

POLITECHNIKA WROCŁAWSKA

BAZY DANYCH 2

PROJEKT

Dokumentacja projektu

STACJONARNY SKLEP MUZYCZNY

TERMIN ZAJĘĆ: PONIEDZIAŁEK 13:15

Autorzy:

Jakub BIAŁECKI, 218281

Damian KORZEKWA, 226132

Prowadzący:

dr inż. Roman PTAK

Wrocław 2018

1 Wstęp

1.1 Cel projektu

Celem projektu było stworzenie bazy danych wraz z aplikacją dostępową dla stacjonarnego sklepu muzycznego.

1.2 Zakres projektu

Utworzona w projekcie baza danych zawiera informacje o produktach, które są sprzedawane w sklepie.

Dodatkowo, baza danych ma za zadanie przechowywać historię transakcji - zarówno transakcje sprzedaży towaru klientom, jak i zamówienia przez sklep produktów z zewnętrznych magazynów.

Aplikacja dostępową ma umożliwiać właścicielowi i pracownikom sklepu dostęp do tej bazy - łatwe dodawanie nowych produktów, edycję już istniejących, usuwanie produktów oraz przeglądanie historii transakcji.

2 Analiza wymagań

2.1 Opis działania i schemat logiczny systemu

Sklep muzyczny „Guitar Hero” znajduje się we Wrocławiu przy ulicy Ruskiej 11. Sklep zajmuje się sprzedażą instrumentów muzycznych. Dodatkowo, w ofercie sklepu, znajdują się artykuły ołomuzyczne.

Sklep prowadzony jest przez właściciela i to on decyduje o tym, jaki asortyment znajduje się w sklepie. Ponadto sklep zatrudnia pracowników, którzy zajmują się doradzaniem klientom sklepu, obsługą zamówień, sprzedażą oraz zarządzaniem towarem w sklepie.

W związku z tym, że lokal nie jest duży, sklep posiada magazyn, w którym przechowywane są instrumenty. W sklepie wystawiona jest jedynie część towaru. Magazyn znajduje się około 500 metrów od sklepu. Towar może być dostarczony z magazynu do sklepu na prośbę klienta lub klient może odebrać towar z magazynu za okazaniem paragonu lub faktury.

Aplikacja wraz z bazą danych ma za zadanie ułatwić właścicielowi, jak i pracownikom sklepu, zarządzanie oferowanymi produktami oraz historią transakcji.

2.2 Wymagania funkcjonalne

- właściciel sklepu ma możliwość dodawania nowych produktów do bazy danych
- właściciel sklepu ma możliwość pełnej modyfikacji dodanych produktów w bazie danych

- właściciel ma możliwość usuwania (wycofania z oferty) produktów z bazy danych
- właściciel ma możliwość dodawania nowych kategorii produktów
- zarówno pracownik jak i właściciel mogą przeglądać historię transakcji
- właściciel i pracownik mają możliwość sprzedaży produktu klientowi (usunięcia egzemplarza produktu z bazy danych) i tym samym dodania takiej transakcji do historii
- właściciel i pracownik mają możliwość zamówienia kolejnych egzemplarzy produktów (dodania nowego egzemplarza produktu do bazy danych) i tym samym dodania takiej transakcji do historii

2.3 Wymagania niefunkcjonalne

- baza danych będzie przechowywać informacje o produktach oferowanych w sklepie wraz ze szczegółowymi informacjami na temat produktu (id, nazwa produktu, cena katalogowa, cena promocyjna, stan magazynowy, stan produktu (nowy, używany, powystawowy), kategoria, opis)
- baza danych będzie przechowywać również historię transakcji wraz ze szczegółowymi informacjami (id transakcji, kwota, strona transakcji (klient-sklep, sklep-producent), data transakcji, produkt transakcji, ilość produktów)

2.3.1 Wykorzystywane technologie i narzędzia

Baza danych będzie wykorzystywać technologię SQLite. Aplikacja dostępowa będzie napisana w języku Java z interfejsem graficznym wykorzystującym technologię JavaFX. Dostęp do bazy danych z poziomu aplikacji w Javie planujemy zapewnić przy użyciu biblioteki JDBC lub SQLJet.

Planujemy wykorzystać IntelliJ IDEA Community jako główne środowisko programistyczne oraz Gita jako system kontroli wersji.

2.3.2 Wymagania dotyczące rozmiaru bazy danych

Szacujemy, że w sklepie będzie w sprzedaży około kilka/kilkanaście tysięcy produktów. Nie wszystkie będą dostępne od razu w sklepie, jednak baza danych musi przechowywać o nich informacje.

Zakładamy, że sklep będzie realizował średnio 15 - 30 transakcji dziennie. Szacujemy ilość transakcji w takiej bazie na około 60000 po około pięciu latach funkcjonowania sklepu.

2.3.3 Wymagania dotyczące bezpieczeństwa systemu

Z racji tego, iż sklep prowadzi jedynie sprzedaż stacjonarną, baza danych nie wymaga specjalistycznych zabezpieczeń. Podstawowym zabezpieczeniem bazy będzie system logowania dla pracowników (w tym dla właściciela sklepu).

2.4 Przyjęte założenia projektowe

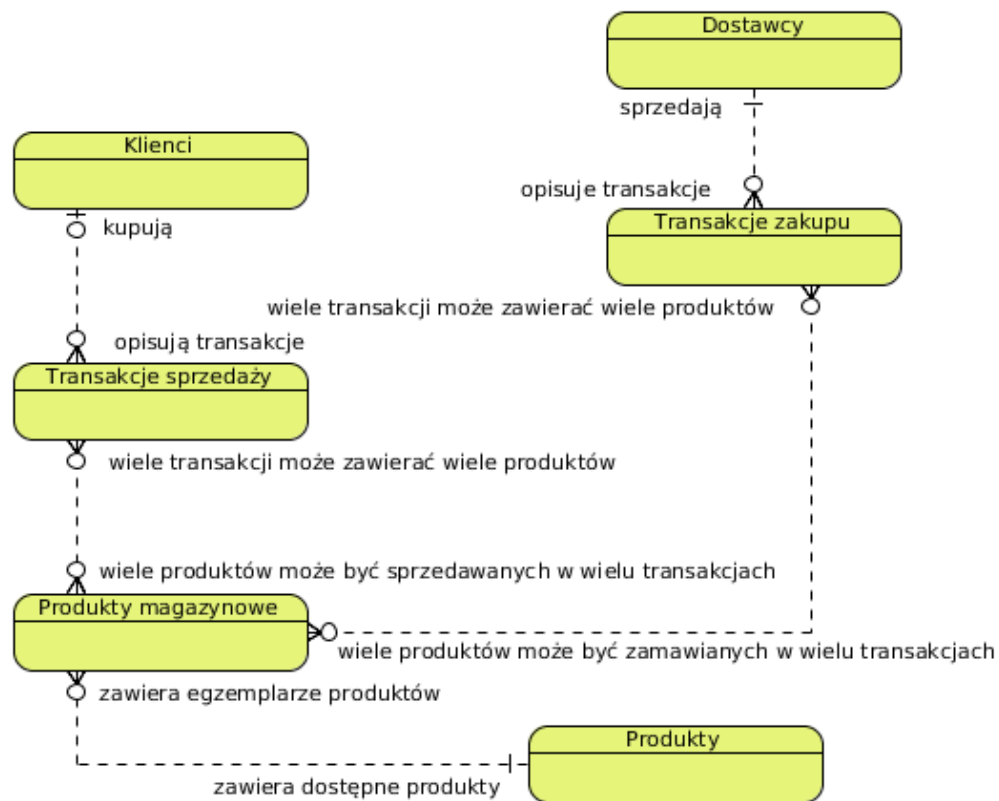
- relacje pomiędzy kluczami głównymi a obcymi
- odpowiednie typy danych i ich wielkości
- użyć odpowiedniego nazewnictwa tabel, kluczy głównych i kluczy obcych
- utworzenie widoków, zwracające istotne dane w sensie działania aplikacji
- użycie złączeń
- utworzenie funkcji dokonujących na danych w tabelach takich operacji jak: dodawanie, usuwanie i aktualizacja

