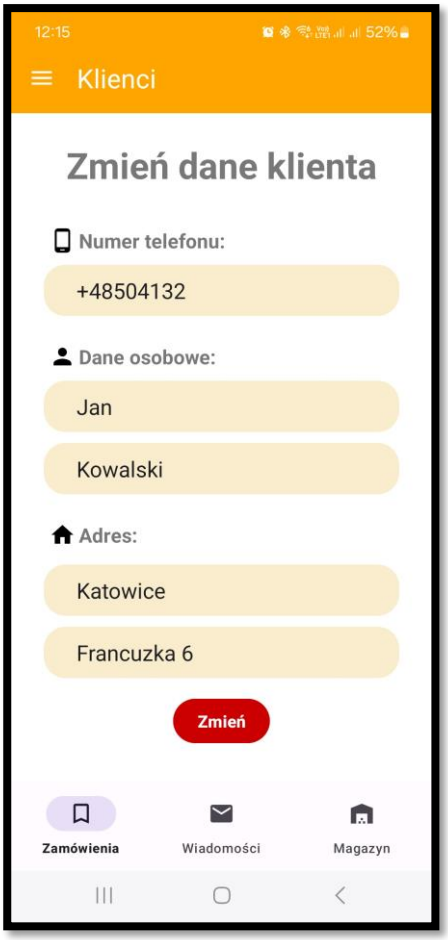
Rysunek 4.7 Okno szczegółów zlecenia.

- Historia zleceń (Rysunek 4.6) – okno to zawiera listę z zleceniami, które zostały już zakończone (zrealizowane lub anulowane). W elemencie listy znajdują się natomiast informacje o klienci, adres doręczenia (lub odbioru), całkowitą kwotę zlecenia, typ (zamówienie lub dostawa) oraz zarówno datę utworzenia zlecenia

oraz jego realizacji. Dodatkowo naciśnięcie elementu listy wyświetla szczegóły zlecenia co pozwala na obserwacje produktów które zostały zamówione, cenę w momencie utworzenia zlecenia oraz ich ilości (Rysunek 4.7).

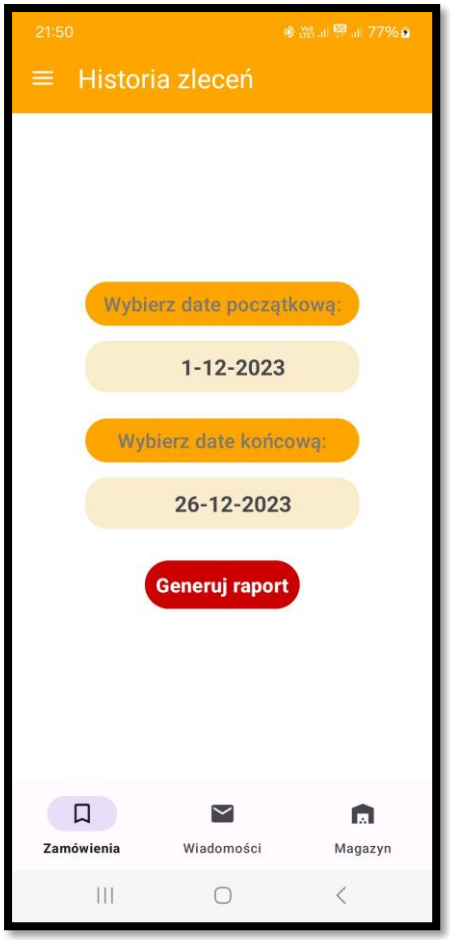


Rysunek 4.8 Okno klienci.



Rysunek 4.9 Okno zmiany danych klienta.

- Klienci (Rysunek 4.8) – zawiera listę klientów, którzy zostali dodani do systemu i są zapisani w relacyjnej bazie danych. Znajdują się tu informacje o ich danych takie jak imię i nazwisko, numer telefonu, oraz adres, na który dostarczane lub z którego odbierane są produkty. Wybranie konkretnej pozycji na liście otwiera okno, które pozwala natomiast na zmianę wszystkich informacji o kliencie (Rysunek 4.9). Dodatkowo w panelu „Klienci" istnieje możliwość zareklamowania sklepu do wszystkich klientów w systemie poprzez wysłanie wiadomości SMS z produktami dostępnymi do kupienia oraz z ich aktualnymi cenami.



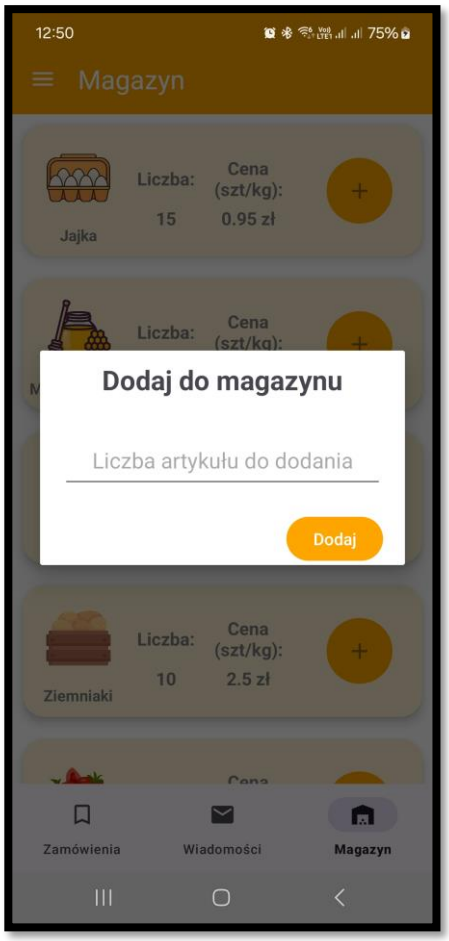
Rysunek 4.10 Okno generowania raportu.

Rysunek 4.11 Okno dialogowe z kalendarzem.

- Raport (Rysunek 4.10) – pozwala na generowanie raportu ilości sprzedanych produktów w wybranym przedziale czasowym. Widok daje możliwość wyboru zarówno daty początkowej jak i końcowej przedziału czasowego przy pomocy okna dialogowego kalendarza (Rysunek 4.11). Po naciśnięciu przycisku „Generuj raport” tworzony oraz otwierany jest dokument tekstowy z pełnym raportem w formacie pdf. Proces oraz dokument wynikowy opisane zostały w rozdziale **5.8 Generowanie Raportu** tej pracy.

4.4.1. Magazyn

Okno magazynu pozwala użytkownikowi na kontrolowanie obecnego stanu magazynowego dostępnych produktów. Dzięki temu w głównym widoku istnieje możliwość obserwacji liczby oraz aktualnych cen artykułów. Naturalnym zjawiskiem jest więc uzupełnianie zapasów magazynowych w czasie sezonowych zbiorów, codziennych obrządków lub dostaw od sprzedawców. Projektując system przewidziano dwie funkcjonalności pozwalające na edycje danych magazynowych.



Rysunek 4.12 Okno dialogowe uzupełniania magazynu.

Rysunek 4.13 Okno edycji informacji o produkcie.

Pierwszą z nich jest możliwość uzupełniania magazynu przez zwiększanie liczby danego produktu. Jest to możliwe na dwa sposoby. Pierwszy z nich pokazano na rysunku 4.12. Po wciśnięciu okrągłego przycisku ze znakiem „+” na elemencie listy produktów otwierane jest okno dialogowe z możliwością podania liczby produktu do dodania. Transakcja zakańczana jest przez wciśnięcie przycisku „Dodaj” a użytkownik powiadamiany jest komunikatem *„Dodano artykuł do magazynu!”*

Inną ewentualnością jest otwarcie okna szczegółów produktu (Rysunek 4.13). W systemie można to zrobić poprzez naciśnięcie wybranego produktu w dowolnym

miejscu na liście. Po otwarciu w oknie ukazuje się obraz oraz nazwa produktu, którego szczegóły są wyświetlane oraz dwa edytowalne pola tekstowe (ang. EditText). W oknie szczegółów produktu użytkownik edytując pierwsze pole ma możliwość ustalenia konkretnej liczby danego produktu jaka znajduje się w magazynie.

Wraz ze zmianą popytu i podaży ceny produktów również mogą ulegać zmianie. Omawiane w tym rozdziale okno pozwala także udostępnia taką funkcjonalność. Użytkownik korzystając i edytując drugie z dostępnych edytowalnych pól tekstowych ma możliwość ustalenia nowej ceny oferowanych produktów za sztukę lub za kilogram.

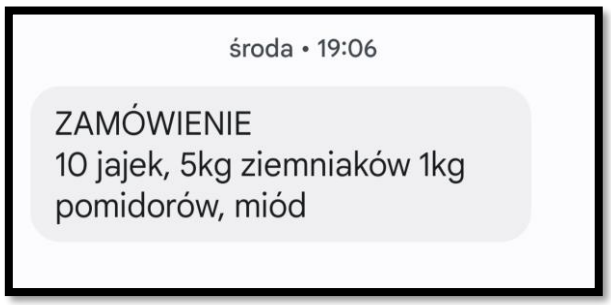
Po wprowadzeniu zamierzonych zmian transakcja realizowana jest poprzez przycisk „Zatwierdź". Po jego naciśnięciu na ekranie urządzenia mobilnego użytkownika mogą zostać wyświetlone trzy scenariusze oraz komunikaty określające rezultat operacji:

- *„Nie wprowadzono żadnych zmian!"* – komunikat wyświetlany w sytuacji, gdy żadne z wartości w edytowalnych polach tekstowych nie ulegnie zmianie.
- *„Wprowadzono nieprawidłowe dane!"* – wyświetlany w sytuacji, gdy zmienione przez użytkownika dane zostały podane w niewłaściwej formie.
- *„Zaktualizowano szczegóły produktu"* – komunikat pojawiający się na ekranie w sytuacji, gdy transakcja została zrealizowana oraz poprawnie zaktualizowano liczbę i cenę wybranego produktu. Sytuacja ta skutkuje również zamknięciem okna szczegółów produktu oraz powrót do głównego okna magazynu.

4.4.2. Wiadomości

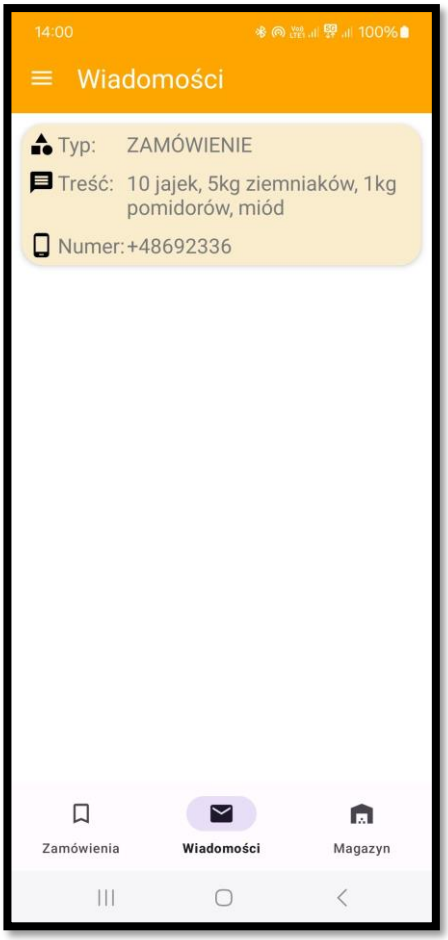
Wiadomości tekstowe są fundamentalnym składnikiem działania systemu. To na ich podstawie tworzone oraz realizowane są zamówienia. Otwarta lub działające w tle aplikacja przechwytuje odbierane wiadomości SMS na podstawie sygnatury:

- *„ZAMÓWIENIE"* – dla zleceń w których klient kupuje produkty.
- *„DOSTAWA"* – dla zleceń w których klient chce sprzedać swoje własne wyprodukowane artykuły spożywcze.



Rysunek 4.14 Wiadomość SMS ze zleceniem kupna.

