Temat: Sklep z odzieżą

1 etap - Specyfikacja wymagań

Autorzy:
Mateusz Błach 264010, Michał Lewandowski 264458, Jakub Jakubowicz 263925

Wstęp:

Nasz projekt zakłada stworzenie aplikacji bazodanowej, która usprawni zarządzanie sklepem z odzieżą online. Aplikacja ta ma na celu ułatwić klientom przeglądanie i zakup produktów, a jednocześnie umożliwić administratorom i pracownikom sprawną obsługę sklepu. Aplikacja zostanie podzielona na trzy główne grupy użytkowników: administratorów, pracowników i klientów, z każdą grupą posiadającą dostęp do określonych funkcjonalności. Dla administratorów, głównym celem będzie efektywne zarządzanie produktami, zamówieniami oraz kontami klientów. Pracownicy będą odpowiedzialni za akceptację i anulację zamówień, a klienci będą mogli wygodnie przeglądać i zamawiać produkty.

Wymagania funkcjonalne:

Dla administratora:

1. **Logowanie:** Administrator ma możliwość zalogowania się do systemu za pomocą unikalnego identyfikatora (login) i hasła
2. **Zarządzanie produktami:** Administrator ma dostęp do funkcji dodawania, edytowania i usuwania produktów w bazie danych.
3. **Zarządzanie zamówieniami:** Administrator powinien być w stanie przeglądać, edytować i usuwać.
4. **Zarządzanie kontami:** Administrator ma prawo tworzyć, edytować i usuwać konta klientów sklepu.
5. **Backup i odzyskiwanie danych:** Administrator ma możliwość stworzenia backupu stanu produktów, użytkowników oraz zamówień.

Dla pracownika:

1. **Przeglądanie produktów:** Pracownik może przeglądać produkty w bazie, sprawdzać dostępność, filtrowanie produktów po różnych parametrach.
2. **Logowanie:** Pracownik ma możliwość zalogowania się do systemu za pomocą unikalnego identyfikatora (login) i hasła.
3. **Akceptacja zamówienia:** Każde zamówienie złożone przez klienta musi zostać zaakceptowane przez pracownika.
4. **Anulowanie zamówienia:** Każde zamówienie złożone przez klienta może zostać anulowane przez pracownika.

Dla klienta:

1. **Zmiana danych osobowych:** Klienci mogą usunąć swoje konto z bazy oraz edytować swoje dane.
2. **Logowanie:** Klienci mogą logować się za pomocą loginu i hasła.
3. **Rejestracja:** Klienci mogą stworzyć konto.
4. **Przeglądanie produktów:** Klienci mogą przeglądać produkty według kategorii, cen i marki.
5. **Dodawanie do zamówienia:** Klienci mogą dodawać produkty do zamówienia.
6. **Składanie zamówień:** Klienci mogą składać zamówienia podając dane dostawy i płatności.
7. **Śledzenie zamówień:** Klienci mogą sprawdzać status swoich zamówień i historię zakupów.

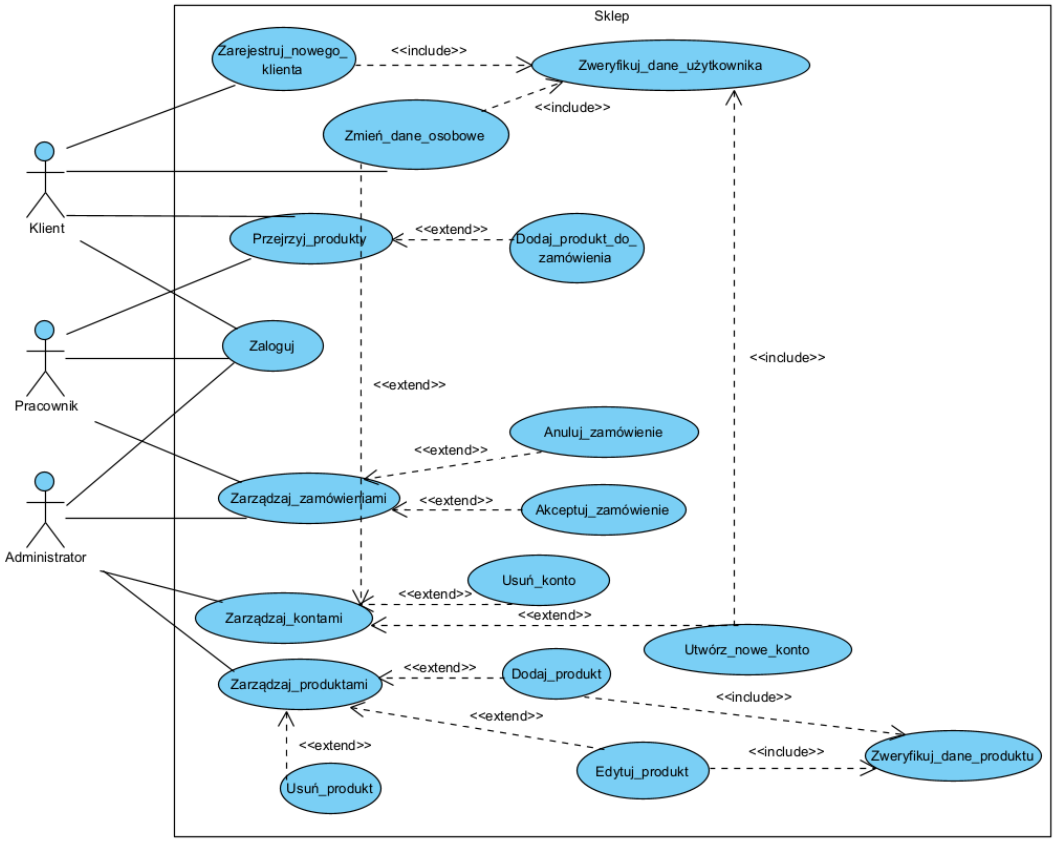
Dla wszystkich użytkowników:

1. **Weryfikacja danych logowania:** Weryfikacja czy konto istnieje w bazie danych dla podanych danych logowania.
2. **Weryfikacja wprowadzonych danych:** Weryfikacja wprowadzonych danych.