3 Projekt systemu

3.1 Projekt bazy danych

3.1.1 Analiza rzeczywistości i uproszczony model konceptualny

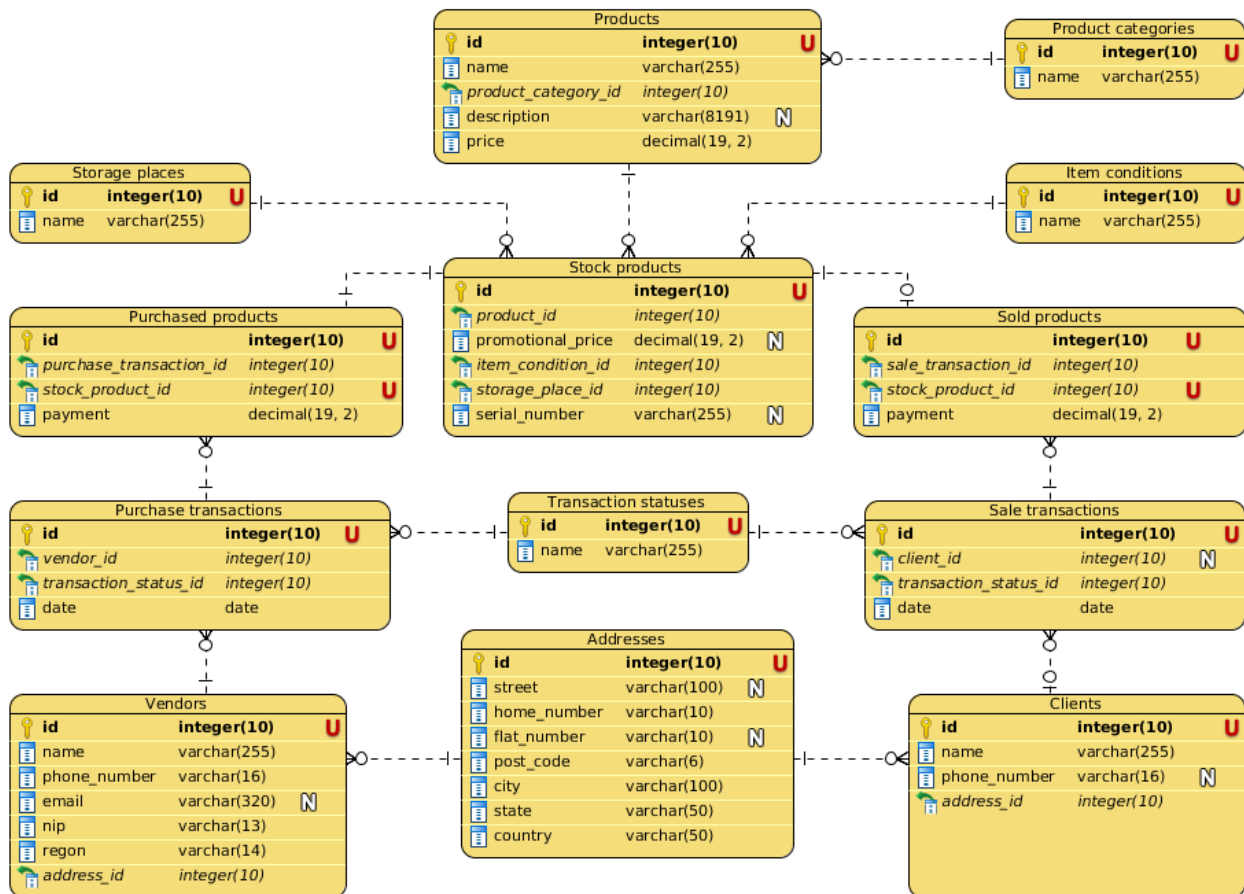
Na podstawie analizy przedstawionej w rozdziale 2, stworzyliśmy konceptualny bazy danych dla sklepu, który został przedstawiony na rysunku 1.