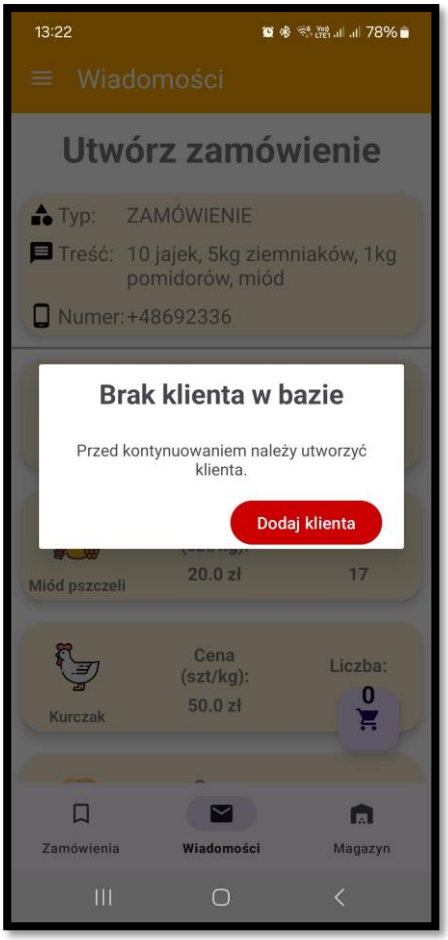
Wiadomości przechwycone i przechowywane w relacyjnej bazie danych wyświetlane są na liście w oknie „Wiadomości” (Rysunek 4.15).



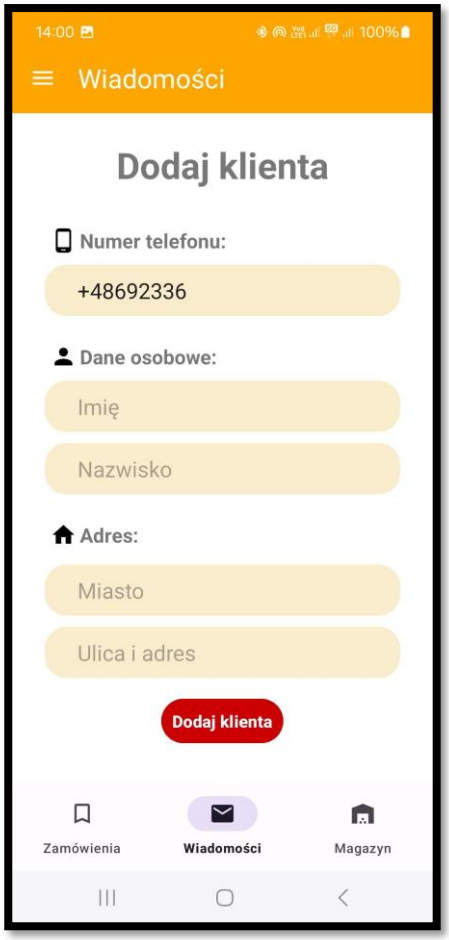
Rysunek 4.15 Okno listy z przechwyconymi wiadomościami.

Na jednym elemencie listy użytkownik jest w stanie zobaczyć typ zlecenia, treści wiadomości oraz numer telefonu klienta. Dalsze procedowanie wiadomości w celu utworzenia zamówienia umożliwione jest poprzez naciśnięcie w dowolne miejsce na wybrany element listy. Po wyborze konkretnej wiadomości i przyciśnięciu na element listy możliwe są dwa scenariusze:

- Brak informacji o kliencie w systemie – konieczność dodania go i wszystkich potrzebnych informacji (numer telefonu, imię i nazwisko, adres) do relacyjnej bazy danych.
- Klient znajduje się w systemie – możliwość kontynuacji i dalszego procedowania zlecenia.



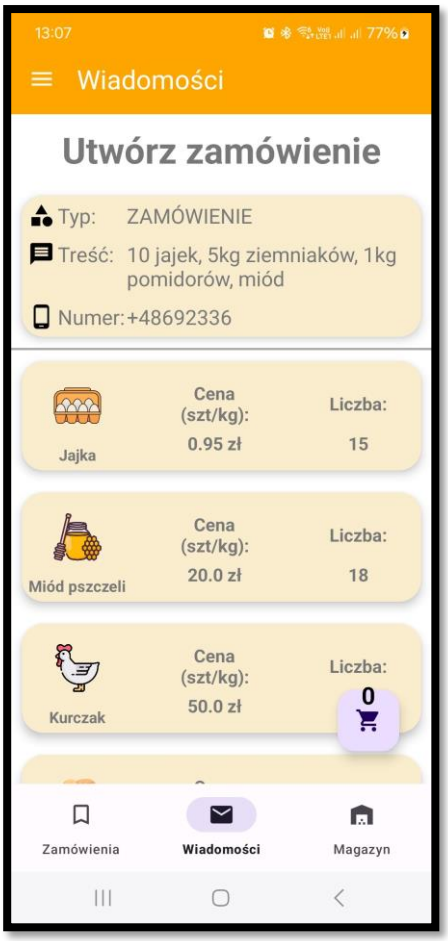
Rysunek 4.16 Okno dialogowe informujące o braku klienta w systemie.



Rysunek 4.17 Okno dodawania klienta do bazy danych

W przypadku braku klienta w bazie danych systemu (Rysunek 4.16), użytkownikowi wyświetlane jest okno dialogowe z informacją o konieczności dodania go, ponieważ bez tego nie możliwe jest dalsze procedowanie zlecenia. Po naciśnięciu przycisku „Dodaj klienta” użytkownikowi otwierane jest okno tworzenia nowego klienta i dodawania go do bazy danych (Rysunek 4.17). Okno to zawiera pięć edytowalnych pól tekstowych (ang. EditText) w które konieczne jest wpisanie danych osobowych klienta – imienia i nazwiska, miasta oraz adresu dostawy. Natomiast numer telefonu zamawiającego usługę wypełniany jest automatycznie na podstawie przechwyconej wiadomości.

Finalizacja działań i dodanie klienta do bazy wykonywane jest przez przycisk „Dodaj klienta”. W przypadku gdy nie wszystkie pola tekstowe są wypełnione naciśnięcie przycisku spowoduje wyświetlenie na ekranie komunikatu *„Uzupełnij wszystkie pola…”* natomiast w przeciwnym razie wyświetlony zostanie komunikat *„Dodano nowego klienta!”*. Poprawne zakończenie operacji skutkuje zamknięciem okna dodawania klienta oraz otwarciem nowego – umożliwiającego utworzenie zamówienia.



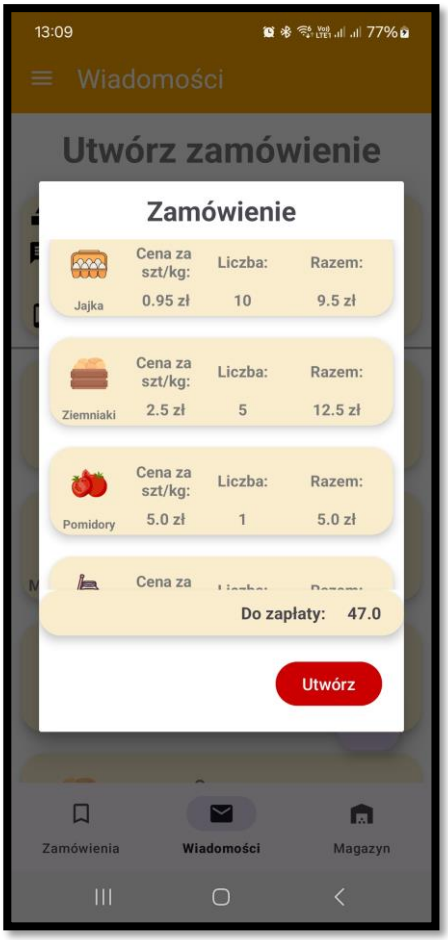
Rysunek 4.18 Okno tworzenia nowego zamówienia.

Rysunek 4.19 Szczegóły produktu.

Aby móc zarządzać i realizować zamówienie w pierwszej kolejności konieczne jest jego utworzenie. W systemie umożliwia to okno „Utwórz zamówienie” (Rysunek 4.18). W oknie tym na podstawie wyszczególnionej wiadomości SMS, produkty dodawane są do zamówienia. Jest to możliwe dzięki liście produktów umieszczonej pod wiadomością, która zawiera wszystkie produkty dostępne na magazynie, ich liczbę oraz cenę za sztukę bądź kilogram. Użytkownik naciskając element (wybrany produkt) na liście, przenoszony jest do okna szczegółów produktu (Rysunek 4.19). W oknie tym za pomocą przycisku inkrementacji „+” i dekrementacji „-” zwiększana lub zmniejszana jest liczba artykułu jaka ma zostać dodana do koszyka oraz wyświetlana jest całkowita wartość dodawanych produktów. Przycisk oznaczony „Dodaj do koszyka” finalizuje operacje dodawania produktu do koszyka oraz zamyka wyświetlane „Szczegóły produktu” wracając do okna poprzedniego – „Utwórz zamówienie”.

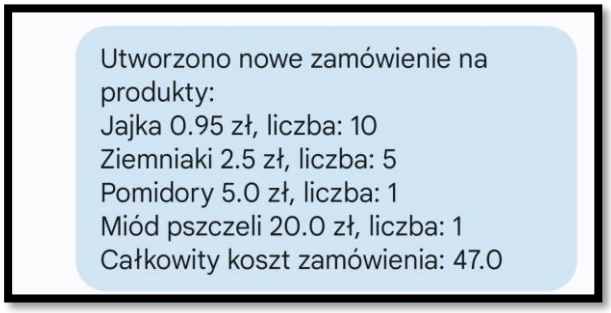
Po skompletowaniu wszystkich produktów, podsumowanie zamówienia otwierane jest przy pomocy przycisku oznaczonym logo wózka sklepowego z liczbą produktów.

Podsumowanie zamówienia (Rysunek 4.20) pozwala na ostateczną weryfikacje produktów dodanych do koszyka. Zawiera listę produktów w koszyku, ich liczbę oraz zarówno cenę poszczególnych artykułów jak i całkowitą wartość zlecenia. W przypadku pomyłki w dodawaniu produktów, przytrzymanie elementu listy pozwala na usunięcie produktu z koszyka po uprzednim potwierdzeniu w wyświetlanym oknie dialogowym z zapytaniem *„Czy na pewno chcesz usunąć produkt z koszyka?”*.



Rysunek 4.20 Okno dialogowe z podsumowaniem zamówienia.

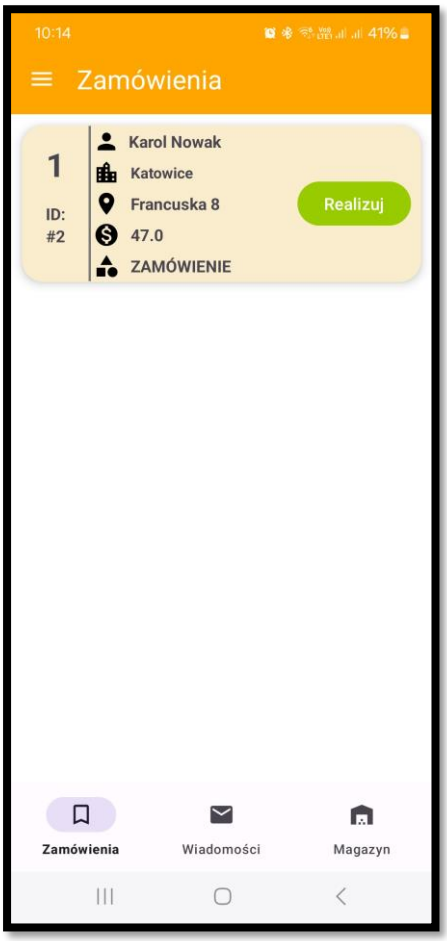
Proces zakańczany jest poprzez przycisk „Utwórz”. Po potwierdzeniu chęci utworzenia zamówienia na ekranie wyświetlany jest komunikat „*Dodano nowe zamówienie!*” oraz jest ono dodawane do relacyjnej bazy danych natomiast przechwycona wiadomość jest z niej usuwana. Jednocześnie w przypadku sprzedaży produktów (typ zlecenia ZAMÓWIENIE) liczba produktów potrzebna do jego realizacji jest dekrementowana z magazynu w celu ich rezerwacji i braku możliwości wykorzystania przy tworzeniu kolejnych zleceń. Dodatkowo po utworzeniu nowego zlecenia na numer telefonu klient wysyłana jest wiadomość SMS z potwierdzeniem (Rysunek 4.21). Od tego momentu dalsze procedowanie zleceń możliwe jest w oknie „Zamówienia”.