Wymagania niefunkcjonalne:

1. **Autoryzacja operacji:** Każda operacja na bazie danych jest dostępna w zależności od typu konta użytkownika.
2. **Walidacja wprowadzonych danych:** Weryfikacja czy podane dane przez użytkownika są prawidłowe.
3. **Wykorzystanie procedur**: Wszystkie zapytania kierowane z aplikacji do bazy danych będą wykonywane poprzez procedury.
4. **Hashowanie haseł:** Zastosowanie funkcji haszujących do bezpiecznego przechowywania haseł w bazie danych.
5. **Jedno konto administratora:** Istnienie jednego wbudowanego konta administratora w bazie danych
6. **Wymaganie logowania:** Konieczność logowania się dla klientów korzystających z aplikacji bazodanowej
7. **Akceptacja zamówień**: Każde zamówienie musi zostać zaakceptowane przez odpowiedniego pracownika przed dalszym przetwarzaniem.

Diagram przypadków użycia:



Scenariusze przypadków użycia:

Nazwa PU: Zarejestruj_nowego_klienta
Cel: Zarejestrowanie nowego klienta
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej

Warunki końcowe: Wprowadzenie unikalnej nazwy użytkownika

Scenariusz:

1. **Należy podać imię, nazwisko, adres, nazwę użytkownika i hasło.**
2. **Należy sprawdzić czy dany użytkownik już nie istnieje i czy hasło jest odpowiednio skonstruowane poprzez PU Zweryfikuj_wprowadzone_dane**
3. **Jeżeli wprowadzone dane nie są prawidłowe należy zakończyć PU, w przeciwnym wypadku należy zapisać dane w bazie danych.**

Nazwa PU: Zweryfikuj_dane_użytkownika
Cel: Zweryfikowanie poprawności wprowadzonych danych

Warunki początkowe: Jest uruchamiany z następujących PU: Zarejestruj_nowego_klienta, Utwórz_nowe_konto, Zmień_dane_osobowe
Warunki końcowe: Zwraca wynik określający czy wprowadzone dane są prawidłowe
Scenariusz:

1. **Porównuje nazwę użytkownika z nazwami użytkownika istniejącymi już w bazie danych.**
2. **W przypadku znalezienia użytkownika o podanej nazwie zwraca te nazwę użytkownika i kończy przeszukiwanie bazy danych.**
3. **W przypadku nieznalezienia podanej nazwy użytkownika zwraca wynik negatywny**

Nazwa PU: Zmień_dane_osobowe
Cel: Zmiana danych osobowych klienta
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej

Warunki końcowe: Wprowadzona nazwa użytkownika musi być unikalna

Scenariusz:

1. **Należy podać dane, które chcemy zmienić**
2. **Jeżeli wśród zmienionych danych występuje nazwa użytkownika to uruchamia PU Zwerfikuj_dane_użytkownika.**
3. **Jeżeli wprowadzone dane nie są prawidłowe należy zakończyć PU, w przeciwnym wypadku należy zapisać dane w bazie danych.**

Nazwa PU: Przejrzyj_produkty
Cel: Przegląd dostępnych produktów
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej
Warunki końcowe: Kryteria przeglądania muszą być poprawne
Scenariusz:

1. **Użytkownik aplikacji podaje określone kryteria wedle których aplikacja filtruje produkty.**
2. **Zgodnie z określonymi kryteriami zwracana jest lista produktów.**

Nazwa PU: Dodaj_produkt_do_zamówienia
Cel: Dodanie wybranego produktu do zamówienia
Warunki początkowe: Jest uruchamiany z PU Przejrzyj_produkty
Warunki końcowe: Wybrany produkt musi być dostępny
Scenariusz:

1. **Użytkownik wybiera produkt**
2. **Jeżeli produkt jest dostępny to dodaje go do zamówienia, w przeciwnym wypadku kończy PU**

Nazwa PU: Zaloguj
Cel: Zalogowanie użytkownika
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej
Warunki końcowe: Użytkownik musi podać prawidłowe dane logowania

Scenariusz:

1. **Należy podać nazwę użytkownika oraz hasło**
2. **Sprawdzana jest poprawność wprowadzonych danych**
3. **Jeżeli podano prawidłowe dane logowania to użytkownik ma dostęp do aplikacji, w przeciwnym wypadku kończymy PU**

Nazwa PU: Zarządzaj_zamówieniami
Cel: Zarządzanie zamówieniami
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej
Warunki końcowe: Użytkownik musi być administratorem lub pracownikiem
Scenariusz:

1. **Użytkownik wybiera zamówienie, które może zaakceptować korzystając z PU Akcpetuj_zamówienie lub anulować korzystając z PU Anuluj_zamówienie**
2. **W przypadku edycji zamówienia należy zapisać zmiany w bazie danych, w innym przypadku należy zakończyć PU**

Nazwa PU: Anuluj_zamówienie
Cel: Anulowanie zamówienia
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_zamówieniami
Warunki końcowe: Zamówienie musi znajdować się w bazie
Scenariusz:

1. **W przypadku anulowania zamówienia jest ono usuwane z bazy danych**
2. **W innym wypadku należy zakończyć PU**

Nazwa PU: Akceptuj_zamówienie
Cel: Akceptacja zamówienia
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_zamówieniami
Warunki końcowe: Zamówienie musi istnieć
Scenariusz:

1. **W przypadku akceptacji zamówienia jest ono dodawane do bazy danych**
2. **W innym wypadku należy zakończyć PU**

Nazwa PU: Zarządzaj_kontami
Cel: Zarządzanie kontami użytkowników
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej
Warunki końcowe: Użytkownik musi być administratorem
Scenariusz:

1. **Użytkownik wybiera konto i może edytować dane osobowe korzystając z PU Zmień_dane_osobowe lub usunać konto korzystając z PU Usuń_konto**
2. **W przypadku, kiedy konto nie istnieje użytkownik może je utworzyć korzystając z PU Utwórz_nowe_konto**
3. **W wypadku edycji/utworzenia/usunięcia konta należy zapisać zmiany w bazie danych w innym przypadku należy zakończyć PU**

Nazwa PU: Usuń_konto
Cel: Usunięcie konta użytkownika
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_kontami
Warunki końcowe: Wybrane konto musi znajdować się w bazie
Scenariusz:

1. **W przypadku usunięcia konta jest ono usuwane z bazy danych**

2. **W przeciwnym wypadku należy zakończyć PU**

Nazwa PU: Utwórz_nowe_konto
Cel: Utworzenia nowego konta użytkownika
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_kontami
Warunki końcowe: Wprowadzone dane do konta muszą być unikalne
Scenariusz:

1. **Należy zweryfikować czy podane dane do konta nie znajdują się już w bazie danych za pomocą PU Zweryfikuj_dane_użytkownika**
2. **Jeżeli wprowadzone dane są unikalne należy zapisać nowe konto w bazie danych, w przeciwnym wypadku należy zakończyć PU**