Rysunek 1: Model konceptualny

3.1.2 Model logiczny i normalizacja

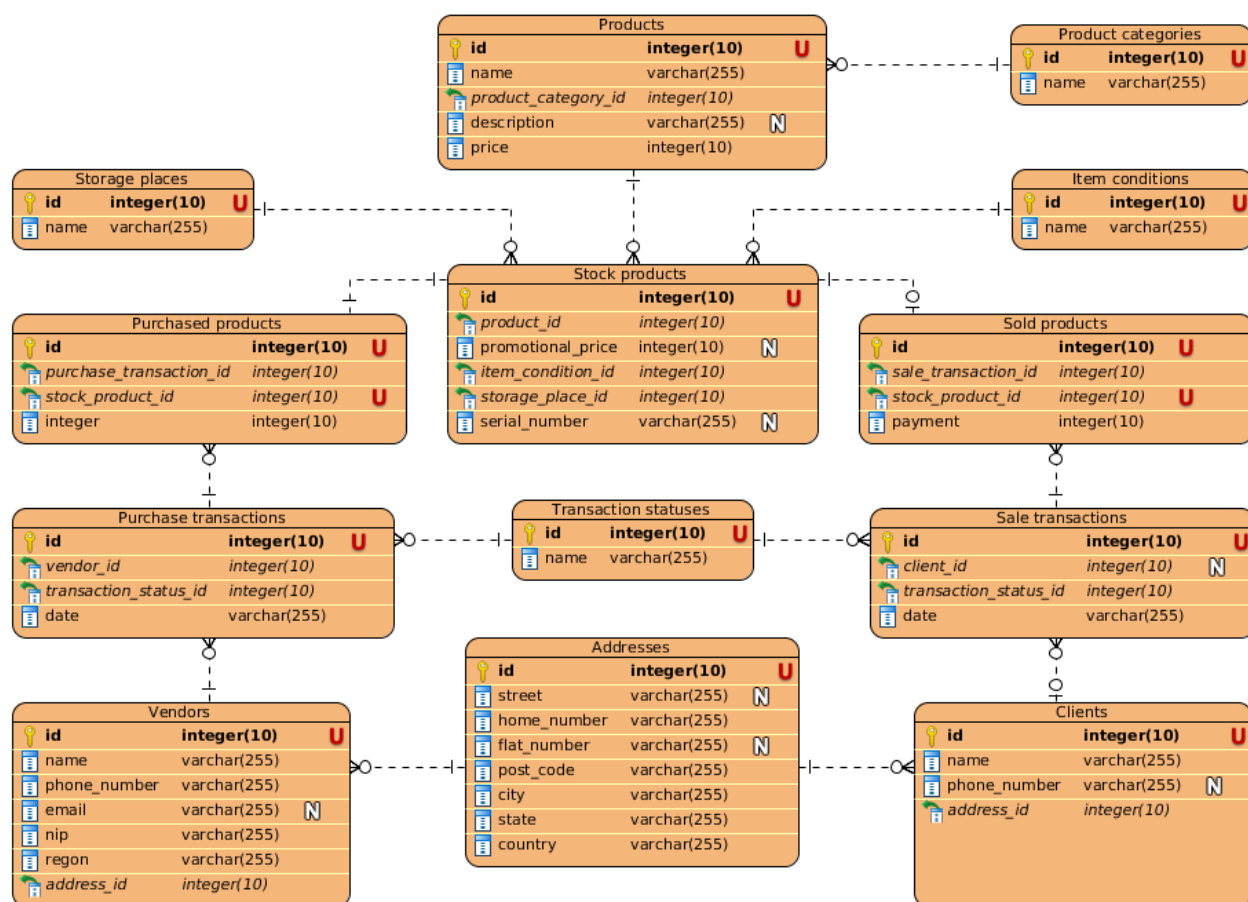
Następnym krokiem było rozbudowanie modelu konceptualnego o nowe tabele, uzupełnienie tabel o kolumny oraz zdefiniowanie bardziej szczegółowych relacji między tabelami. Rysunek 2 przedstawia gotowy model logiczny.



Rysunek 2: Model logiczny

3.1.3 Model fizyczny i ograniczenia integralności danych

Po stworzeniu modelu logicznego, przeszliśmy do stworzenia modelu fizycznego bazy danych. Po głębszej analizie, okazało się, że w SQLite nie ma możliwości wprowadzenia ograniczeń wielkości danych w poszczególnych kolumnach. Dodatkowo, ustalone przez nas wstępnie typy danych należało zmienić na zdecydowanie bardziej uproszczone. Rysunek 3 przedstawia gotowy model.



Rysunek 3: Model logiczny

Wszystkie możliwe typy danych sprowadziliśmy do dwóch: `varchar(255)` oraz `integer(10)`. Z powodu ograniczeń programu Visual Paradigm nie byliśmy w stanie zamienić tych typów na te, które rzeczywiście występują w SQLite, czyli odpowiednio `TEXT` oraz `INTEGER`.

3.1.4 Inne elementy schematu – mechanizmy przetwarzania danych

- Indeksy

Celem indeksów jest przyspieszenie operacji wyszukiwania, więc starano się znaleźć atrybuty, po których pracownik będzie najczęściej przeszukiwał tabelę. Oczywiście indeksy

mają swoje wady. Wydłużają one czas wykonywania operacji wstawiania, modyfikacji i usuwania w tabeli.

Zdecydowaliśmy się na stworzenie następujących indeksów:

- `stock_products_index` – `product_id`, `storage_place_id`, `serial_number`
- `clients_index` – `name`, `phone_number`, `address_id`
- `vendors_index` – `name`, `address_id`, `phone_number`

- **Widoki**

Widoki umożliwiają dostęp do podzbioru kolumn i wierszy tabel lub tabeli.

Gdy korzysta się często z jakiejś tabeli ze stałymi parametrami warto stworzyć widok, w celu zaoszczędzenia czasu. Widoku używa się także gdy jest potrzeba ograniczenia dostępu użytkownika.

Zdecydowaliśmy się na zdefiniowanie następujących widoków:

- z ograniczeniem `WITH READ ONLY` tabeli `products` i `product_categories` do której dostęp będą mieli pracownicy bez uprawnień właściciela

3.1.5 Projekt mechanizmów bezpieczeństwa na poziomie bazy danych

- **Kopie zapasowe**

Dane w bazie danych są archiwizowane raz na pół roku. Okresowo kopie zapasowe powinny być testowane pod względem rzeczywistej możliwości wykorzystania ich w sytuacji koniecznej. Dane dotyczące płatności są przechowywane przez 5 lat. Dane o klientach i zamówieniach są przechowywane przez 5 lat.