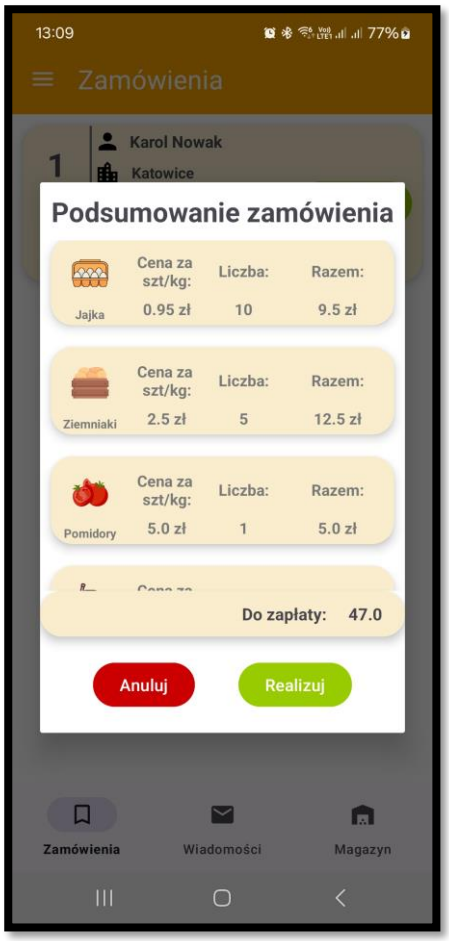
Rysunek 4.21 Wiadomość SMS
z potwierdzeniem zamówienia.

4.4.3. Zamówienia

Okno „Zamówienia” dolnego panelu aplikacji umożliwia zarządzanie oraz ostateczną realizacje zleceń (Rysunek 4.22).



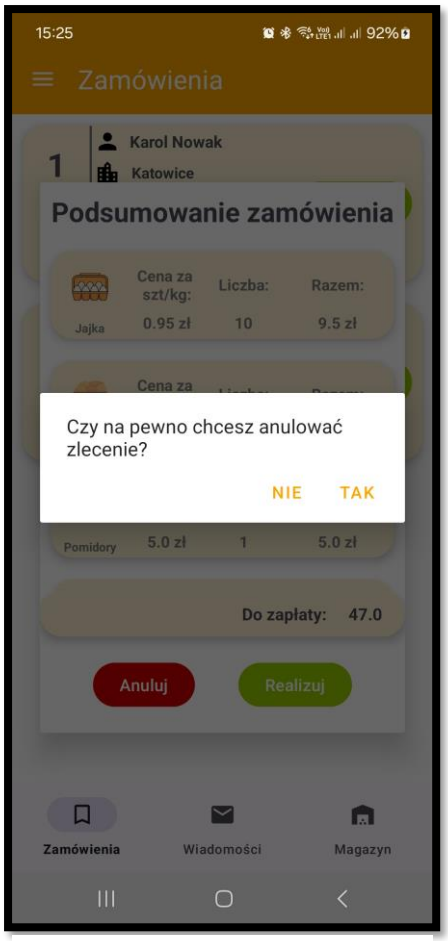
Rysunek 4.22 Okno z listą
zamówień.



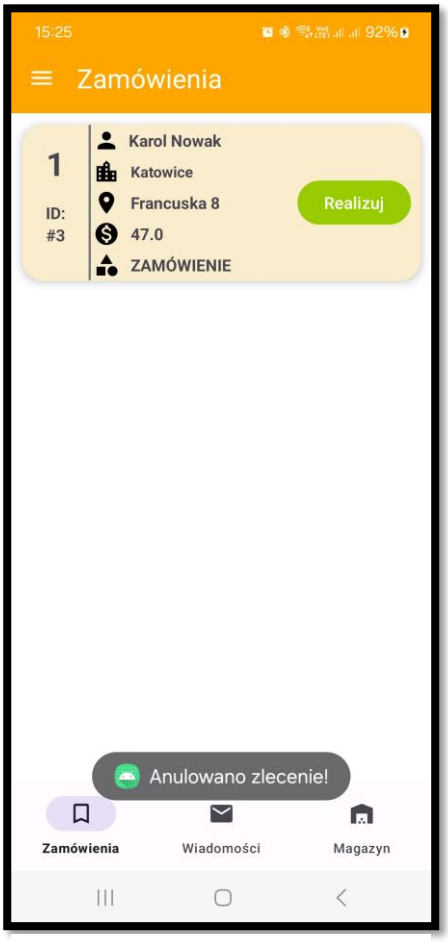
Rysunek 4.23 Okno dialogowe
z podglądem zamówienia.

Omawiane w tym rozdziale okno zawiera listę, w której elementami są utworzone zamówienia. Na każdym elemencie znajduje się numer porządkowy oraz identyfikacyjny (ID) zlecenia, szczegóły dotyczące klienta i adresu doręczenia (imię, nazwisko, miasto i adres) a także całkowitą wartość zlecenia oraz jego typ. Z poziomu listy możliwa jest realizacja zamówienia za pomocą przycisku „Realizuj" po wcześniejszym potwierdzeniu w oknie dialogowym z zapytaniem *„Czy na pewno chcesz zrealizować zlecenie?"*. Po zakończeniu zlecenia element usuwany jest z listy a na ekran użytkownika wyświetlany jest komunikat *„Zrealizowano zlecenie"*.

Naciśnięcie elementu listy sprawia, że wyświetlane jest okno dialogowe z podglądem zamówienia (Rysunek 4.23). W nim wyświetlana jest lista produktów, które zostały zamówione, ich liczba, wartość cząstkowa każdego produktu oraz całkowita wartość zlecenia. Z poziomu podglądu również można zrealizować zamówienie. Podobnie jak w oknie z listą zamówień proces ten odbywa się przy pomocy przycisku „Realizuj" oraz potwierdzenia w oknie dialogowym. Dodatkowo istnieje również możliwość anulowania zlecenia używając przycisku „Anuluj". Chcąc anulować zlecenie podobnie jak w przypadku jego realizacji wymagane jest potwierdzenie w oknie dialogowym z zapytaniem *„Czy na pewno chcesz anulować zlecenie?"* (Rysunek 4.24). Po anulowaniu na ekranie wyświetlany jest komunikat *„Anulowano zlecenie!"* (Rysunek 4.25) a ponadto, gdy jest to zlecenie kupna produktów (typ zlecenia *ZAMÓWIENIE*), magazyn systemu jest uzupełniony o produkty z podsumowania.



Rysunek 4.24 Okno dialogowe z zapytaniem o anulowanie zlecenia.



Rysunek 4.25 Komunikat z potwierdzeniem anulowania zlecenia

5. Specyfikacja wewnętrzna

Dokumentacja każdego projektowanego systemu skupia się na przedstawieniu aspektów wewnętrznych projektu, jego implementacji oraz decyzji technicznych. Ma ona na celu usprawnić zrozumienie poruszanego problemu, wykorzystanych rozwiązań oraz wprowadzenie w kod źródłowy programu. Rozdział ten skupia się na omówieniu technologii, rozwiązań oraz wykorzystywanych metod programowania aplikacji mobilnych w środowisku Android Studio. Zawiera również szczegółowy opis użytej bazy oraz modelu danych. Dodatkowo w rozdziale przedstawiono wykorzystanie mechanizmów pozwalających na korzystanie z wbudowanych możliwości urządzenia mobilnego jak np. obsługa wiadomości SMS oraz powiadomień.

5.1. Architektura systemu

W architekturze android **„Activity”** oraz **„Fragment”** są dwoma kluczowymi komponentami używanymi do tworzenia interfejsu użytkownika i zarządzania cyklem życia całej aplikacji. Klasa aktywności (ang. Activity Class) implementuje jedno pełnowymiarowe okno z graficznym interfejsem użytkownika – jeden ekran. Aplikacje w większości posiadają wiele ekranów co oznacza, że składają się z wielu aktywności, przy czym jeden z nich jest punktem wejścia do programu. Mogą się również komunikować się ze sobą oraz otwierać inne aktywności w celu wykonania działań [9]. W trakcie wykonywania programu **„Activity”** przechodzi przez stadia cyklu życia natomiast programista ma możliwość zarządzania nim poprzez implementacje metod *onCreate(), onStart(), onPause(), onResume(),onStop()* oraz *onDestroy().*

„Fragment” również reprezentuje okno graficznego interfejsu użytkownika które definiuje i zarządza własnym układem oraz posiada własny cykl życia. Jest to modułowy komponent, który daje możliwość ponownego i wielokrotnego użycia w różnych aktywnościach a dodatkowo pozwala na podział interfejsu na mniejsze fragmenty co upraszcza zarządzanie nim. Jednak Fragmenty nie mogą istnieć samodzielnie. Wymagają, aby ich rodzicem (ang. host) była aktywność (Activity) bądź inny fragment [10].

W omawianym w tej pracy systemie zastosowano architekturę wykorzystującą jedną klasę aktywności, fragmenty oraz widoki (ang. layout) opisane za pomocą plików XML.

To podejście nazywane jest w środowisku deweloperskim **Single-Activity Architecture**. Klasa aktywności pełni w nim role głównego kontenera, z którego uruchamiane są fragmenty. Każdy z nich reprezentuje niezależne części interfejsu użytkownika. Mają one własny cykl życia, zarządzają własnym widokiem oraz obsługują logikę z nimi związaną. Główną z cech tej architektury jest łatwość utrzymania. Tylko jedna klasa aktywności przyczynia się do zminimalizowania kodu źródłowego w kontekście jego cyklu życia a dodatkowo ułatwia zrozumienie struktury oraz utrzymanie systemu. Dodatkowo fragmenty poprzez ich ładowanie oraz usuwanie w zależności od stanu aplikacji mobilnej pozwalają na optymalizację oraz bardziej efektywne zarządzanie pamięcią.

W systemie do obsługi sprzedaży i dostaw produktów i przetworów rolnych główną a zarazem jedyną aktywnością jest klasa *MainActivity*, która jest punktem wejścia do programu. Posiada własny plik układu widoku *axtivity_main.xml* **(Rysunek 1 dodatek Źródła)**, który opisuje rozmieszczenie elementów okna na ekranie urządzenia mobilnego. Zdefiniowane komponenty *BottomNavigationView* i *NavigationView*, odpowiadają dolnemu oraz wysuwanemu z lewej strony panelowi nawigacyjnemu i pozwalają użytkownikowi na uruchamianie funkcjonalności oferowanych przez system natomiast układ ramki (*FrameLayout*) to część okna w której są one wyświetlane.

Zadaniem klasy *MainActivity* jest zarządzanie panelami nawigującymi oraz przełączanie okien systemu. Uruchamia fragmenty odpowiedzialne za pozostałą część interfejsu graficznego oraz logiki systemu takie jak wyświetlanie listy zamówień czy produktów na magazynie. Za obsługę i interakcję z dolnym panelem nawigacyjnym odpowiedzialny jest *Słuchacz* (ang. Listener) zaimplementowany i przypisany w metodzie *onCreate()* (Rysunek 5.1). Zajmuje się on porównaniem identyfikatora elementu na panelu nawigacyjnym a następnie uruchomieniem jednego z trzech fragmentów:

- *MessagesFragment* – wyświetlający przechwycone wiadomości
- *OrdersFragment* – wyświetlający zamówienia utworzone w systemie
- *WarehouseFragment* – wyświetlający listę produktów na magazynie

```
binding.bottomNavigationView.setOnItemSelectedListener(item -> {
        if (item.getItemId() == R.id.messages) {
            replaceFragments(new MessagesFragment());
            this.setTitle("Wiadomości");
        } else if (item.getItemId()== R.id.order) {
            replaceFragments(new OrdersFragment());
            this.setTitle("Zamówienia");
        } else if (item.getItemId()==R.id.warehouse) {
            replaceFragments(new WarehouseFragment());
            this.setTitle("Magazyn");
        }
        return true;
    });
```

Rysunek 5.1 Słuchacz dolnego panelu nawigacyjnego.