Nazwa PU: Zarządzaj_produktami
Cel: Zarządzanie produktami
Warunki początkowe: Uruchomienie programu aplikacji bazodanowej
Warunki końcowe: Użytkownik musi być administratorem
Scenariusz:

1. **Użytkownik wybiera produkt, który może edytować korzystając z PU Edytuj_produkt lub usunąć korzystając z PU Usuń_produkt**
2. **Jeżeli produkt nie istnieje użytkownik może go utworzyć korzystając z PU Dodaj_produkt**
3. **W przypadku edycji/utworzenia/usunięcia produktu należy zapisać zmiany w bazie danych, w przeciwnym wypadku należy zakończyć PU**

Nazwa PU: Dodaj_produkt
Cel: Dodawanie produktu
Warunki początkowe: Jest uruchamiany z PU Zarządzaj produktami
Warunki końcowe: Wprowadzone dane produktu muszą być unikalne
Scenariusz:

1. **Należy zweryfikować czy wprowadzone dane produktu są unikalne za pomocą PU Zweryfikuj_dane_produktu**
2. **Jeżeli wprowadzone dane produktu są unikalne i poprawne następuje dodanie produktu do bazy danych**
3. **W innym przypadku należy zakończyć PU**

Nazwa PU: Zweryfikuj_dane_produktu
Cel: Weryfikacja poprawności danych produktu
Warunki początkowe: Jest uruchamiany z PU Dodaj_produkt lub PU Edytuj_produkt
Warunki końcowe: Zwraca wynik określający czy wprowadzone dane są poprawne lub nie
Scenariusz:

1. **Porównuje dane produktu z danymi produktów znajdujących się już w bazie danych**

2. **W przypadku znalezienia produktu o podanych danych zwracany jest wynik pozytywny i następuje koniec przeszukiwania bazy danych**
3. **W innym przypadku zwracany jest wynik negatywny**

Nazwa PU: Usuń_produkt
Cel: Usunięcie produktu
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_produktami
Warunki końcowe: Wybrany produkt musi znajdować się w bazie
Scenariusz:

1. **W przypadku usunięcia produkt jest usuwany z bazy danych**
2. **W innym wypadku należy zakończyć PU**

Nazwa PU: Edytuj_produkt
Cel: Edycja produktu
Warunki początkowe: Jest uruchamiany z PU Zarządzaj_produktami
Warunki końcowe: Wprowadzone dane produktu muszą być unikalne
Scenariusz:

1. **Należy zweryfikować czy wprowadzone dane produktu nie znajdują się już w bazie danych za pomocą PU Zweryfikuj_dane_produktu**
2. **Jeżeli dane produktu nie pokrywają się z danymi z bazy danych należy zaktualizować produkt w bazie danych, w innym wypadku należy zakończyć PU**

2 etap - Projekt, implementacja i testy bazy danych

1. Identyfikacja encji.

   Tabela Users:

   - user_id
   - login
   - password
   - email
   - phone
   - name
   - surname
   - role

   Tabela Orders:

   - order_id
   - user_id
   - clothes_id
   - date
   - amount
   - delivery_id
   - payment_id

   Tabela Clothes:

   - clothes_id
   - material
   - size
   - sex
   - price
   - collection_id

   Tabela Collections:

   - collection_id
   - name
   - start_date
   - end_date

   Tabela Delivery:

   - delivery_id
   - city
   - street

- number
- postal_code
- country

Tabela Payments:

- payment_id
- status
- payment_form
- date

2. Sformułowanie wymagań dotyczących dostępu do bazy i jej zawartości.

   Użytkownik z bazy danych może:

   - Pobierać informację znajdując się w bazie danych o swoim koncie, poza jego user_id

     SELECT login, name, surname, role, email, phone FROM Users WHERE user_id = :user_id;

   - Modyfikować niektóre informacje o swoim koncie (użytkownik nie może modyfikować user_id, login, name, surname, role):

     UPDATE Users SET email = :email, phone = :phone WHERE user_id = :user_id;

   - Pobierać dostępne w tabeli Clothes ubrania i filtrować wyszukiwania po wszystkich jej atrybutach oraz filtrować po atrybutach z tabeli Collections

     ```
     SELECT * FROM Clothes WHERE collection_id IN (
        SELECT collection_id FROM Collections WHERE start_date <= CURRENT_DATE
     AND end_date >= CURRENT_DATE
     );

     SELECT * FROM Clothes
     WHERE (:material IS NULL OR material = :material)
       AND (:size IS NULL OR size = :size)
       AND (:sex IS NULL OR sex = :sex)
       AND (:price IS NULL OR price <= :max_price);
     ```

   - Tworzyć zamówienie:
     INSERT INTO Orders (user_id, clothes_id, date, amount, delivery_id, payment_id)
     VALUES (:user_id, :clothes_id, :date, :amount, :delivery_id, :payment_id);

   - Tworzyć (dokonywać) płatności za stworzone przez siebie zamówienie:

     INSERT INTO Payments (status, payment_form, date, user_id)
     VALUES (:status, :payment_form, CURRENT_DATE, :user_id);

   - Pobierać informacje o płatności dokonanej przez swoje konto:

     SELECT status, payment_form, date FROM Payments WHERE payment_id = :payment_id;

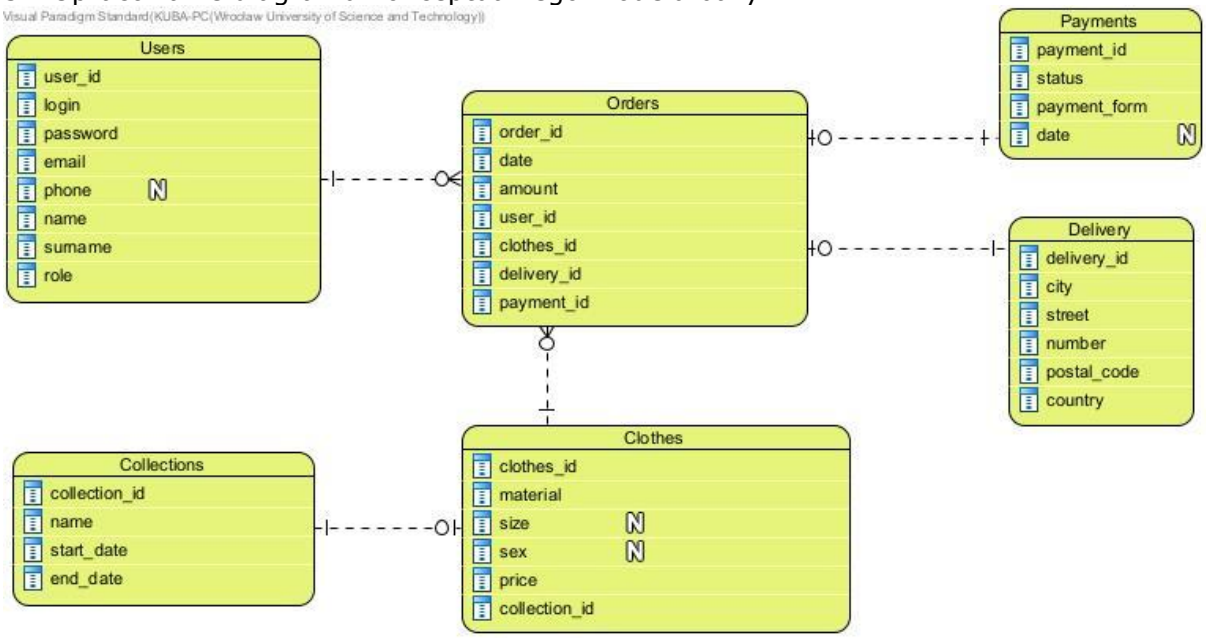
- Tworzyć adres dostawy dla zamówienia:

  INSERT INTO Delivery (city, street, number, postal_code, country)
  VALUES (:city, :street, :number, :postal_code, :country);

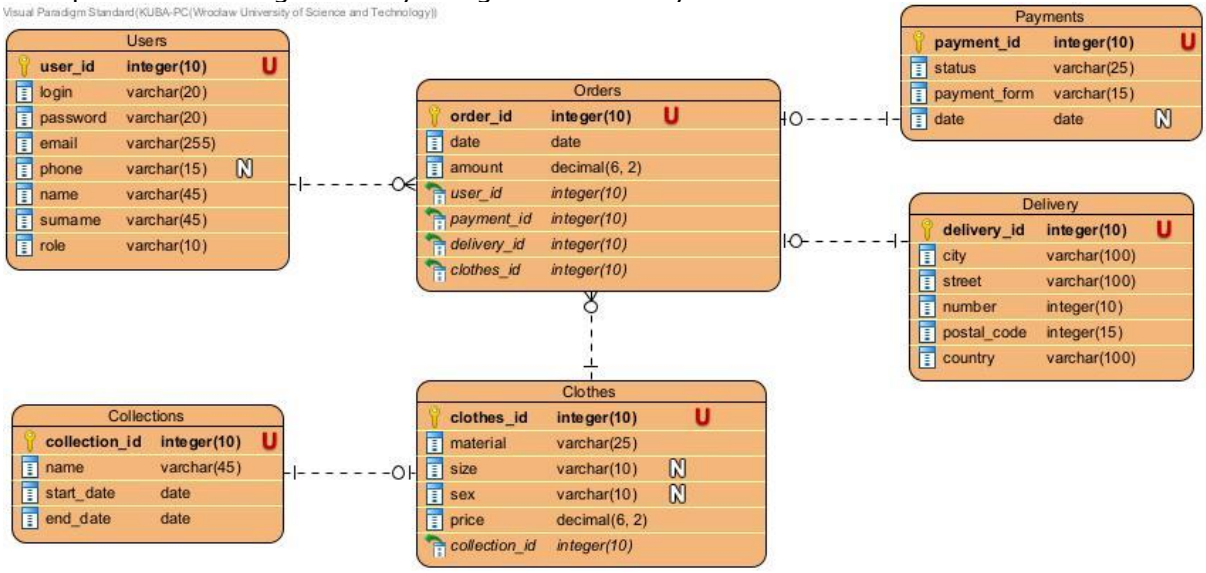
- Modyfikować adres dostawy swoich zamówień:

  UPDATE Delivery SET city = :city, street = :street, number = :number, postal_code = :postal_code, country = :country
  WHERE delivery_id = :delivery_id;