- **Uprawnienia użytkowników**

W - uprawnienia właściciela sklepu

P - uprawnienia pracownika

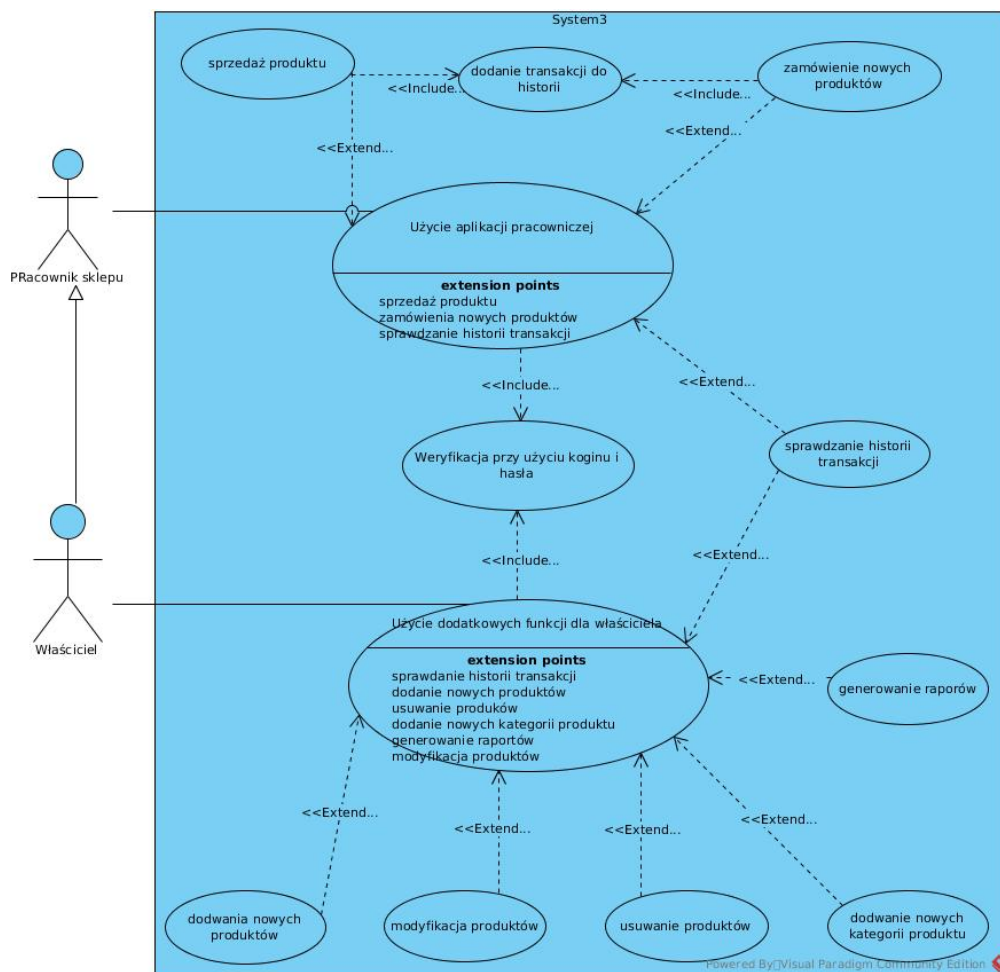
Tabela 1: Uprawnienia użytkowników

Tabela	Dodawanie	Modyfikacja	Usuwanie	Przeglądanie
addresses	W, P	W	W	W
clients	W, P	W	W	W
item_conditions	W	W	W	W, P
product_categories	W	W	W	W, P
products	W	W	W	W, P
purchase_transactions	W, P	W	W	W, P
purchased_products	W, P	W	W	W, P
sale_transactions	W, P	W	W	W, P
sold_products	W, P	W	W	W, P
stock_products	W, P	W, P	W, P	W, P
storage_places	W	W	W	W, P
transaction_statuses	W	W	W	W
vendors	W, P	W	W	W

3.2 Projekt aplikacji użytkownika

3.2.1 Architektura aplikacji i diagramy projektowe

Utworzyliśmy następujący diagram przypadków użycia aplikacji, widoczny na rysunku 4.



Rysunek 4: Diagram przypadków użycia

3.2.2 Interfejs graficzny i struktura menu

Wstępny projekt interfejsu wykonaliśmy w programie Pencil. Efekty można zobaczyć na rysunkach 5 i 6.

The screenshot shows two parts of the GuitarHero 2.0 interface. On the left is a menu with tabs: Sprzedaż, Raporty, Zarządzanie, and Historia transakcji. Below the tabs are two buttons: SPRZEDAJ PRODUKT and DODAJ TRANSAKCJE DO HISTORII. Below the buttons is a table with product information.

Nazwa produktu	Ilość	Cena produktu	Numer seryjny produktu
Gitara akustyczna	63	429.99	DJGKDL43SERG
Bęben basowy	51	219.99	ROTLGK4S11FG
Pokrowiec na gitarę	21	119.99	GRKF43
Werbel	4	329.99	TKGL5S32RREF
Klarnet	3	599.99	FGKRL6693FD

On the right is a form titled 'Sprzedaj produkt'. It contains input fields for: Imię (Jan), Nazwisko (Kowalski), Nr telefonu (587473849), Ulica (Piastowska), Nr domu (53), Nr mieszkania (3), Miasto (Wrocław), Kod pocztowy (50-394), and Kraj (Polska). There are also fields for Ilość towaru (2) and Data wysyłki (20-12-2017).

Rysunek 5: Projekt interfejsu graficznego

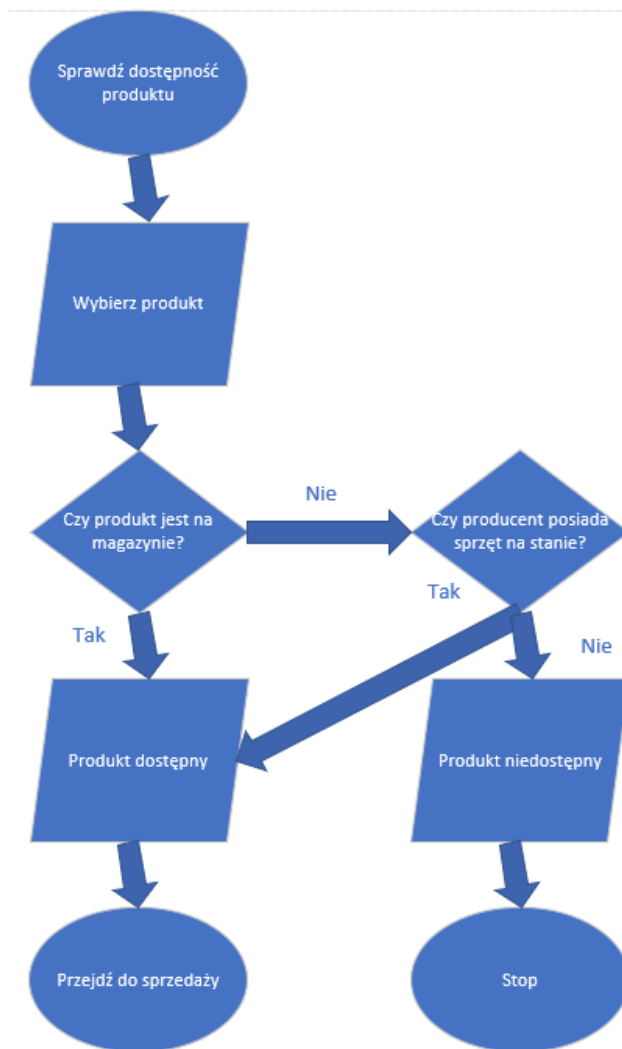
The screenshot shows two parts of the GuitarHero 2.1 interface. On the left is a menu with tabs: Sprzedaż, Raporty, Zarządzanie, and Historia transakcji. Below the tabs is a button: Sprzedaj produkt. Below the button is a table with product information.