Za uruchomienie fragmentów odpowiada metoda *replaceFragments()* (Rysunek 5.2). W niej przy pomocy menadżera fragmentów (ang. *FragmentManager*) wykonywana jest transakcja zamiany okna fragmentu w przeznaczonej do tego ramce (*FrameLayout*), na nowy fragment przekazany do metody jako parametr jej wywołania. Dodatkowo w zależności od uruchomionego fragmentu ustawiane jest nowy tytuł okna.

```
1  private void replaceFragments(Fragment fragment){
2      getSupportFragmentManager()
3      .beginTransaction()
4      .setCustomAnimations(
5  androidx.navigation.ui.R.animator.nav_default_enter_anim,
6          androidx.navigation.ui.R.animator.nav_default_exit_anim)
7      .replace(R.id.frame,fragment)
8      .commit();
9  }
```

Rysunek 5.2 Metoda zamiany fragmentu w oknie i jego uruchomienia.

Analogiczne działanie zostało zastosowane w przypadku wysuwanego panelu nawigacyjnego. Nadpisana (ang. Override) metoda *onNavigationItemSelectedd()* w klasie *MainActivity* (Rysunek 5.3) pozwala na uruchomienie fragmentów i ich zmianie w docelowej ramce za pomocą metody *replaceFragments()*, po której następuje zmiana tytułu okna oraz zamknięcie panelu.

```
1  @Override
2  public boolean onNavigationItemSelected(@NonNull MenuItem item) {
3      int id = item.getItemId();
4
5      if(id == R.id.clients){
6          replaceFragments(new ClientsFragment());
7          setTitle("Klienci");
8      } else if (id ==R.id.orderHistory) {
9          replaceFragments(new OrdersHistoryFragment());
10         setTitle("Historia zleceń");
11     }
12     else if (id ==R.id.orderHistory) {
13         replaceFragments(new GenerateReportFragment());
14         setTitle("Historia zleceń");
15     }
16     drawerLayout.closeDrawer(GravityCompat.START);
17     return true;
18 }
```

Rysunek 5.3 Nadpisana metoda onNavigationView uruchamiająca fragmenty z wysuwanego panelu nawigacyjnego.

5.2. Uprawnienia

Kluczowym elementem w systemie Android są uprawnienia, które umożliwiają aplikacji dostęp do funkcji oraz narzędzi urządzenia mobilnego. System uprawnień androida daje użytkownikom możliwość kontrolowania dostępu aplikacji do określonych funkcji. w przypadku oprogramowania omawianego w tej pracy wykorzystywane są uprawnienia umożliwiające kompleksową obsługę wiadomości SMS, ich odbieranie, wysyłanie oraz czytanie, wysyłanie powiadomień na ekran użytkownika oraz zapis i odczyt plików z pamięci zewnętrznej urządzenia mobilnego.

Uprawnienia, które aplikacja mobilna wykorzystuje deklarowane są w pliku *AndroidManifest*. Jest to plik konfiguracyjny znajdujący się w każdym projekcie Android Studio i zawiera informacje na temat struktury i funkcji aplikacji. Oprócz zdefiniowanych uprawnień znajdują się w nim dane konfiguracyjne na temat tytuły oraz ikony aplikacji a także deklaracje wszystkich komponentów takich jak aktywności.

W systemie, aby korzystać z potrzebnych funkcji jakie daje urządzenie mobilne zadeklarowano uprawnienia (Rysunek 5.4):

- *RECEIVE_SMS* – pozwala na odbieranie wiadomości SMS, uprawnienie kluczowe dające możliwość przechwycenia przychodzących wiadomości.
- *READ_SMS* – umożliwia odczytanie treści wiadomości SMS, wykorzystane w celu analizy zawartości wiadomości.
- *SEND_SMS* – daje możliwość wysyłania wiadomości SMS, dzięki niemu system pozwala powiadamiać klientów o utworzonym zamówieniach
- *POST_NOTIFICATIONS* – pozwala na wysyłanie powiadomień, umożliwia informowanie użytkownika w przypadku przechwycenia nowego zlecenia
- *WRITE_EXTERNAL_STORAGE* – zezwala na zapis plików do pamięci urządzenia, wymagane podczas generowania raportu wyników sprzedaży.
- *READ__EXTERNAL_STORAGE* – umożliwia odczyt plików z pamięci urządzenia, konieczne podczas otwierania wygenerowanego raportu.

```
1  <uses-permission android:name="android.permission.RECEIVE_SMS" />
2  <uses-permission android:name="android.permission.READ_SMS" />
3  <uses-permission android:name="android.permission.SEND_SMS" />
4  <uses-permission
5  android:name="android.permission.POST_NOTIFICATIONS" />
6  <uses-permission
7  android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
8  <uses-permission
9  android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

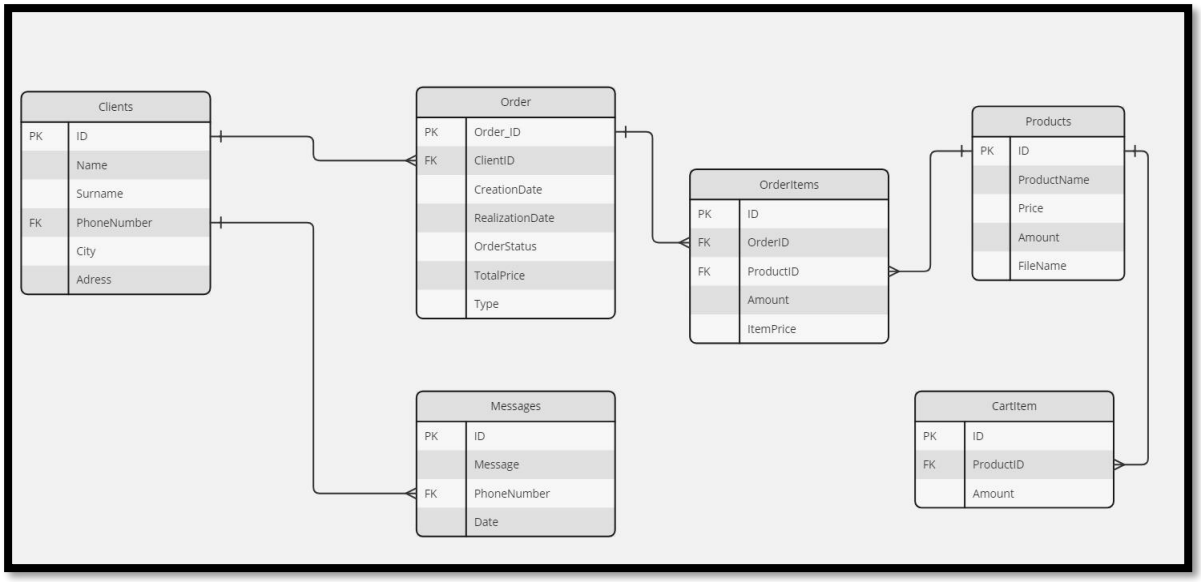
Rysunek 5.4 Fragment pliku AndroidManifest.xml z deklaracją uprawnień.

5.3. Model danych

Model danych w omawianym w tej pracy systemie został zaprojektowany, aby przechowywać wszystkie istotne informacje. Aby system realizował przeznaczone funkcje wyróżniono sześć koniecznych tabel oraz utworzono je wykorzystując przy tym **Room Persistance Library**. Wyróżnione następujące tabele:

- *Clients* (Klienci) – przechowuje informacje o klientach potrzebne do ich weryfikacji, imię i nazwisko, numer telefonu oraz adres.
- *Messages* (Wiadomości) – zawiera treść, numer telefonu nadawcy oraz datę przechwycenia wiadomości
- *Orders* (Zamówienia) – znajdują się w niej informacje o numerze identyfikacyjnym klienta, dacie utworzenia i realizacji, typie, statusie oraz całkowitej wartości zamówienia
- *Products* (Produkty) – przechowuje dane o nazwie i cenie produktu, dostępnym stanie magazynowym oraz nazwie pliku obrazu z wizualizacją produktu.
- *OrderItems* (Produkty w zamówieniu) – zawiera informacje o numerach identyfikacyjnych zamówień oraz produktów, liczbę oraz cenę jaką miały produkty w momencie tworzenia zamówienia.
- *CartItems*(Produkty w koszyku) – przechowuje informacje o produktach, które dodano do koszyka, o ich numerze identyfikacyjnym oraz liczbie.

Do każdej z tabel dodano również atrybut pozwalający przechować informacje o numerach identyfikacyjnych rekordów. Pełnią one role kluczy głównych tabel, aby zapewnić ich unikalność wartości są numerowane automatycznie. Rysunek 5.5 przedstawia kompletny schemat relacyjnej bazy danych zawierających nazwy poszczególnych tabel, atrybutów w nich występujących, zarówno klucze główne (oznaczone jako **PK** ang. **Primary Key**) i obce (oznaczone jako **FK** ang. **Foreign Key**) oraz relacje „**Jeden do Wielu”** które wykorzystano do utworzenia połączeń między poszczególnymi tabelami.



Rysunek 5.5 Schemat relacyjnej bazy danych.

Biblioteka Room dostarcza komponenty oznaczone odpowiednimi adnotacjami potrzebne do utworzenia lokalnej bazy danych w pamięci urządzenia mobilnego. Trzy główne z nich to:

- Encje danych (ang. Data entities) reprezentujące tabele [11].
- Komponent dostępu do danych (DAO ang. Data Access Object) udostępniające metody do odczytywania oraz edytowania danych w tabelach [11].
- Klasa bazy danych, która przechowuje instancje bazy danych oraz stanowi główny punkt dostępu do danych w tabelach [11].

Do stworzenia reprezentacji tabel wykorzystano specjalnie oznaczone klasy dostępne w języku Java. Korzystając z adnotacji *@Entity* oraz parametru *tableName* utworzono klasy, które pełnią jednoczesną rolę tabel w bazie danych. Każde pole w klasach modelu danych oznaczone zostało adnotacją *@ColumnInfo* z parametrem *name* ustawiając tym samym nazwę atrybutu w tabeli. Dla pól przechowujących informację o numerze identyfikacyjnym zastosowano specjalną adnotację *@PrimaryKey* określającą go jako klucz główny tabeli a dodatkowo w celu możliwości automatycznej numeracji rekordów w adnotacji wykorzystano parametr *autoGenerate*. Opisane działania zastosowano dla każdej klasy reprezentującej dane w systemie. Rysunek 5.6 przedstawia jedną z klas danych występujących w systemie. Jest to klasa *Client* reprezentująca jednego klienta, a jednocześnie schemat utworzenia tabeli *Clients* w bazie danych.

```
@Entity(tableName = "Clients")
public class Client {

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "ID") private int ID;
    @ColumnInfo(name = "Name")private String name;
    @ColumnInfo(name = "Surname")private String surname;
    @ColumnInfo(name = "Phonenumber")private String phoneNumber;
    @ColumnInfo(name = "City")private String city;
    @ColumnInfo(name = "Address")private String address;
//
... metody dostępu i ustawiania pól klasy (gettery i settery)
//
}
```

Rysunek 5.6 Klasa Client reprezentują klienta sklepu.

Komponenty DAO są interfejsami umożliwiającymi wykonywanie operacji na rekordach w tabelach bazy danych. Dla każdej klasy danych, utworzono odpowiadający jej obiekt dostępu do danych. Wykorzystano dostępny w języku programowania Java interface wraz z adnotacją *@Dao*. W DAO możliwe jest tworzenie dodawanie rekordów do tabeli *(@Insert*), usuwanie *(@Delete*), aktualizowanie *(@Update*) oraz wysyłanie specjalistycznych zapytań *(@Query*) konstruowanych przy wykorzystaniu strukturalnego języka zapytań SQL. Rysunek 5.7 przedstawia komponent dostępu do danych *ClientDao* odpowiadający klasie *Client* oraz umożliwiający operacje na danych w tabeli *Clients*. Zadeklarowano w nim metody pozwalające na dodawanie oraz aktualizacje danych, a także zapytania pozwalające na pobranie wszystkich bądź wybranych rekordów z tabeli.