3. Opracowanie diagramu konceptualnego modelu bazy.

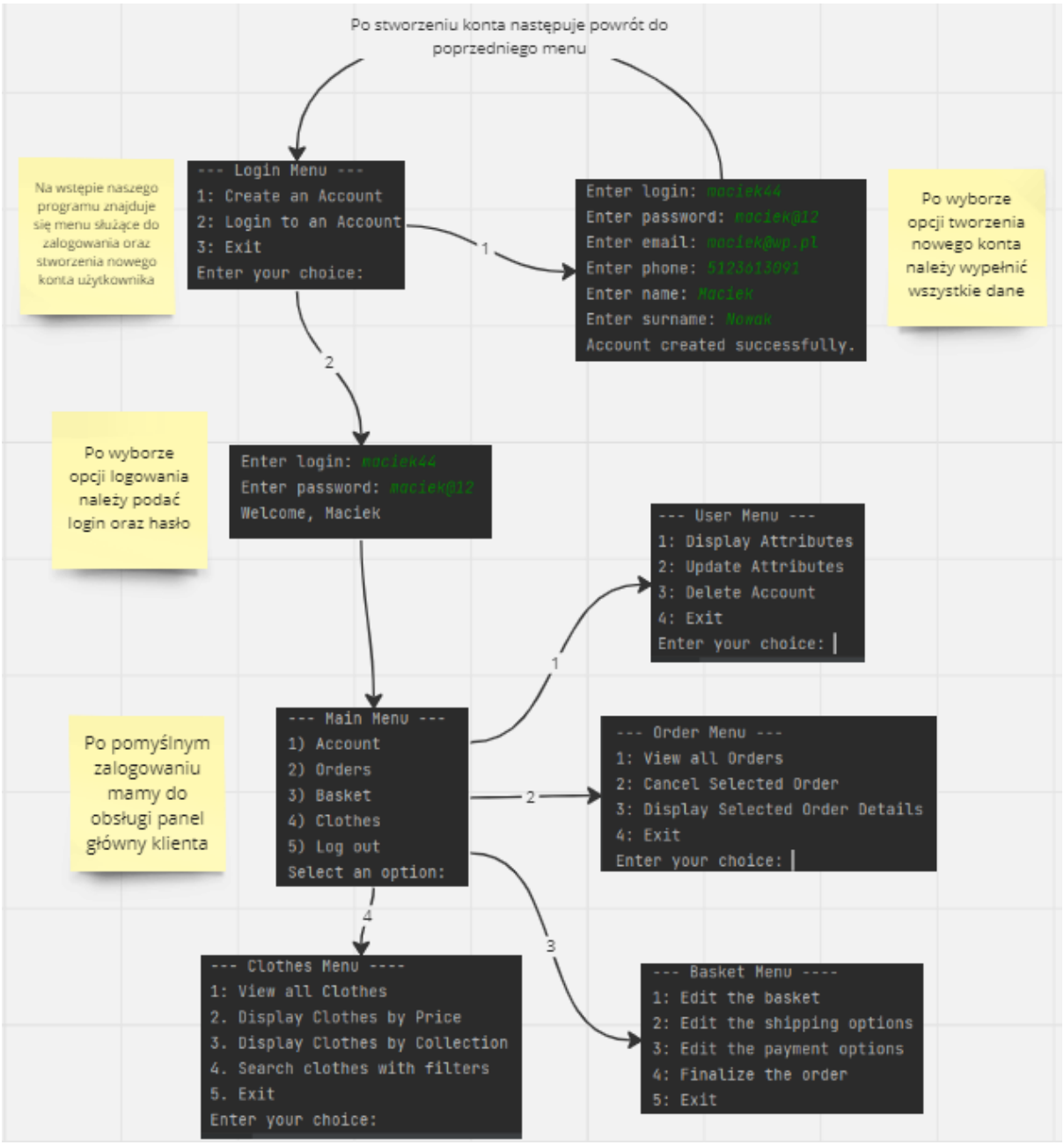


4. Opracowanie diagramu fizycznego modelu bazy

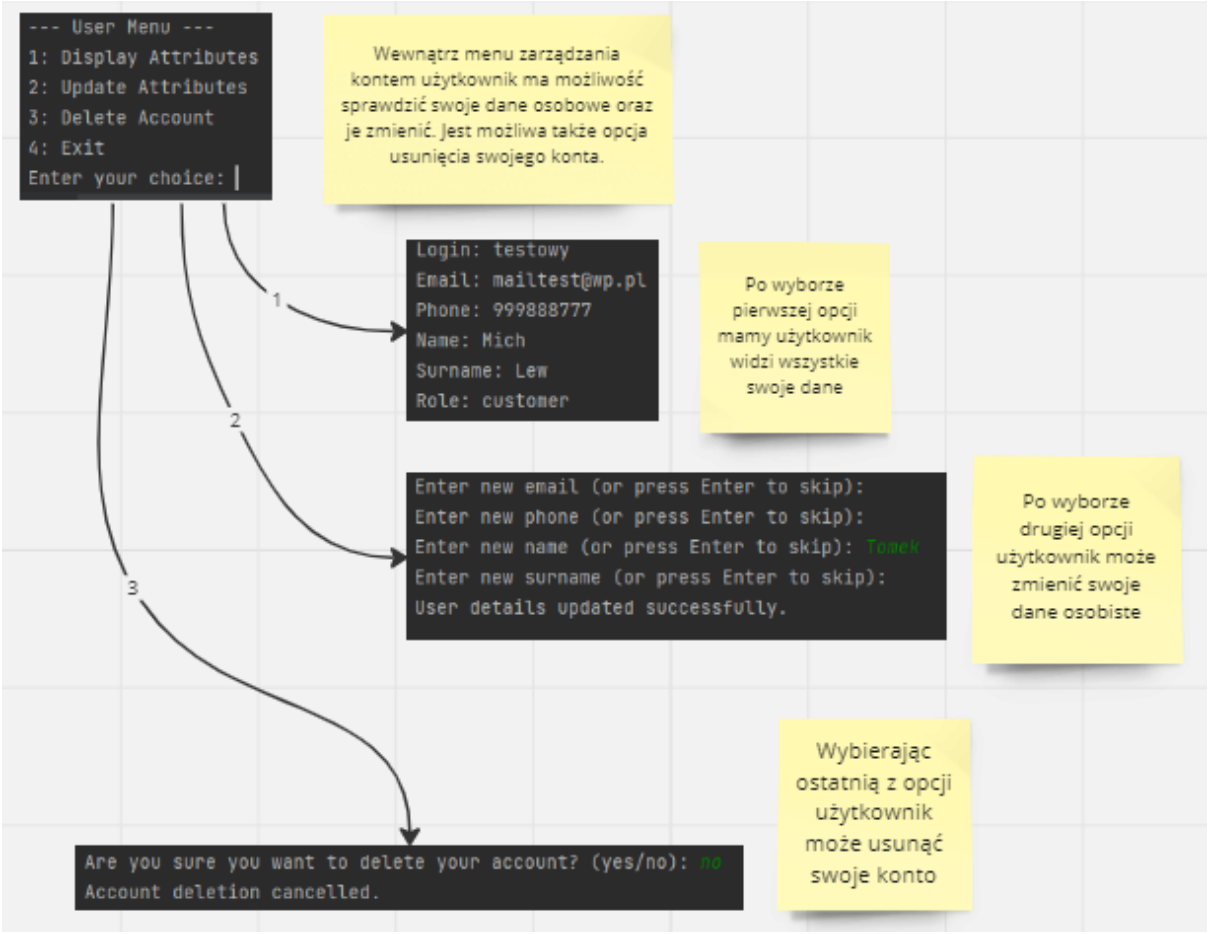