Nazwa produktu	Ilość	Cena produktu
Gitara akustyczna	63	429.99
Bęben basowy	21	381.99
Pokrowiec na gitarę	42	129.99
Werbel	6	329.99
Klarnet	3	519.99

On the right is a form titled 'GuitarHero 2.1 - Sprzedaj produkt'. It contains input fields for: Imię (Jan), Nazwisko (Kowalski), Telefon (999999999), Ulica (Piastowska), Nr domu (23), Nr mieszkania (3), Miasto (Wrocław), Kod pocztowy (50-003), and Kraj (Polska). There are also fields for Ilość towaru (2) and Data wysyłki (12-12-2017). At the bottom are two buttons: Akceptuj and Anuluj.

Rysunek 6: Projekt interfejsu graficznego

3.2.3 Projekt wybranych funkcji systemu



Rysunek 7: Projekt funkcji systemu

3.2.4 Metoda podłączania do bazy danych – integracja z bazą danych

Jako bazę danych dla aplikacji wybraliśmy SQLite, ponieważ jest to lekka baza i w pełni wystarczająca do obsługi sklepu.

Dostęp do bazy będziemy zapewniać przy użyciu JDBC w Javie.

3.2.5 Projekt zabezpieczeń na poziomie aplikacji

Aplikacja zapewni będzie zabezpieczenia do bazy danych za pomocą loginu i hasła podawanego w aplikacji.

4 Implementacja systemu baz danych

4.1 Tworzenie tabel i definiowanie ograniczeń

Do stworzenia tabel wykorzystaliśmy program DB Browser, który zapewnił możliwość stworzenia praktycznie całej bazy danych bez znajomości języka SQL. Kod dla poszczególnych tabel wygenerowanych przez program prezentujemy poniżej.

Tabela addresses:

```
CREATE TABLE 'addresses' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'street' TEXT,  
  'home_number' TEXT NOT NULL,  
  'flat_number' TEXT,  
  'post_code' TEXT NOT NULL,  
  'city' TEXT NOT NULL,  
  'state' TEXT NOT NULL,  
  'country' TEXT NOT NULL  
);
```

Tabela clients:

```
CREATE TABLE 'clients' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'name' TEXT NOT NULL,  
  'phone_number' TEXT,  
  'address_id' INTEGER NOT NULL,  
  FOREIGN KEY('address_id') REFERENCES 'addresses'('id')  
);
```

Tabela item_conditions:

```
CREATE TABLE 'item_conditions' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'name' TEXT NOT NULL  
);
```

Tabela product_categories:

```
CREATE TABLE 'product_categories' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'name' TEXT NOT NULL  
);
```

Tabela products:

```
CREATE TABLE 'products' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'name' TEXT NOT NULL,  
  'product_category_id' INTEGER NOT NULL,  
  'description' TEXT,  
  'price' INTEGER NOT NULL,  
  FOREIGN KEY('product_category_id') REFERENCES 'product_categories'('id')  
);
```

Tabela purchase_transactions:

```
CREATE TABLE 'purchase_transactions' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'vendor_id' INTEGER NOT NULL,  
  'transaction_status_id' INTEGER NOT NULL,  
  'date' TEXT NOT NULL,  
  FOREIGN KEY('transaction_status_id') REFERENCES 'transaction_statuses'('id'),  
  FOREIGN KEY('vendor_id') REFERENCES 'vendors'('id')  
);
```

Tabela purchased_products:

```
CREATE TABLE 'purchased_products' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'purchase_transaction_id' INTEGER NOT NULL,  
  'stock_product_id' INTEGER NOT NULL UNIQUE,  
  'payment' INTEGER NOT NULL,  
  FOREIGN KEY('purchase_transaction_id') REFERENCES  
    'purchase_transactions'('id'),  
  FOREIGN KEY('stock_product_id') REFERENCES 'stock_products'('id')  
);
```

Tabela sale_transactions:

```
CREATE TABLE 'sale_transactions' (  
  'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,  
  'client_id' INTEGER,  
  'transaction_status_id' INTEGER NOT NULL,  
  'date' TEXT NOT NULL,  
  FOREIGN KEY('transaction_status_id') REFERENCES 'transaction_statuses'('id'),  
  FOREIGN KEY('client_id') REFERENCES 'clients'('id')  
);
```

Tabela sold_products:

```
CREATE TABLE 'sold_products' (  

```

```

'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
'sale_transaction_id' INTEGER NOT NULL,
'stock_product_id' INTEGER,
'payment' INTEGER NOT NULL,
FOREIGN KEY('stock_product_id') REFERENCES 'stock_products'('id'),
FOREIGN KEY('sale_transaction_id') REFERENCES 'sale_transactions'('id')
);

```

Tabela stock_products:

```

CREATE TABLE 'stock_products' (
'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
'product_id' INTEGER NOT NULL,
'promotional_price' INTEGER,
'item_condition_id' INTEGER NOT NULL,
'storage_place_id' INTEGER NOT NULL,
'serial_number' TEXT,
FOREIGN KEY('item_condition_id') REFERENCES 'item_conditions'('id'),
FOREIGN KEY('product_id') REFERENCES 'products'('id'),
FOREIGN KEY('storage_place_id') REFERENCES 'storage_places'('id')
);

```

Tabela storage_places:

```

CREATE TABLE 'storage_places' (
'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
'name' TEXT NOT NULL
);

```

Tabela transaction_statuses:

```

CREATE TABLE 'transaction_statuses' (
'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
'name' TEXT NOT NULL
);

```

Tabela vendors:

```

CREATE TABLE 'vendors' (
'id' INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
'name' TEXT NOT NULL,
'phone_number' TEXT NOT NULL,
'email' TEXT,
'nip' TEXT NOT NULL,
'regon' TEXT NOT NULL,
'address_id' INTEGER NOT NULL,
FOREIGN KEY('address_id') REFERENCES 'addresses'('id')
);

```

);

4.2 Implementacja mechanizmów przetwarzania danych

- Indeksy

Indeksy zostały napisane ręcznie, a oto ich nazwy oraz kody:

– stock_products_index

```
CREATE INDEX 'stock_products_index' ON 'stock_products' (  
    'product_id',  
    'promotional_price',  
    'storage_place_id' ASC,  
    'serial_number'  
);
```

– clients_index

```
CREATE INDEX 'clients_index' ON 'clients' (  
    'name' ASC,  
    'phone_number',  
    'address_id'  
);
```

– vendors_index

```
CREATE INDEX 'vendors_index' ON 'vendors' (  
    'name' ASC,  
    'phone_number',  
    'address_id'  
);
```

- Widoki

Widok products_view

```
CREATE VIEW products_view AS SELECT  
    p.product_id AS id,  
    p.name AS name,  
    c.name AS product_category_name,  
    p.description AS description,  
    p.price AS price  
FROM products p  
INNER JOIN product_categories c  
ON p.product_category_id = c.id;
```

4.3 Implementacja uprawnień i innych zabezpieczeń

Niestety w SQLite nie ma możliwości tworzenia uprawnień i nasza baza zostanie zabezpieczona jedynie w aplikacji loginem oraz hasłem dostępu.