```
@Dao
public interface ClientDao {
    @Query("SELECT * FROM Clients")
    List<Client> getAllClients();

    @Insert
    void insertAll(Client ... clients);

    @Query("SELECT * FROM Clients WHERE Phonenumber == :number")
    Client getClientByPhoneNumber(String number);

    @Query("SELECT EXISTS(SELECT * FROM Clients
            WHERE Phonenumber ==:number)")
    boolean getClientByNumber(String number);

    @Query("SELECT * FROM Clients WHERE ID == :clientID")
    Client getClientByID(int clientID);

    @Update
    void updateClient(Client client);
}
```

Rysunek 5.7 Komponent dostępu do danych.

Komponent bazy danych jest klasą abstrakcyjną stanowiącą konfiguracje oraz główny punkt dostępu do danych przechowywanych w pamięci urządzenia mobilnego. Klasę bazy danych zdefiniowano korzystając z adnotacji *@Database*. Jednym z jej parametrów jest *entities* określający listę wszystkich klas danych wchodzących w skład bazy danych. Dodatkowo wykorzystano adnotację *@TypeConverters* pozwalającą na wykorzystanie konwerterów typów danych na takie które możliwe są do przechowywania w bazie danych biblioteki **Room**. W ciele klasy bazy danych zdefiniowano również abstrakcyjne, bezargumentowe metody zwracające instancje DAO dające możliwości dostępu do wykonywania operacji na tabelach bazy.

Istotną kwestią jest zapewnienie, aby w systemie istniała tylko jedna instancja klasy bazy danych. Aby to zapewnić wykorzystano kreacyjny wzorzec projektowy **Singleton**. Dodatkowo pozwala on na globalny dostęp do instancji tego obiektu dzięki czemu możliwe jest odwołanie się do bazy danych z różnych komponentów systemu android (aktywności lub fragmentów) co skutkuje w bardziej optymalnym zarządzaniu zasobami oraz redukuje konflikt w dostępie do danych.

Rysunek 5.8 przedstawia klasę bazy danych w omawianym systemie. Klasa *Database* zawiera metody pozwalające uzyskać dostęp do wszystkich tabel oraz instancję samej bazy danych *INSTANCE*. Dostęp do bazy zapewnia metoda *getInstance()* **(Rysunek 2 dodatek Źródła)** zwracająca instancję bazy lub tworzy nową w przypadku gdy takowa jeszcze nie istnieje.

```
1  @Database(entities = {Message.class, Product.class,
2  Client.class,Order.class,OrderItem.class,CartItem.class},
3  version = 1,exportSchema = false)
4  @TypeConverters(Converters.class)
5  public abstract class Database extends RoomDatabase {
6
7      public abstract MessageDao messageDao();
8      public abstract ProductDao productDao();
9      public abstract ClientDao clientDao();
10     public abstract OrderDao orderDao();
11     public abstract OrderItemDao orderItemDao();
12     public abstract CartItemDao cartItemDao();
13
14     public static Database INSTANCE;
15
16     public static Database getInstance(Context context);
17 }
```

Rysunek 5.8 Klasa bazy danych.

5.4. Obsługa wiadomości SMS

Wiadomości tekstowe są kluczowym aspektem działania systemu do obsługi sprzedaży i dostaw produktów i przetworów rolnych. To na ich podstawie tworzone są nowe zlecenia. Korzystając z możliwości systemu android wykorzystano mechanizm pozwalający aplikacji mobilnej na przechwytywanie przychodzących wiadomości. Za to zadanie odpowiedzialna jest w systemie klasa *SMSReceiver* rozszerzająca klasę nadrzędną *BroadcastReceiver* pozwalającą reagować na wszelakie zdarzenia bądź powiadomienia systemowe. W klasie tej nadpisana (ang. override) metoda *onReceive()* wywoływana jest w momencie gdy nowa wiadomość zostanie zarejestrowana w systemie (Rysunek 5.9). W jej ciele sprawdzane jest przy pomocy metody *checkMessage()* (Rysunek 5.10) czy treść przechwyconej wiadomości zawiera sygnatury *ZAMÓWIENIE* lub *DOSTAWA* zdefiniowane typem wyliczeniowym *OrderType* (Rysunek 5.11). Za dodanie wiadomości do bazy danych odpowiedzialna jest metoda *createMessage()* (Rysunek 5.12). W niej tworzony jest nowy obiekt wiadomości *Message* a następnie dodawany jest nowy rekord do tabeli *Messages* w bazie danych. Ostatecznie *Słuchacz* (ang. Listener) zawiadamia listę o przyjęciu nowego elementu w celu jej aktualizacji a na ekran użytkownika wysyłane jest powiadomienie za pomocą klasy *NotificationSender* omówionej w rozdziale **5.5 Powiadomienia**. W klasie *SMSReceiver* zaimplementowano również *Słuchacza SMSReceiverListener* (Rysunek 5.12*)* służącego do powiadamiania fragmentu przechowującego komponentu listę wiadomości po to, aby w przypadku przechwycenia nowej wiadomości, interfejs użytkownika został zaktualizowany o dodatkową wiadomość.

```
1  @Override
2  public void onReceive(Context context, Intent intent) {
3      Bundle data = intent.getExtras();
4      Object[] smsObj = (Object[]) data.get("pdus");
5
6      for(Object obj : smsObj){
7          SmsMessage smsMessage = SmsMessage.createFromPdu((byte[])
8  obj);
9          if(checkMessage(smsMessage.getDisplayMessageBody())){
10             createMessage(context,smsMessage);
11         }
12     }
13 }
```

Rysunek 5.9 Kod źródłowy metody onReceive z klasy SMSReceiver.

```
1  private boolean checkMessage(String message){
2      return message.contains(OrderType.ORDER.getDescription()) ||
3             message.contains(OrderType.DELIVERY.getDescription());
4  }
```

Rysunek 5.10 Kod źródłowy metody checkMessage z klasy SMSReceiver.

```
public enum OrderType {
    ORDER(”ZAMÓWIENIE”),DELIVERY (”DOSTAWA”);
...
...pole ‘description’ oraz metody dostępu
...
}
```

Rysunek 5.11 Typ wyliczeniowy OrderType.

```
private void createMessage(Context context, SmsMessage smsMessage){
    String number = smsMessage.getDisplayOriginatingAddress();
    String message = smsMessage.getDisplayMessageBody();
    LocalDateTime time = LocalDateTime.now();

    Message messageObject = new Message(message,number,time);
    Database.getInstance(context).messageDao()
    .insertAll(messageObject);
    Toast.makeText(context,”Przechwycono nowe zlecenie!”,
    Toast.LENGTH_LONG).show();
    smsReceiverListener.onSMSReceived();
    NotificationSender.makeNotification(context);
}
```

Rysunek 5.12 Metoda CreateMessage z klasy SMSReceiver.

Aby system spełniał swoją funkcję jego praca musi być kontynuowana również w przypadku, gdy użytkownik bezpośrednio z niego nie korzysta. W tym celu wykorzystano możliwość działania aplikacji mobilnej w tle i klasę *SMSService* dziedziczącą po klasie nadrzędnej *Service* umożliwiającą realizacje opisywanego zadania (Rysunek 5.13). W metodzie *onStartCommand()* klasy *SMSService* rejestrowany jest obiekt *SMSReceiver* służący do przechwytywania wiadomości tekstowych a zwracana wartość *START_STICKY* zapewnia uruchomienie usługi w przypadku gdy zostanie zatrzymana co zapewnia długoterminowe działanie nawet gdy użytkownik zamknie system.

```
public class SMSService extends Service{

    private SMSReceiver receiver = new SMSReceiver();
    @Override
    public int onStartCommand(Intent intent, int flags, int startId)
    {
        registerReceiver(receiver,
     new IntentFilter(”android.provider.Telephony.SMS_RECEIVED”));
        return START_STICKY;
    }
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

Rysunek 5.13 Klasa SMSService umożliwiająca działanie systemu w tle.

Oprócz przyjmowania wiadomości system również je wysyła. W tym celu wykorzystano klasę *SMSSender* (Rysunek 5.13) która zawiera tylko jedną metodę *sendMessage()*. Metoda ta przyjmuje dwa parametry: numer telefonu, na który ma zostać wysłana wiadomość, oraz jej treść. Możliwość wysyłania wiadomości zapewnia instancja klasy *SmsManager*. Następnie jej treść dzielona jest na maksymalnie możliwej długości fragmenty mogące być wysłane poprzez wiadomość SMS oraz wysyłana jest przy pomocy metody *sendMultipartTextMessage()* z klasy *SmsManager*.

```
1  public class SMSSender {
2
3    public static void sendMessage(String phoneNumber,String message){
4        SmsManager manager = SmsManager.getDefault();
5
6        ArrayList<String> messageParts =
7        manager.divideMessage(message);
8        manager.sendMultipartTextMessage(phoneNumber,null,
9                messageParts, null, null);
10   }
11 }
```

Rysunek 5.14 Klasa SMSSender dająca możliwość wysyłania wiadomości tekstowych.

5.5. Powiadomienia

Powiadomienia wyświetlane są na ekranie urządzenia mobilnego użytkownika w sytuacji, gdy zostanie przechwycona nowa wiadomość SMS. Powodem takiego rozwiązania jest chęć poinformowania o tym użytkownika bez konieczności wchodzenia do aplikacji. Za powiadomienia w systemie odpowiada klasa *NotificationSender*. Funkcja *makeNotification()* (Rysunek 5.15) używając *Budowniczego* (ang. *Builder*) przygotowuje i konfiguruje powiadomienie ustawiając jej tytuł, wiadomość i priorytet.

```
1  private static final String CHANNEL_ID = "CHANNEL_ID_NOTIFICATION";
2
3  public static void makeNotification(Context context){
4      NotificationCompat.Builder builder  = new NotificationCompat
5              .Builder(context.getApplicationContext(), CHANNEL_ID)
6              .setSmallIcon(R.drawable.notification_icon)
7              .setContentTitle("Nowe zamówienie!")
8              .setContentText("Przechwycono nowe zamówienie")
9              .setStyle(new NotificationCompat.BigTextStyle())
10             .setPriority(NotificationCompat.PRIORITY_DEFAULT);
11
12     createNotificationChannel(context);
13     NotificationManager notificationManager = context
14     .getSystemService(NotificationManager.class);
15     notificationManager.notify(0,builder.build());
16 }
```

Rysunek 5.15 Funkcja makeNotification klasy NotificationSender.

Następnie prywatna metoda *createNorificationChannel()* (Rysunek 5.16) korzystając z menadżera powiadomień (ang. NotificationManager) tworzy kanał o identyfikatorze *CHANNEL_ID* umożliwiający ich wysłanie jednocześnie go konfigurując. Ostatecznie wykorzystując metodę *notify()* instancji klasy *NotificationManager* powiadomienie wysyłane jest na ekran urządzenia mobilnego.

```
1  private static void createNotificationChannel(Context context) {
2
3      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
4          NotificationManager notificationManager = context
5          .getSystemService(NotificationManager.class);
6          NotificationChannel notificationChannel =notificationManager
7          .getNotificationChannel(CHANNEL_ID);
8
9          if(notificationChannel == null) {
10             int importance = NotificationManager.IMPORTANCE_DEFAULT;
11             notificationChannel = new NotificationChannel(CHANNEL_ID
12             ,"ChannelID", importance);
13             notificationChannel.setLightColor(Color.GREEN);
14
15             notificationChannel.enableVibration(true);
16             notificationManager
17             .createNotificationChannel(notificationChannel);
18         }
19     }
20 }
```

Rysunek 5.16 Metoda createNotificationChannel klasy NotificationSender.

5.6. Lista elementów - RecyclerView

W wielu oknach otwieranych w systemie wykorzystywana jest lista przechowująca elementy z tabel bazy danych. Są to np. lista klientów w oknie klienci (Rysunek 4.8) lub lista produktów na magazynie w oknie magazynu (Rysunek 4.4). W tym celu w plikach xml opisujących widok okien skorzystano z komponentu *RecyclerView* (Rysunek 5.17) służącego do wyświetlania dużej ilości danych w formie listy.

```
1  <androidx.recyclerview.widget.RecyclerView
2      android:id="@+id/clientsRecyclerAdapter"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent" />
```

Rysunek 5.17 Komponent RecyclerView zdefiniowany w pliku fragment_clients.xml.