3 etap – Projekt, implementacja i testy aplikacji bazodanowej

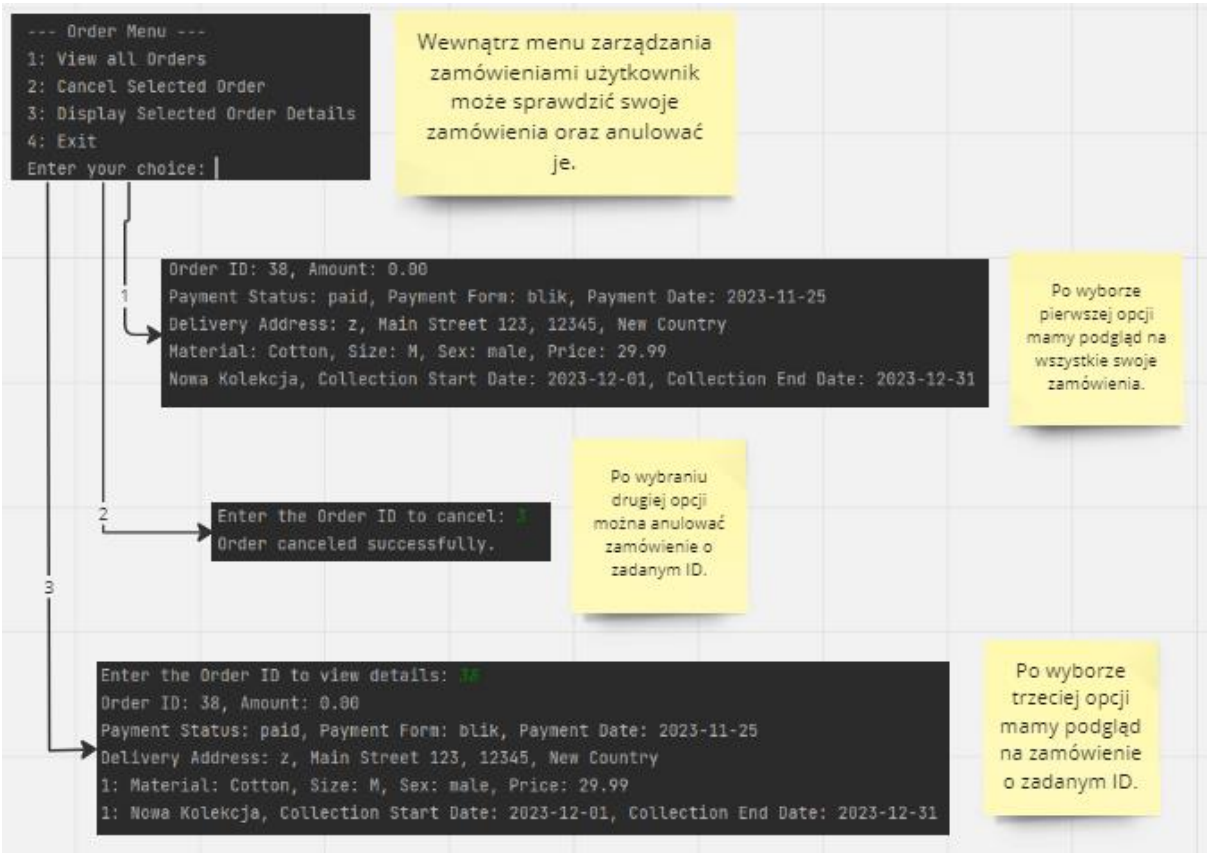
1. Makieta interfejsu graficznego aplikacji
   1.1. Główne Menu