4.4 Testowanie bazy danych na przykładowych danych

- dodawanie rekordu do tabeli

```
INSERT INTO addresses
('id', 'street', 'home_number', 'flat_number', 'post_code', 'city', 'state',
'country')
VALUES
(3000, "kochanowskiego", "10", "2", "50-234", "Wroclaw", "dolnoslaskie",
'polska');
```

id	street	home_number	flat_number	post_code	city	state	country
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	Oławska	32	8	55-123	Wrocław	Dolnośląskie	Polska
2	Kochanowskie...	53	1	51-602	Wrocław	Dolnośląskie	Polska
3	Płocka	30a	NULL	01-231	Warszawa	Mazowieckie	Polska
4	Dr. Külz-Ring	594	12	01-067	Drezno	Saksonia	Niemcy
3000	kochanowskiego	10	2	50-234	Wroclaw	dolnoslaskie	polska

Rysunek 8: Wynik operacji wstawiania

- edycja rekordów w tabeli

id	name	oduct_category_	description	list_price
Filter	Filter	Filter	Filter	Filter
1	Gitara	1	opis	19999
2	Perkusja(kom...	2	opis	59999
3	Klarnet	3	opis	31999
4	Pokrowiec na ...	3	opis	8999
5	Struny	1	opis	3499
6	Werbel	2	opis	39999

Rysunek 9: Tabela przed edycją

```
UPDATE products SET description = 'Promocja'
WHERE list_price < 9000;
```

id	name	oduct_category_	description	list_price
Filter	Filter	Filter	Filter	Filter
1	Gitara	1	opis	19999
2	Perkusja(kom...	2	opis	59999
3	Klarnet	3	opis	31999
4	Pokrowiec na ...	3	Promocja	8999
5	Struny	1	Promocja	3499
6	Werbel	2	opis	39999

Rysunek 10: Tabela po edycji

- usuwanie rekordów

id	name	phone	address_id
Filter	Filter	Filter	Filter
1	Nowak	565431324	1
2	Józefowicz	645344523	1
3	Urban	734345431	2

Rysunek 11: Tabela przed edycją

```
DELETE FROM clients WHERE name = 'Nowak';
```

id	name	phone	address_id
Filter	Filter	Filter	Filter
2	Józefowicz	645344523	1
3	Urban	734345431	2

Rysunek 12: Tabela po edycji

5 Implementacja i testy aplikacji

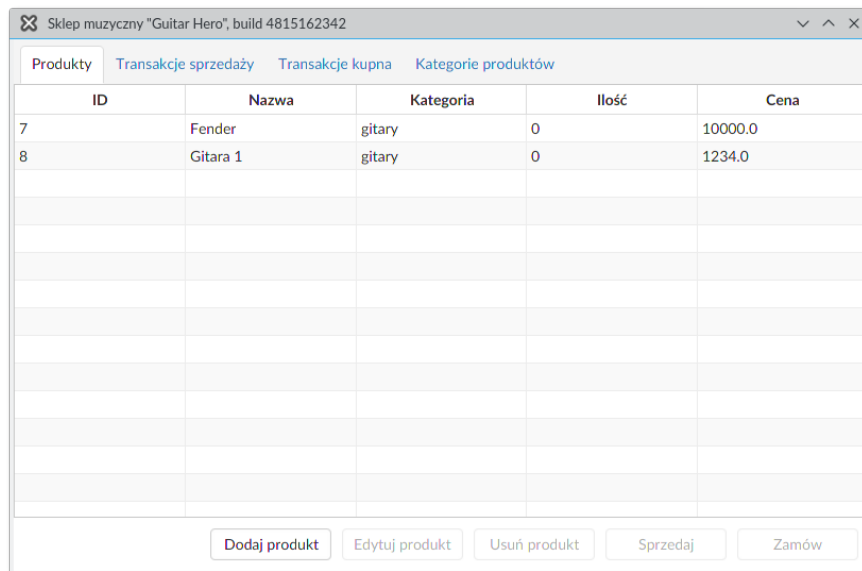
5.1 Instalacja i konfigurowanie systemu

Na chwilę obecną, oprogramowanie dostarczamy w postaci kodu źródłowego oraz pliku z bazą danych. Niestety na chwilę obecną nie byliśmy w stanie wygenerować gotowych plików binarnych, które można byłoby uruchomić przy użyciu maszyny wirtualnej Javy.

Program można bez problemu uruchomić korzystając z dowolnego środowiska programistycznego do Javy (na przykład IntelliJ IDEA).

5.2 Instrukcja użytkowania aplikacji

Po uruchomieniu aplikacji, przywita nas widok, jak na rysunku 13.



ID	Nazwa	Kategoria	Ilość	Cena
7	Fender	gitary	0	10000.0
8	Gitara 1	gitary	0	1234.0

Rysunek 13: Domyślny widok aplikacji

Z tego miejsca możemy zobaczyć, jakie produkty są w naszej bazie oraz wykonywać na nich podstawowe operacje dodawania, edycji oraz usuwania.

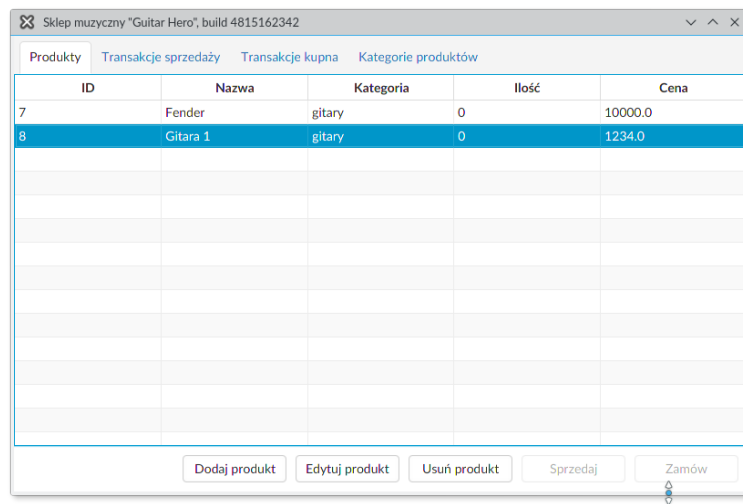
Możemy przenieść się do innej zakładki, gdzie możemy podejrzeć transakcje sprzedaży i zakupu (zamówienia towaru przez sklep) oraz przenieść się do edycji kategorii produktów.