Za dostarczenie i prezentacje danych na liście *RecyclerView* odpowiadają klasy Adapterów. Występujące w systemie adaptery tworzone są indywidualnie dla każdego odpowiadającego mu widoku. Jego budowa zostanie omówiona na podstawie listy klientów i klasy *ClientsListRecyclerAdapter*. Elementami, z których składają się adaptery są:

- Konstruktor przyjmujący obligatoryjnie listę elementów do wyświetlenia. Dodatkowym parametrem jest obiekt *Słuchacza RecyclerViewClickListener* implementujący metodę *onClick()* pozwalającą na wykonanie działań w przypadku interakcji użytkownika z elementem listy(Rysunek 5.18)

```
1  private List<Client> clients;
2  private RecyclerViewClickListener listener;
3
4  public ClientsListRecyclerAdapter(List<Client> clients,
5  RecyclerViewClickListener listener) {
6      this.clients = clients;
7      this.listener = listener;
8  }
```

Rysunek 5.18 Pola i konstruktor klasy ClientListRecyclerAdapter

- Wewnętrznej klasy *ViewHolder* przechowującej referencje do elementów widoku, które w konstruktorze klasy są inicjalizowane za pomocą metody *findViewById()* oraz identyfikatora zasobów (Rysunek 5.19).

```
1  public class ViewHolder extends RecyclerView.ViewHolder implements
2  View.OnClickListener {
3
4      private TextView position, fullname, phoneNumber, city, address;
5
6      public ViewHolder(final View view) {
7          super(view);
8          position = view.findViewById(R.id.position);
9          fullname = view.findViewById(R.id.clientName);
10         phoneNumber = view.findViewById(R.id.phoneNumber);
11         city = view.findViewById(R.id.city);
12         address = view.findViewById(R.id.address);
13         itemView.setOnClickListener(this);
14     }
15
16     @Override
17     public void onClick(View v) {
18         listener.onClick(v, getAdapterPosition());
19     }
20 }
```

Rysunek 5.19 Wewnętrzna klasa ViewHolder.

- Metody *onCreateViewHolder()* odpowiedzialnej za utworzenie nowego obiektu *ViewHolder*, który przechowuje jeden element w liście. Obiekt tworzony jest na podstawie pliku widoku opisującego jeden elementy listy w momencie otwarcie okna z komponentem *ReycyclerView* oraz w momencie przewijania listy (Rysunek 5.20).

```
1  @NonNull
2  @Override
3  public ClientsListRecyclerAdapter.ViewHolder
4  onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
5      View itemView = LayoutInflater
6      .from(parent.getContext())
7      .inflate(R.layout.clients_list_item, parent, false);
8      return new ClientsListRecyclerAdapter.ViewHolder(itemView);
9  }
```

Rysunek 5.20 Metoda onCreateViewHolder

- Metody *onBindViewHolder()* wywoływanej w momencie wyświetlania kolejnego elementu na określonej pozycji na liście. W przypadku adaptera listy klientów, dane obiektu *client* ustawiane są na określonych pozycjach w obiekcie klasy *ViewHolder* (Rysunek 5.21).
- Metody *getItemCount()* zwracające całkowity rozmiar listy (Rysunek 5.21).

```
1  @Override
2  public void onBindViewHolder(@NonNull
3  ClientsListRecyclerAdapter.ViewHolder holder, int position) {
4      Client client = clients.get(position);
5      holder.position.setText(Integer.toString(position+1));
6      holder.fullname.setText(client.getName() + " " +
7  client.getSurname());
8      holder.phoneNumber.setText(client.getPhoneNumber());
9      holder.city.setText(client.getCity());
10     holder.address.setText(client.getAddress());
11 }
12
13 @Override
14 public int getItemCount() {return clients.size();}
```

Rysunek 5.21 Metoda onBindViewHolder oraz getItemCount.

5.7. Realizacja zleceń

Proces zakańczania zleceń może być zrealizowany w systemie z dwóch miejsc. Pierwszym jest okno z listą wszystkich utworzonych zleceń, gdzie użytkownik ma możliwość jego realizacji, natomiast druga ewentualnością jest podsumowanie zlecenia po interakcji użytkownika z konkretnym zamówieniem. W oknie dialogowym podsumowania istnieje możliwość zarówno realizacji jak i anulowania zleceń.

Przy interakcji użytkownika z przyciskiem „Realizuj” wywoływana jest metoda *fulfillOrder()* (Rysunek 5.22), która w pierwszej kolejności wyświetla okno dialogowe z zapytaniem *„Czy na pewno chcesz zrealizować zlecenie?”* oraz wymaganiem potwierdzenia realizacji, a po pozytywnej odpowiedzi następuje wywołanie metod pomocniczych aktualizujących dane w bazie oraz w widokach systemu.

```
1  private void fulfillOrder(View view,int index,
2  List<Product>products, AlertDialog dialog){
3
4      AlertDialog.Builder confirmationBuilder = new AlertDialog
5      .Builder(view.getContext());
6      confirmationBuilder
7      .setMessage(”Czy na pewno chcesz zrealizować zlecenie?”);
8      confirmationBuilder.setCancelable(true);
9
10     confirmationBuilder.setPositiveButton(
11             ”Tak”,
12             (dialog_, id) -> {
13                 changeOrderStatus(view,index,OrderStatus.COMPLETED);
14                 finishOrder(view,index,products);
15                 Toast.makeText(view.getContext()
16                 ,”Zrealizowano zlecenie!”,Toast.LENGTH_LONG).show();
17                 dialog.dismiss();
18             }
19             );
20
21     confirmationBuilder.setNegativeButton(
22             ”Nie”,
23             (dialog_, id) -> dialog_.cancel());
24     AlertDialog confirm = confirmationBuilder.create();
25     confirm.show();
26 }
```

Rysunek 5.22 Metoda fulfillOrder służąca do realizacji zleceń.

Wywoływana w linii nr. 14 (Rysunek 5.22) metoda *changeOrderStatus()* (Rysunek 5.23) odpowiedzialna jest za zmianę statusu zamówienia w bazie danych. W pierwszej kolejności z instancji bazy danych pobieranych jest obiekt *OrderDao* służący do dostępu do danych z tabeli *Orders*. Następnie poprzez metodę *updateOrderStatus* z interfejsu *OrderDao* aktualizowane jest status zlecenia z pozycji *position* na liście. W zależności od przekazanego parametru, *OrderStatus* ustawiany jest na:

- *OrderStatus.COMPLETED* – w przypadku realizacji zlecenia
- *OrderStatus.CANCELED* – w przypadku jego anulowania.

Ostatnim krokiem jest ustawienie daty zakończenia zlecenia. Jest to realizowane przez metodę *updateRealizationDate()* interfejsu *OrderDao* oraz klasę *LocalDateTime*.

```
1  private void changeOrderStatus(View view,
2                                 int position,OrderStatus status){
3      OrderDao orderDao = Database
4      .getInstance(view.getContext())
5      .orderDao();
6      orderDao.updateOrderStatus(orders.get(position)
7                                 .order.getOrderID(), status);
8      orderDao.updateRealizationDate(orders.get(position)
9                                 .order.getOrderID(),
10                                LocalDateTime.now());
11 }
```

Rysunek 5.23 Metoda changeOrderStatus

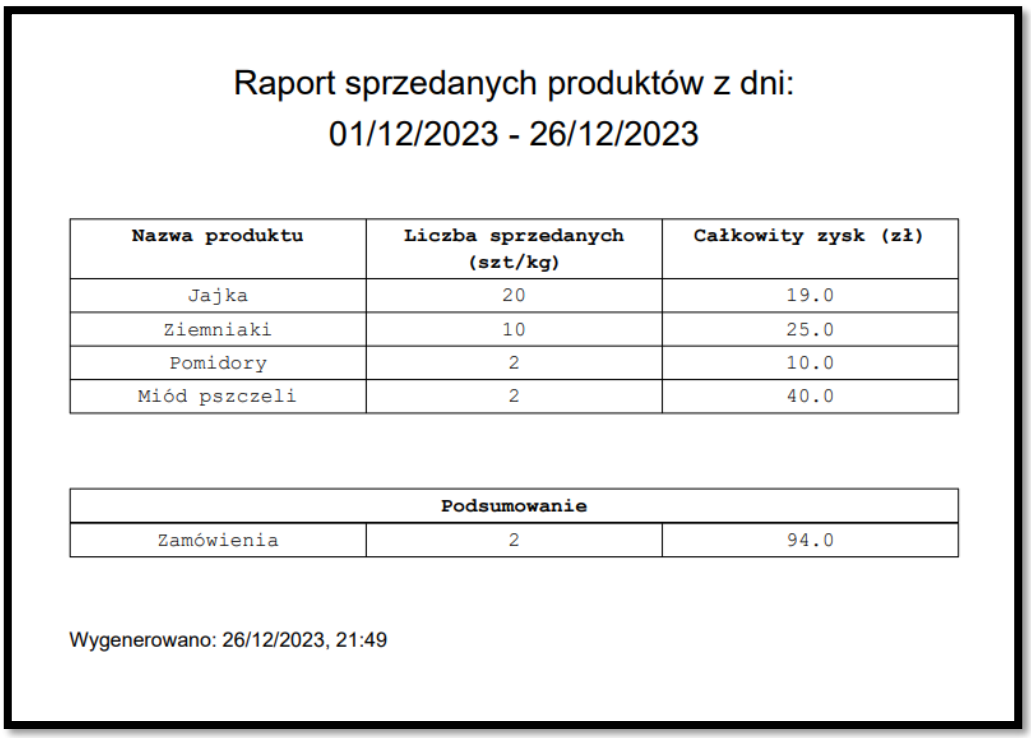
Metoda *finishOrder()* (Rysunek 5.25) z linii nr. 15 (Rysunek 5.22) finalizuje zlecenie. W jej ciele w przypadku odbioru produktów aktualizowany jest również magazyn. Funkcja *updateWarehouse()* inkrementuje produkty na magazynie o liczbę zawartą w finalizowanym zleceniu. Obiekt zamówienia jest także usuwany z listy a widok aktualizowany poprzez metodę *notifyItemRemoved()*.

```
1  private void finishOrder( View view, int position,
2                            List<Product> products){
3      if(orders.get(position).order.getType() == OrderType.DELIVERY){
4          updateWarehouse(view,products);
5      }
6      orders.remove(position);
7      notifyItemRemoved(position);
8  }
9
10 private void updateWarehouse(View view,
11                              List<Product> productsInCart){
12     Database database = Database.getInstance(view.getContext());
13     for(Product product : productsInCart){
14         database.productDao()
15                 .addToWarehouse(product.getID(),product
16                 .getAmount());
17     }
18 }
```

Rysunek 5.24 Metoda finishOrder

5.8. Generowanie raportu

Możliwość generowania raportów w aplikacjach daje użytkownikom możliwość analizy statystyk i wyników wykonywanych transakcji bądź zleceń. W systemie do obsługi sprzedaży i dostaw produktów i przetworów rolnych zaimplementowano funkcjonalność pozwalającą użytkownikowi systemu na generowanie raportu z wynikami sprzedaży w określonym przedziale czasowym. Generowany do dokumentu tekstowego raport zawiera tabele z listą sprzedanych produktów, ich liczbą oraz sumaryczną ceną za dany produkt. Zawiera on również podsumowanie wszystkich zleceń. Raport pokazuje liczbę zrealizowanych zleceń oraz całkowity zysk (Rysunek 5.25).