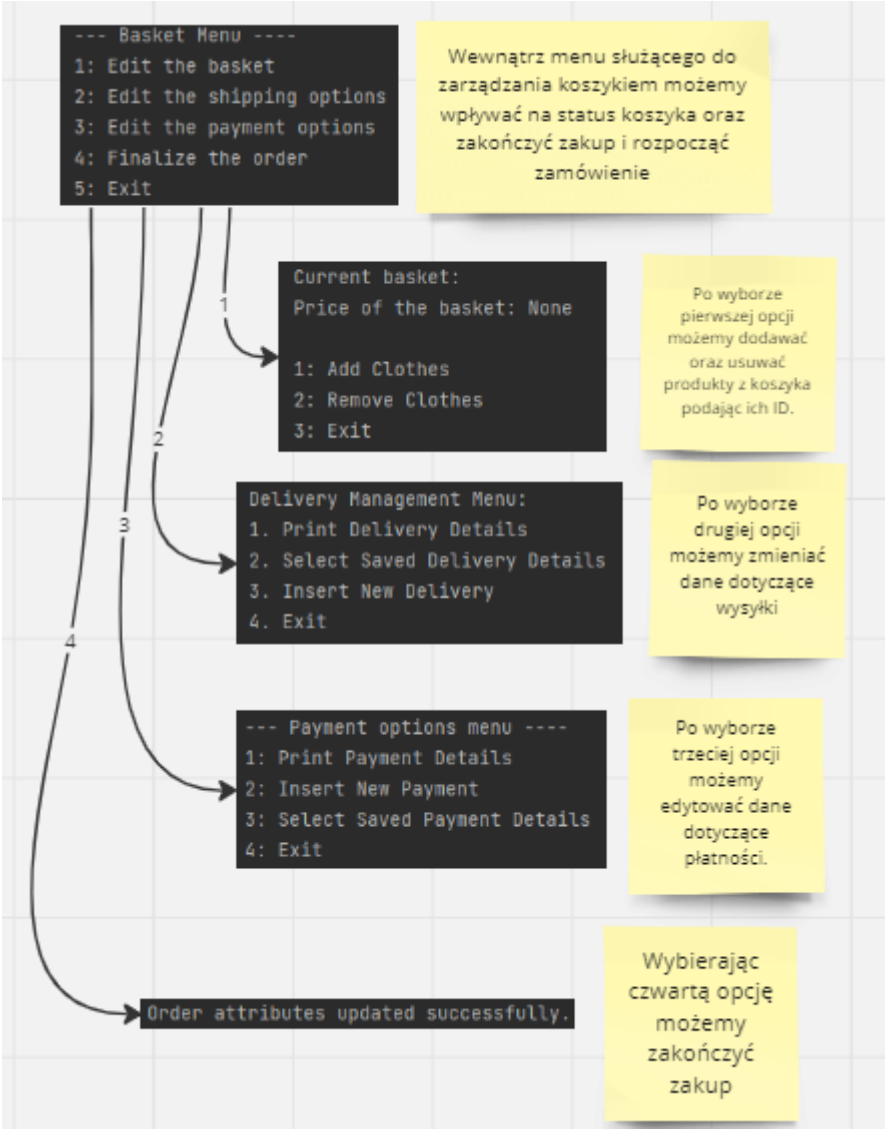
1.2. Zarządzanie kontem



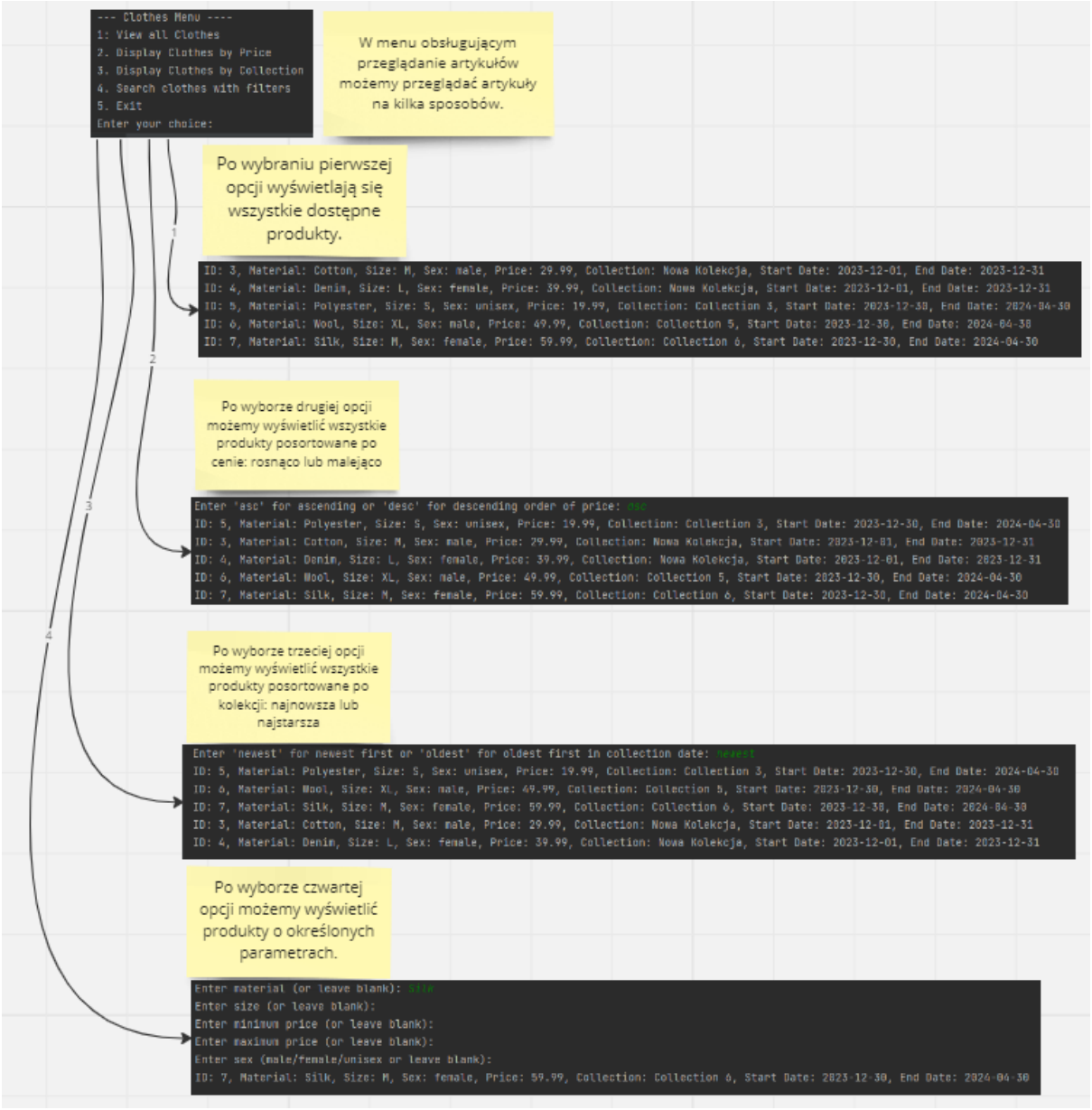
1.3. Zarządzanie zamówieniami



1.4. Zarządzanie koszykiem

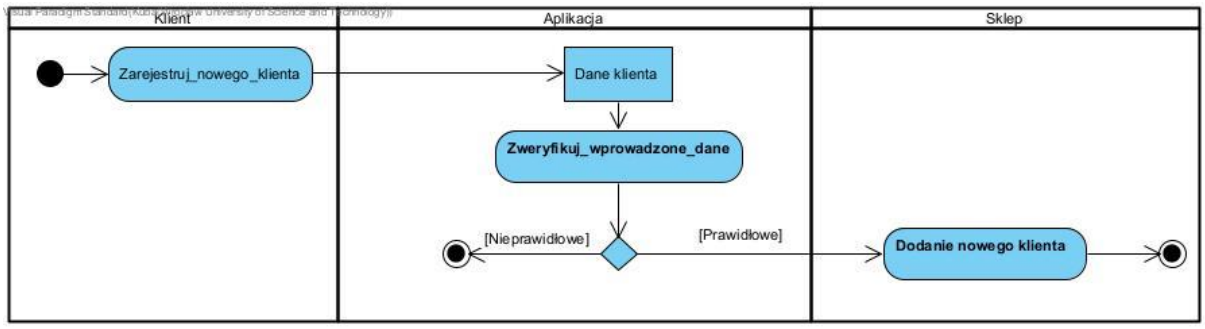


1.5. Przeglądanie artykułów

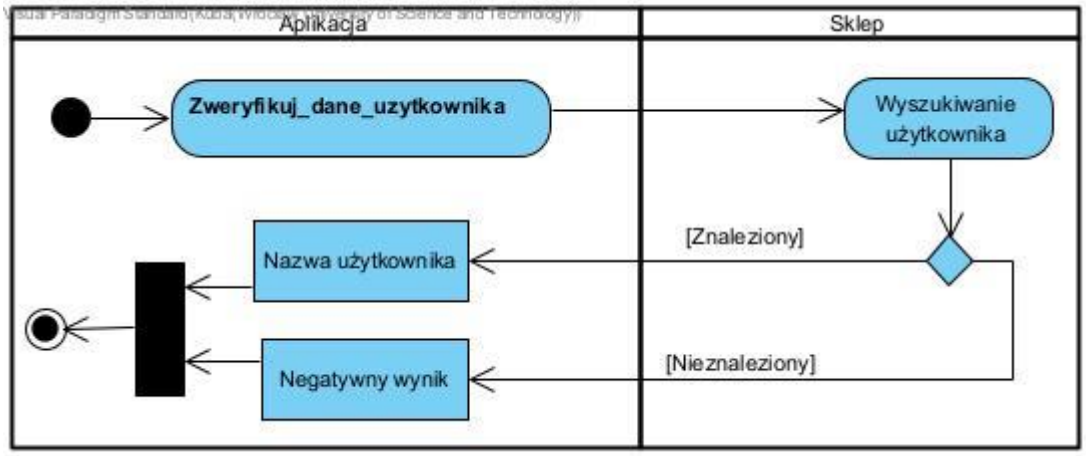


2. Diagramy czynności

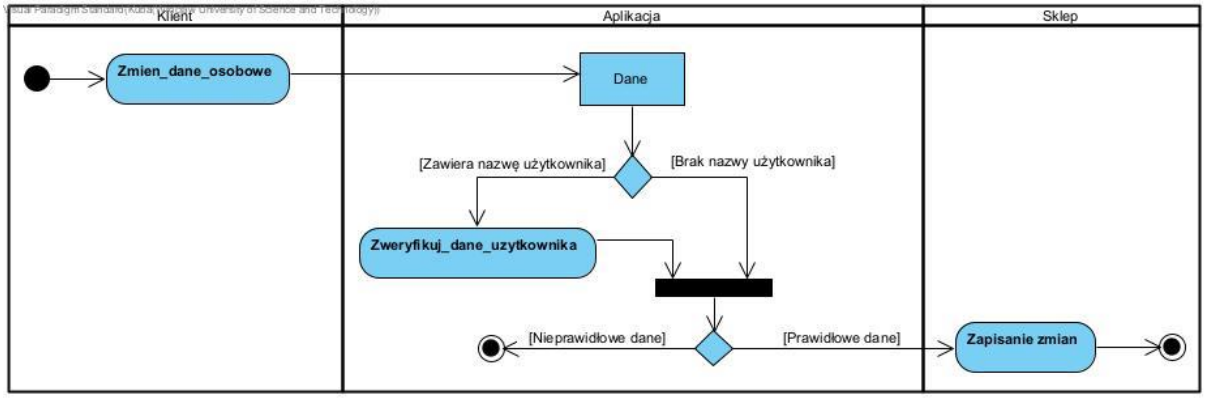
2.1 Rejestracja nowego klienta



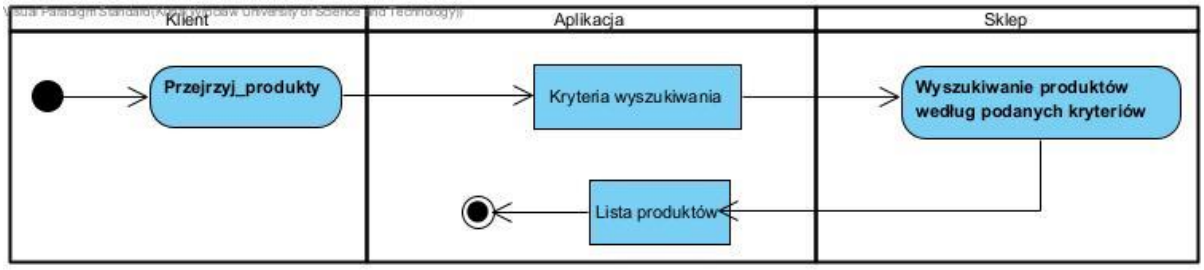