Aplikacja wydaje się być bardzo intuicyjna w użytkowaniu.

5.3 Testowanie opracowanych funkcji systemu

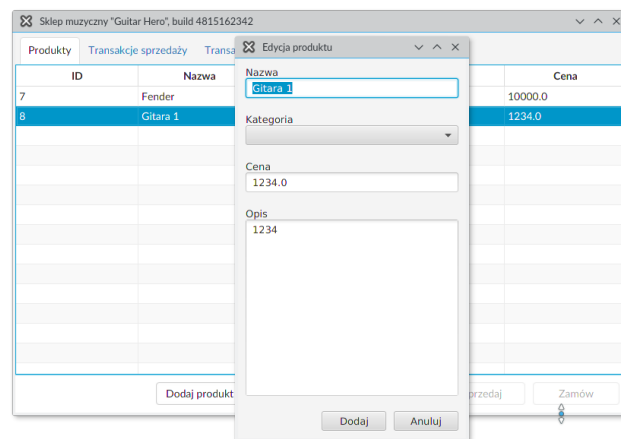
Przeprowadźmy test poprawności działania aplikacji na przykładzie edycji przykładowego produktu.

Rysunek 14 przedstawia stan aplikacji przed rozpoczęciem testu.



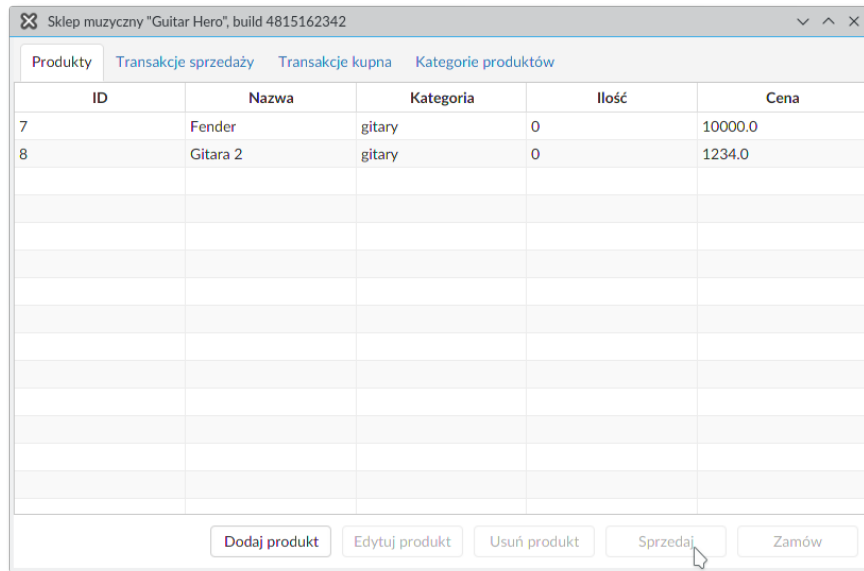
Rysunek 14: Zaznaczony element tabeli

Następnie naciskamy 'Dodaj produkt' i tym samym przechodzimy do edycji produktu. Efekt możemy zobaczyć na rysunku 15.



Rysunek 15: Okno edycji produktu

Modyfikujemy dane w polu 'Nazwa' na *Gitara2* i sprawdzamy jaki będzie efekt. Po kliknięciu na 'Dodaj' program powróci do poprzedniego widoku. Rysunek 16 przedstawia efekt modyfikacji.



Rysunek 16: Zmodyfikowane dane

Podobne testy zostały wykonane dla pozostałych funkcjonalności aplikacji tj. dodawania i usuwania produktów oraz dodawania, modyfikacji i usuwania kategorii produktów.

5.4 Omówienie wybranych rozwiązań programistycznych

5.4.1 Implementacja interfejsu dostępu do bazy danych

W celu dostępu do bazy danych wykorzystaliśmy sterownik JDBC w Javie. Umożliwia on komunikację z bazą danych przy użyciu zapytań SQL. Dane są zwracane w postaci obiektów `ResultSet`, które można następnie odczytać.

Dla każdej tabeli w bazie, zaimplementowaliśmy jej odpowiednik jako model danych w Javie. Dla przykładu, tak prezentuje się klasa *ItemCondition*, która reprezentuje jeden wiersz tabeli `item_conditions`:

```
public class ItemCondition implements TableRow {
    private final int id;
    private final String name;

    public ItemCondition(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public int getId() {
        return id;
    }
}
```

```

    }

    public String getName() {
        return name;
    }

    @Override
    public PreparedStatement prepareInsertStatement(Connection connection, String
        sql) throws SQLException {
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        preparedStatement.setString(1, name);
        return preparedStatement;
    }

    @Override
    public PreparedStatement prepareUpdateStatement(Connection connection, String
        sql) throws SQLException {
        PreparedStatement preparedStatement = connection.prepareStatement(sql);
        preparedStatement.setString(1, name);
        preparedStatement.setInt(2, id);
        return preparedStatement;
    }
}

```

Jak można zauważyć, klasa ta nadpisuje pewne metody z interfejsu *TableRow*. Interfejs *TableRow* ma za zadanie ułatwić komunikację z bazą danych, udostępniając możliwość utworzenia tzw. *statements*, które są wykorzystywane przy dodawaniu wierszy do tabeli, bądź edycji tych wierszy.

Interfejs ten posiada zdefiniowane domyślne metody, które nie muszą być nadpisywane przez klasy implementujące. Tak prezentuje się klasa *TableRow*, wraz ze wspomnianymi metodami:

```

public interface TableRow {
    public int getId();

    public default void insertInto(Database database, String sql) throws
        SQLException {
        prepareInsertStatement(database.connection(), sql)
            .executeUpdate();
    }

    public default void updateIn(Database database, String sql) throws
        SQLException {
        prepareUpdateStatement(database.connection(), sql)
            .executeUpdate();
    }
}

```

```

public default void deleteFrom(Database database, String sql) throws
    SQLException {
    prepareDeleteStatement(database.connection(), sql)
        .executeUpdate();
}

public PreparedStatement prepareInsertStatement(Connection connection, String
    sql) throws SQLException;

public PreparedStatement prepareUpdateStatement(Connection connection, String
    sql) throws SQLException;

public default PreparedStatement prepareDeleteStatement(Connection connection,
    String sql) throws SQLException {
    PreparedStatement preparedStatement = connection.prepareStatement(sql);
    preparedStatement.setInt(1, getId());
    return preparedStatement;
}
}

```

Domyślnie zostały zdefiniowane 4 metody, które nie różnią się wcale w zależności od tabeli, więc zrezygnowano z ich nadpisania w klasach implementujących interfejs. 3 metody są odpowiedzialne za wstawianie, modyfikację oraz usuwanie danych z tabeli. Czwarta metoda jest podobna do tych wspomnianych wcześniej w klasie *ItemCondition*. Służy do przygotowania tzw. *statement* dla usuwania wiersza z tabeli. Metoda ta również jest identyczna dla wszystkich klas implementujących interfejs.