Raport sprzedanych produktów z dni:
01/12/2023 - 26/12/2023

Nazwa produktu	Liczba sprzedanych (szt/kg)	Całkowity zysk (zł)
Jajka	20	19.0
Ziemniaki	10	25.0
Pomidory	2	10.0
Miód pszczeli	2	40.0

Podsumowanie		
Zamówienia	2	94.0

Wygenerowano: 26/12/2023, 21:49

Rysunek 5.25 Raport sprzedażowy w dokumencie tekstowym

Generowanie raportów możliwe jest w klasie fragmentu *GenerateRaportFragment* wraz z widokiem opisanym w pliku xml. Po naciśnięciu przycisku „Generuj raport" wywoływana jest funkcja obsługująca generowanie dokumentu. W pierwszej kolejności z bazy danych przy pomocy metody *getOrdersBetween()* z intefejsu *OrderDao* pobierane są wszystkie zlecenia znajdujące się w przedziale czasowym określonym przez użytkownika. Dodatkowo podawane są dwa parametry określające typ zlecenia jako zamówienie oraz status jako zrealizowany (Rysunek 5.26).

```
@Transaction
@Query("SELECT * FROM Orders
WHERE RealizationDate BETWEEN :startDate AND :endDate
AND OrderStatus = :status AND Type = :type")
List<OrderWithOrderItemsAndProducts> getOrdersBetween
(LocalDateTime startDate, LocalDateTime endDate,
OrderStatus status, OrderType type);
```

Rysunek 5.26 Metoda getOrderBetween z interfejsu OrderDao.

Kolejnym etapem jest przygotowanie danych do wygenerowania raportu. Po wczytaniu ich z bazy danych, lista obiektów przekazywana jest jako argument do funkcji *prepareReportData()* (Rysunek 5.27). W ciele funkcji do przechowania przygotowanych danych o produktach wykorzystano strukturę danych *LinkedHashMap*. Zaletą tej struktury jest zachowana kolejność dodawanych elementów co jest znacząco przydatne przy chęci iteracji po nich. Wykorzystana struktura *productSummary* parametryzowana jest dwoma wartościami:

- *String* – klucz główny mapy przechowujący unikalny tekstowy typ danych, w przypadku omawianego systemu jest to nazwa produktu.
- *Map.entry<Integer, double>* - wartość klucza głównego mapy, gdzie:
  - *Integer* – wartość całkowita określająca sumę poszczególnego produktu w zrealizowanych zleceniach.
  - *Double* – wartość zmiennoprzecinkowa określająca zsumowaną cenę danego produktu.

We wnętrzu funkcji dwie zagnieżdżone pętlę for przechodzą po wszystkich zamówieniach w liście a następnie po produktach w poszczególnych zleceniach. W wewnętrznej pętli wykonywane są instrukcje warunkowe na podstawie których podejmowane są następne decyzję w algorytmie.

Jeśli produkt o danym kluczu (nazwie produktu) istnieje już w mapie *productSummary.containsKey(productName)* jego cena oraz liczba jest aktualizowana. W przeciwnym wypadku do mapy dodawany jest nowy wpis.

Pętla zewnętrzna to natomiast zliczanie całkowitej kwoty zrealizowanych zamówień. Z każdego zlecenia wyciągana jest jego wartość końcowa oraz dodawana do zmiennej *ordersTotalPrice* określająca kwotę całkowitą wszystkich zamówień. Ostatecznym krokiem algorytmu jest dodanie ostatniego wpisu do mapy określającego zamówienia (klucz „Zamówienia”), ich liczby oraz sumaryczną kwotę.

```
1  private Map<String, Map.Entry<Integer, Double>>
2  prepareReportData(List<OrderWithOrderItemsAndProducts> orders) {
3  Map<String, Map.Entry<Integer, Double>> productSummary =
4                                      new LinkedHashMap<>();
5  double ordersTotalPrice = 0.0;
6
7  for (OrderWithOrderItemsAndProducts order : orders) {
8      for (OrderItemAndProduct orderItemAndProduct : order.orderItems)
9  {
10         String productName = orderItemAndProduct.product
11                                .getProductName();
12         int amount = orderItemAndProduct.orderItem
13                        .getAmount();
14         double priceForProducts = orderItemAndProduct.orderItem
15                                    .getItemPrice() * amount;
16
17             if (productSummary.containsKey(productName)) {
18                 // Jeśli produkt już istnieje w mapie, aktualizuj
19                    ilość i cenę
20                 Map.Entry<Integer, Double> existingEntry =
21                     productSummary.get(productName);
22                 int totalAmount = existingEntry.getKey();
23                 double totalPrice = existingEntry.getValue();
24
25                 existingEntry = Map.entry(totalAmount + amount,
26                                    totalPrice + priceForProducts);
27                 productSummary.put(productName, existingEntry);
28             } else {
29             // Jeśli produkt nie istnieje w mapie, dodaj nowy wpis
30                 Map.Entry<Integer, Double> newEntry =
31                 Map.entry(amount, priceForProducts);
32                 productSummary.put(productName, newEntry);
33             }
34         }
35         ordersTotalPrice += order.order.getTotalPrice();
36     }
37     productSummary.put("Zamówienia",
38     Map.entry(orders.size(), ordersTotalPrice));
39     return productSummary;
40 }
```

Rysunek 5.27 Metoda prepareReportData przygotowująca dane do raportu.

Do generacji dokumentu tekstowego wykorzystywana jest klasa *PDFService*. Zawiera ona statyczną metodę publiczną *createPdfReport()*. Jest to metoda sterująca obsługująca proces generowania reportu i dyktująca kolejne kroki w zależności od spełnionych warunków. Dodatkowo w klasie znajdują się prywatne metody pomocnicze takie jak między innymi *prepareFile()* czy *prepareFileName()* przygotowujące proces raportowania. Klasa *PDFService* korzysta z biblioteki **iTextPDF,** która jest otwartym oprogramowaniem służącym do generowania i edytowania dokumentów w formacie PDF.

Główną funkcją tworzącą i zapisującą do dokumentu tekstowego jest *createAndWriteDocument()*. Korzystając z możliwości biblioteki iTextPDF, za pomocą klasy *Document* tworzona jest instancja dokumentu tekstowego oraz klasy *PdfFont* - czcionki opisujące wygląd tekstu w pliku (Rysunek 5.28). Zostały wykorzystane dwie

czcionki, **Courier Bold** w nagłówkach tabel (*PdfFont header*) oraz **Courier** w opisie zawartości tabeli i nagłówkach (*PdfFont font*).

```
1  PdfDocument pdfDocument = new PdfDocument(new PdfWriter(file));
2  Document document = new Document(pdfDocument);
3  PdfFont header =
4  PdfFontFactory.createFont(StandardFonts.COURIER_BOLD,
5  PdfEncodings.CP1250);
6  PdfFont font = PdfFontFactory.createFont(StandardFonts.COURIER,
7  PdfEncodings.CP1250);
```

Rysunek 5.28 Utworzenie dokumentu tekstowego i czcionek w nim wykorzystywanych.

W celu dodania do dokumentu tabeli wykorzystano klasę *Table* dostępną w używanej bibliotece iTextPDF. W konstruktorze obiektu wielkość tabeli ustawiana jest na trzy kolumny a powierzchnia rozszerzana jest na całą dostępną szerokość pliku możliwą do edycji (linia nr 1 i 2 Rysunek 5.29). Korzystając z metody *addCell()* oraz metody pomocniczej klasy *PDFService createCell()* (Rysunek 5.30) tworzone są komórki nagłówkowe tabeli o odpowiedniej czcionce. Następnie tabela wypełniana jest strukturą *productSummary a* w kolejnych wierszach znajdują się odpowiednio nazwa, liczbą sprzedanych sztuk lub kilogramów produktów oraz ich całkowita cena. Finalnie obiekt tabeli za pomocą metody *add()* dodawany jest do struktury dokumentu.

```
1  Table table = new Table(UnitValue.createPercentArray(3))
2                          .useAllAvailableWidth();
3
4  table.addCell(createCell("Nazwa produktu", header));
5  table.addCell(createCell("Liczba sprzedanych (szt/kg) ", header));
6  table.addCell(createCell("Całkowity zysk (zł) ", header));
7
8  for (Map.Entry<String, Map.Entry<Integer, Double>> entry :
9  productSummary.entrySet()) {
10
11     table.addCell(createCell(entry.getKey(), font));
12     table.addCell(createCell(entry.getValue()
13                                    .getKey().toString(), font));
14     table.addCell(createCell(entry.getValue()
15                                    .getValue().toString(), font));
16 }
17 document.add(table);
```

Rysunek 5.29 Fragment kodu służący do stworzenia tabeli z produktami w dokumencie tekstowym.

```
1  private static Cell createCell(String cellHeader, PdfFont font) {
2      Cell cell = new Cell()
3                  .add(new Paragraph(cellHeader))
4                  .setFont(font)
5                  .setFontColor(ColorConstants.BLACK);
6      cell.setTextAlignment(TextAlignment.CENTER);
7      return cell;
8  }
```

Rysunek 5.30 Metoda createCell klasy PDFService.

Dodatkowo do dokumentu tekstowego korzystając z klasy *Paragraph* dodano nagłówek *„Raport sprzedanych produktów z dni:”* wraz z przedziałem dat podlegających raportowi oraz datę wygenerowania na jego końcu. Ostatecznie dokument jest zapisywany i zamykany przy pomocy metody *close()*.

Za wyświetlenie wygenerowanego raportu w systemie odpowiedzialna jest metoda *displayPdf()*. Po sprawdzeniu istnienia pliku tworzona jest nowa intencja (ang. Intent) o akcji *ACTION_VIEW* służąca do otwierania i wyświetlania danych. Następnie wykorzystując *FileProvider* uzyskiwany jest identyfikator **URI** (ang. **Uniform Resource Identifier**) umożliwiający uzyskanie dostępu do zasobu w formacie dokumentu PDF oraz nadawane jest tymczasowe uprawnienie pozwalające na odczyt pliku z dostarczonego URI przez inne aplikacje. Finalnym krokiem jest uruchomienie aktywności poprzez metodę *startActivity()* co skutkuje otwarciem wygenerowanego raportu w zewnętrznej aplikacji służącej do wyświetlania dokumentów tekstowych w formacie PDF.

```
1  private static boolean displayPdf(File file, Context context) {
2      if (file.exists()) {
3          Intent target = new Intent(Intent.ACTION_VIEW);
4          target.setDataAndType(FileProvider.getUriForFile(context,
5          ”com.example.saleshandlingapp.provider”, file),
6          ”application/pdf”);
7          target.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
8          try {
9              context.startActivity(target);
10         } catch (ActivityNotFoundException e) {
11             return false;
12         }
13     } else
14         return false;
15     return true;
16 }
```

Rysunek 5.31 Metoda displayPdf do otwierania i wyświetlania dokumentu tekstowego.

6. Weryfikacja i walidacja

Kluczowymi aspektami procesu tworzenia oprogramowania jest zapewnienie, że system spełnia stawiane przed nim wymagania, działa zgodnie z oczekiwaniami użytkowników oraz jest pozbawiony błędów. Rozdział ten skupia się na omówieniu wykorzystanych platform testowych oraz przedstawieniu sposobu weryfikacji poprawności opracowanych rozwiązań.

6.1. Platformy testowe

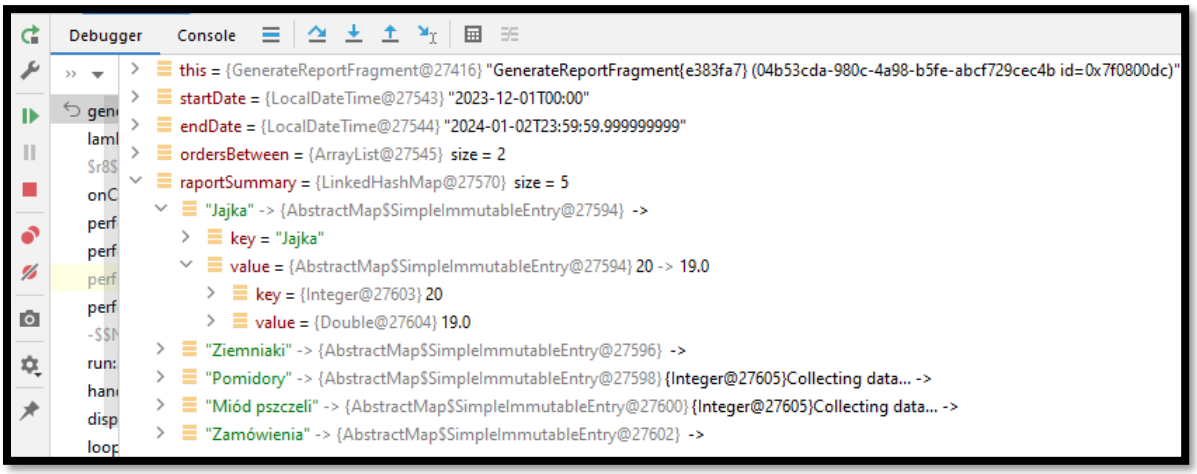
Jedną z platform testowych opracowywanego systemu był emulator w środowisku programistycznym Android Studio. Jest to narzędzie pozwalające na emulację urządzenia mobilnego z systemem operacyjnym Android w środowisku programistycznym. Został wykorzystany emulator urządzenia Google Pixel 4 z systemem Android w wersji 13. Pozwolił on na początkową weryfikację działania systemu oraz ułatwił proces debugowania.