Podobnie jak pojedyncze wiersze, zostały reprezentowane całe tabele. Zdecydowano się stworzyć szereg klas, które będą umożliwiać pobranie całych tabel. W celu ułatwienia zadania, dodano interfejs *Table*, który reprezentuje tabelę:

```

public interface Table {
    public default ArrayList<? extends TableRow> selectFrom(Database database,
        String sql) throws SQLException {
        Statement statement = database.connection().createStatement();
        ArrayList<? extends TableRow> table = toList(database,
            statement.executeQuery(sql));
        return table;
    }

    public ArrayList<? extends TableRow> toList(Database database, ResultSet r)
        throws SQLException;

    public default int count(Database database, String sql, int id) throws
        SQLException {
        Statement statement = database.connection().createStatement();

```

```
        ResultSet resultSet = statement.executeQuery(sql);
        return resultSet.getInt(1);
    }
}
```

Najważniejszym elementem jest tutaj metoda `selectFrom`, która zwraca kolekcję zawierającą elementy dziedziczące po *TableRow*. Umożliwia to szybkie "wybranie" całej tabeli i osadzenie jej w kolekcji.

Kolejnym elementem implementacji bazy danych jest klasa *Database*, która odpowiada za nawiązanie połączenia z bazą danych. Obiekt tej klasy jest przekazywany do wszystkich operacji, które wykonujemy w bazie.

```
public class Database {
    Connection connection;

    public Database(String connectionString) throws SQLException {
        connection = DriverManager.getConnection(connectionString);
    }

    public Connection connection() {
        return connection;
    }

    public void closeConnection() throws SQLException {
        connection.close();
    }
}
```

Tworząc obiekt typu *Database* od razu nawiązujemy połączenie z bazą, którą podajemy przy użyciu `connectionString`.

5.4.2 Implementacja wybranych funkcjonalności systemu

Dodawanie nowych produktów do bazy danych zostało zrealizowane w następujący sposób.

W JavaFX, aby wywołać akcję przy użyciu elementów interfejsu, należy zdefiniować kontroler. Zdefiniowaliśmy taki kontroler o nazwie *ProductsController* dla widoku, gdzie znajduje się nasza tabela z produktami. W tym miejscu znalazła się metoda, która uruchamia się po naciśnięciu przycisku:

```
@FXML
private void handleNewProduct() {
    main.showAddProductDialog(null, "Dodawanie produktu");
}
```

Metoda wywołuje metodę `showAddProductDialog(ProductView, String)` w klasie *Main*. Metoda ta odpowiada za wywołanie nowego okna, które umożliwia dodanie nowego produktu. Dla nowo stworzonego okna także zdefiniowano kontroler, który wywołuje odpowiednią metodę po naciśnięciu przycisku 'Dodaj':

```
@FXML
private void handleAdd() {
    if (isInputValid()) {
        int id = 0;
        if (productView != null) {
            id = Integer.parseInt(productView.getId());
        }
        Product product = new Product.Builder(id)
            .name(nameTextField.getText())
            .productCategoryId(productCategoryComboBox.getValue().getId())
            .description(descriptionTextArea.getText())
            .price((int) Float.parseFloat(priceTextField.getText()) * 100)
            .build();
        stage.close();
        if (productView != null) {
            try {
                product.updateIn(main.data().database(), "UPDATE products SET name =
                    ?, product_category_id = ?, description = ?, price = ? WHERE id =
                    ?");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        } else {
            try {
                product.insertInto(main.data().database(), "INSERT INTO products(name,
                    product_category_id, description, price) values(?, ?, ?, ?)");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        overviewController.productsController().update();
    }
}
```

Metoda, na początku sprawdza czy wprowadziliśmy odpowiednie dane w polach. Następnie sprawdzamy czy produkt (*ProductView*), już istnieje (będzie edytowany) czy nie istnieje, i należy go utworzyć.

W następnym kroku tworzymy nowy obiekt na podstawie pobranych danych z formularzy. Zamykamy okno i dodajemy produkt do bazy, bądź go modyfikujemy.

W ostatnim kroku, wywołujemy funkcję `update` na kontrolerze z produktami, w celu zak-

tualizowania widoku tabeli.

5.4.3 Implementacja mechanizmów bezpieczeństwa

Z powodu braku zabezpieczeń bazy SQLite zdecydowaliśmy się stworzyć system logowania dla użytkowników aplikacji. Niestety funkcjonalności nie udało nam się w porę zaimplementować.

6 Podsumowanie i wnioski

Realizacja projektu pozwoliła na zdobycie cennych umiejętności pracy z bazami danych. Mogliśmy poznać nowe, nieznane dotychczas technologie i wykorzystać je przy pisaniu aplikacji, która byłaby w stanie funkcjonować w prawdziwych warunkach produkcyjnych.

Brak czasu w trakcie realizacji projektu sprawił, że nie udało się zaimplementować wszystkich funkcjonalności, które chcielibyśmy w takiej aplikacji widzieć.

Ponadto, nie wszystkie decyzje projektowe ostatecznie okazały się trafionymi pomysłami. W celu integracji bazy danych z aplikacją mogliśmy wykorzystać system *ORM* (jak na przykład Hibernate), który oszczędził by sporo pracy przy pisaniu kodu pobierającego dane z bazy danych.

Nietrafionym pomysłem było również wykorzystanie bazy danych SQLite. Brak wsparcia dla zarządzania użytkownikami oraz mechanizmów zabezpieczeń powoduje spore utrudnienia w utrzymaniu i zabezpieczeniu systemu.

Błędem projektowym okazał się również brak dodatkowych mechanizmów w bazie (na przykład triggerów), które ułatwiłyby zarządzanie danymi na poziomie kodu źródłowego aplikacji. Ponadto mogliśmy wykorzystać więcej widoków, w celu ułatwienia dostępu do bazy i zapewnienia lepszych możliwości rozbudowy systemu na przyszłość.

Trudnością w czasie realizacji aplikacji dostępowej okazała się JavaFX, która znacznie spowolniła pracę, głównie przez nasz brak umiejętności w zakresie budowania aplikacji z interfejsem graficznym.

Podsumowując. Projekt okazał się dość ciężki w realizacji, co jednak zaowocowało lepszym zrozumieniem tematu i daniem możliwości unikania podobnych błędów na przyszłość.