Drugą z platform testowych było urządzenie mobilne Samsung A54 5G z systemem Android w wersji 14. Po początkowych testach na emulatorze w środowisku programistycznym, urządzenie mobilne posłużyło do weryfikacji działania w docelowych warunkach pracy i codziennego korzystania z systemu.

6.2. Testowanie

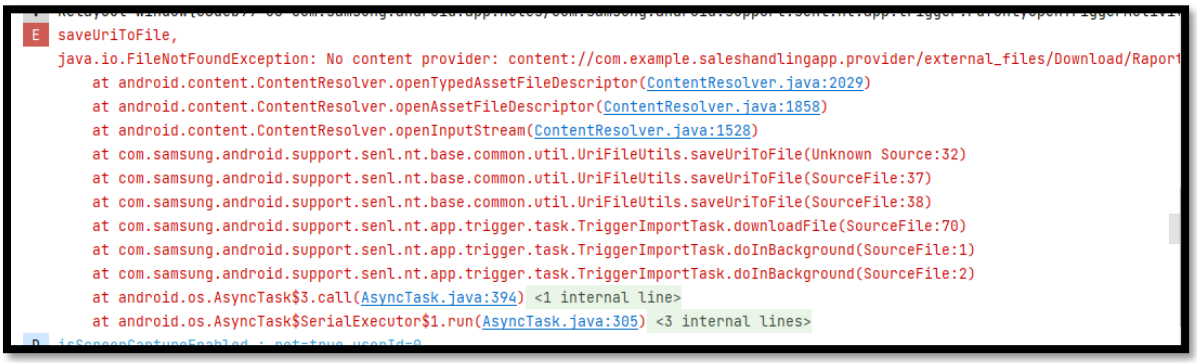
W trakcie testowania systemu szeroko wykorzystywanym narzędziem dostępnym w środowisku programistycznym Android Studio był debugger który pozwala na nadzorowanie wykonywania programu. Zauważone przykłady błędnego działania poszczególnych funkcjonalności były głęboko analizowane a nieprawidłowości analizowane. Wykorzystano do tego również konsole debuggera oferowaną przez środowisko Android Studio. Dzięki niej oraz krokowemu wykonaniu kodu źródłowego

programu możliwa była obserwacja zmiennych występujących w programie w trakcie jego wykonywania oraz analiza ich zmian w kolejnych krokach programu (Rysunek 6.1).



Rysunek 6.1 Konsola debuggera umożliwiająca obserwacje zmiennych w programie.

W procesie weryfikowania działania systemu wykorzystano również narzędzie umożliwiające przeglądanie i analizowanie logów systemowych generowanych przez urządzenie mobilne lub emulator. Narzędziem tym jest **Logcat** (Rysunek 6.2). Wyświetlane w konsoli logi systemowe zawierają informacje o działaniu systemu, zdarzeniach w trakcie jego wykonywania, ostrzeżeniach oraz błędach. Monitorowanie wpisów w narzędziu pozwoliło na identyfikacje błędów i ich przyczyn co ułatwiało proces ich eliminacji oraz przyśpieszyło rozwój systemu.



Rysunek 6.2 Narzędzie Logcat z zarejestrowanym wpisem o błędzie (ang. Error).

7. Podsumowanie

W niniejszej pracy omówiono proces projektowania oraz implementacji systemu do obsługi sprzedaży i dostaw produktów i przetworów rolnych przeznaczonego na urządzania mobilne. W sposób szczegółowy zostały przedstawione wymagania stawiane przed systemem, narzędzia, które zostały wykorzystane w celu ich realizacji a także zostało przeprowadzone wstępne zapoznanie czytelnika z jego użyciem oraz przedstawiono rozwiązania techniczne zastosowane do zrealizowania zamierzonych celów.

7.1. Realizacja założeń

System omawiany w tej pracy skierowany jest do małych przedsiębiorców chcących sprzedawać produkty wytwarzane we własnych gospodarstwach lub też skupować je od innych. Motywacją do zaprojektowania oprogramowania była chęć zautomatyzowania procesów przyjmowania zleceń od klientów oraz tworzenia i realizacji zamówień. Zaprojektowany system przeznaczony na urządzenia mobilne upraszcza pracę przedsiębiorcy pomagając mu w zachowaniu informacji o zamówieniach klientów w jednym miejscu jakim jest smartfon w jego kieszeni. Funkcje wbudowane w urządzenia mobilne takie jak wiadomości SMS skutecznie sprawdzają się jako możliwość składania zleceń a ich przechwytywanie w systemie przyczynia się do ograniczenia czasu potrzebnego do jego przyjęcia. Dodatkowo działanie aplikacji mobilnej w tle sprawia, że wiadomości przychwytywane są mimo braku używania smartfonu i systemu a wyświetlane na ekran powiadomienia na bieżąco informują użytkownika o potencjalnych nowych zleceniach.

Funkcjonalności opracowane w omawianym w tej pracy oprogramowaniu dają możliwość obsługi pełnego procesu transakcji handlowych począwszy od przyjęcia zamówienia do jego realizacji. System przechowując wszystkie istotne dane zarówno o zamówieniach aktualnych jak i tych już zrealizowanych, klientach i produktach dostępnych na magazynie daje możliwość ich obserwacji, analizy a także modyfikacji. Dodatkowo spełnia on główne stawianie przed nim zadanie jakim jest pomoc w zarządzaniu sprzedażą i dostaw produktów i przetworów rolnych.

7.2. Możliwości rozwoju

Mimo tego, iż opracowany system jest w pełni operatywny oraz spełnia stawiane przed nim zadania możliwe są dalsze prace nad jego rozwojem. Funkcjonalności, o które mógłby zostać rozwinięty system to:

- Dodawanie produktów przez użytkownika – w obecnym stanie, aby dodać nowy produkt, którego nie ma w magazynie użytkownik systemu zmuszony jest skontaktować się z programistą. To na nim spoczywa odpowiedzialność modyfikacji kodu źródłowego aplikacji mobilnej oraz dostarczenia nowej wersji oprogramowania do użytkownika. Po dodaniu takiej funkcjonalności użytkownik miałby możliwość dodania produktu o podanej nazwie, wyboru ilustracji mu odpowiadającej, ustaleniu ceny oraz inicjalizacji początkowego stanu magazynowego.
- Automatyczne tworzenie zamówień – system przechwytuje wiadomości SMS a na jej podstawie użytkownik tworzy zlecenia. Możliwość ich tworzenia w sposób automatyczny w dalszy sposób usprawniłoby pracę oraz przyśpieszyło zarówno proces przyjmowania jak i realizacji zleceń. Stosując takie rozwiązanie zadanie użytkownika zostałoby ograniczone do weryfikacji poprawności automatycznie utworzonych zamówień.
- Integracja systemu płatności – usprawnienie systemu poprzez dodanie możliwości płatności elektronicznych w dalszy sposób zautomatyzowałyby proces realizacji zamówień zarówno po stronie klienta jak i użytkownika aplikacji. Takie rozwiązanie zapewniłoby wygodę oraz umożliwiłoby efektywną obsługę przeprowadzania transakcji.

Bibliografia

[1] S. Konkol, „slideshare,” [Online]. Available: https://www.slideshare.net/qwertyra/charakterystyka-informatycznych-systemw-komputerowych. [Data uzyskania dostępu: 13 Listopad 2023].

[2] K. A. Łobejko, „Analiza różnic pomiędzy szkieletami aplikacji natywnych i wieloplatformowych,” *Journal of Computer Sciences Institute*, pp. 119-124, 2019 vol.11.

[3] A. Zoła, Programowanie w języku Java, Helion, 2004.

[4] M. Shavirov, „The Kotlin Blog,” [Online]. Available: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/. [Data uzyskania dostępu: 29 Listopad 2023].

[5] G. K. Daniel Sulowski, „Analiza porównawcza języków Kotlin i Java używanych do tworzenia aplikacji na system Android,” *JCSI - Journal of Computer Sciences Institute,* pp. 354 - 358, https://doi.org/10.35784/jcsi.1332, 2019 vol.13.

[6] „Oficjalna Dokumentacja języka programowania Kotlin,” [Online]. Available: https://kotlinlang.org/docs/home.html. [Data uzyskania dostępu: 29 listopad 2023].

[7] D. K. Kornaś, „Porównanie sposobów przechowywania danych w systemie Android,” *Journal of Computer Sciences Institute,* pp. 378--382, https://doi.org/10.35784/jcsi.2759, 2021 vol. 21.

[8] „https://www.flaticon.com/free-icons/logout,” 26 Październik 2023. [Online]. [Data uzyskania dostępu: 26 Październik 2023].

[9] „Android Developers - Activities,” Google, [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities. [Data uzyskania dostępu: 14 Grudzień 2023].

[10] „Android Developers - Fragments,” Google, [Online]. Available: https://developer.android.com/guide/fragments. [Data uzyskania dostępu: 14 Grudzień 2023].

[11] „Android Developers - Data Storage,” [Online]. Available: https://developer.android.com/training/data-storage/room#java. [Data uzyskania dostępu: 22 Grudzień 2023].

Dodatki

Spis skrótów i symboli

SMS	krótka wiadomość tekstowa (ang. *Short Message Service*)
PK	klucz główny w tabeli bazy danych (ang. *Primary Key*)
FK	klucz obcy w tabeli bazy danych (ang. *Foreign Key*)
DAO	komponent umożliwiający komunikacje z bazą danych (ang. *Data Access Object*)
URI	w systemie Android, identyfikator umożliwiający jednoznaczne odwołanie do zasobów takich jak pliki lub adresy stron (ang. *Uniform Resource Identifier*)

Źródła

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    <include
        android:id="@+id/appBar"
        layout="@layout/app_bar"/>

    <FrameLayout
        android:id="@+id/frame"
        android:layout_width="0dp"
        android:layout_height="0dp"
app:layout_constraintBottom_toTopOf="@+id/bottomNavigationView"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/appBar">
    </FrameLayout>

  <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottomNavigationView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:labelVisibilityMode="labeled"
        app:backgroundTint="@android:color/transparent"
        app:menu="@menu/bottom_nav_menu"/>
    </androidx.constraintlayout.widget.ConstraintLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/navigationView"
        android:layout_width="wrap_content"
        android:background="@color/white"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/nav_header"
        android:fitsSystemWindows="true"
        app:menu="@menu/nav_menu"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

Rysunek 1 Plik .xml ze zdefiniowanym panelem widoku.

```
1  public static Database getInstance(Context context){
2  if(INSTANCE == null){
3  synchronized (Database.class){
4      if(INSTANCE == null){
5      INSTANCE = Room.databaseBuilder(
6          context.getApplicationContext(),
7          Database.class,"Database")
8              .allowMainThreadQueries()
9              .addCallback(new Callback() {
10                 @Override
11                 public void onCreate(@NonNull
12                 SupportSQLiteDatabase db) {
13                     super.onCreate(db);
14                     Executors.newSingleThreadScheduledExecutor()
15                     .execute(new Runnable() {
16                         @Override
17                         public void run() {
18                             ...
19                             inicjalizacja tabeli Produkty
20                             ...
21                         }
22                     });
23                 }
24             })
25             .build();
26     }
27 }
28 }
29 return INSTANCE;
30 }
```

Rysunek 2 Metoda dostępu do instancji obiektu bazy danych z klasy Database

Lista dodatkowych plików, uzupełniających tekst pracy

W systemie, do pracy dołączono dodatkowe pliki zawierające:

- Źródło programu umożliwiające jego instalację.

Spis rysunków