2.2 Weryfikacja wprowadzonych danych



2.3 Zmiana danych osobowych

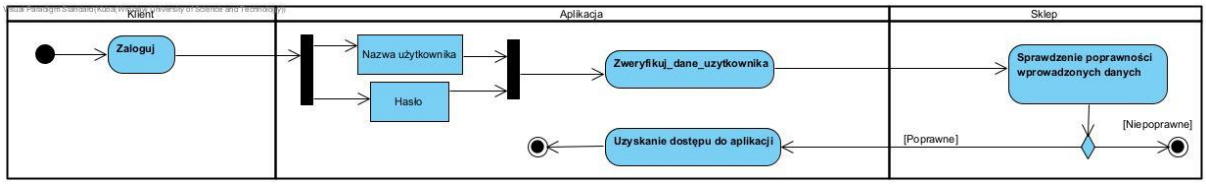


2.4 Przeglądanie produktów według podanych kryteriów

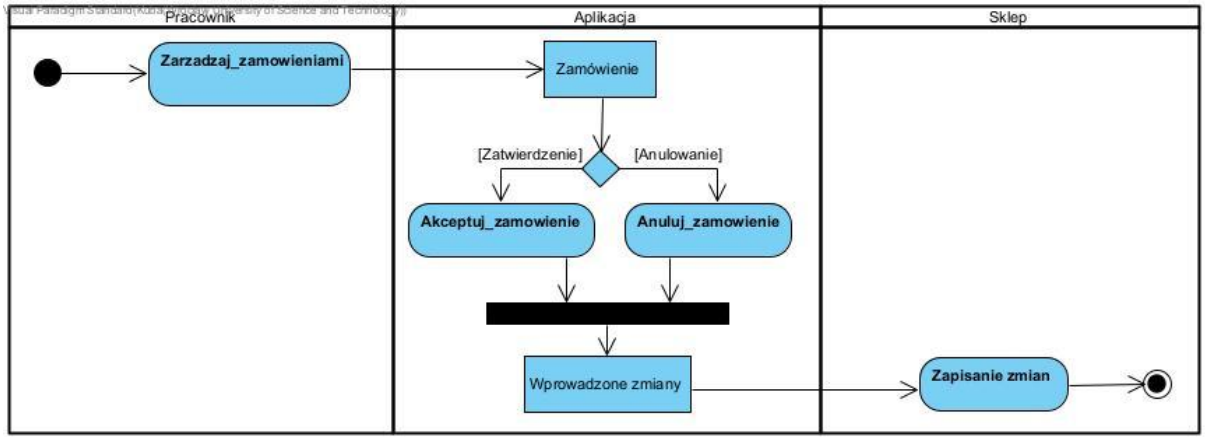


2.5 Dodawanie produktu do zamówienia

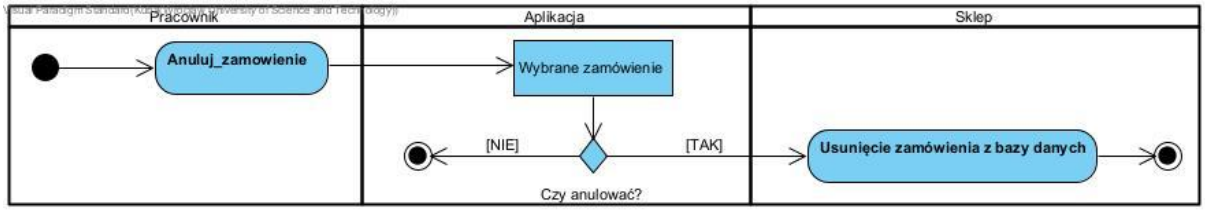
2.6 Logowanie do aplikacji



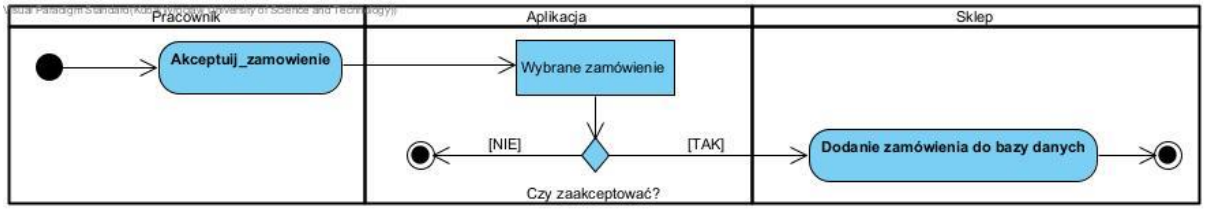
2.7 Zarządzanie zamówieniami



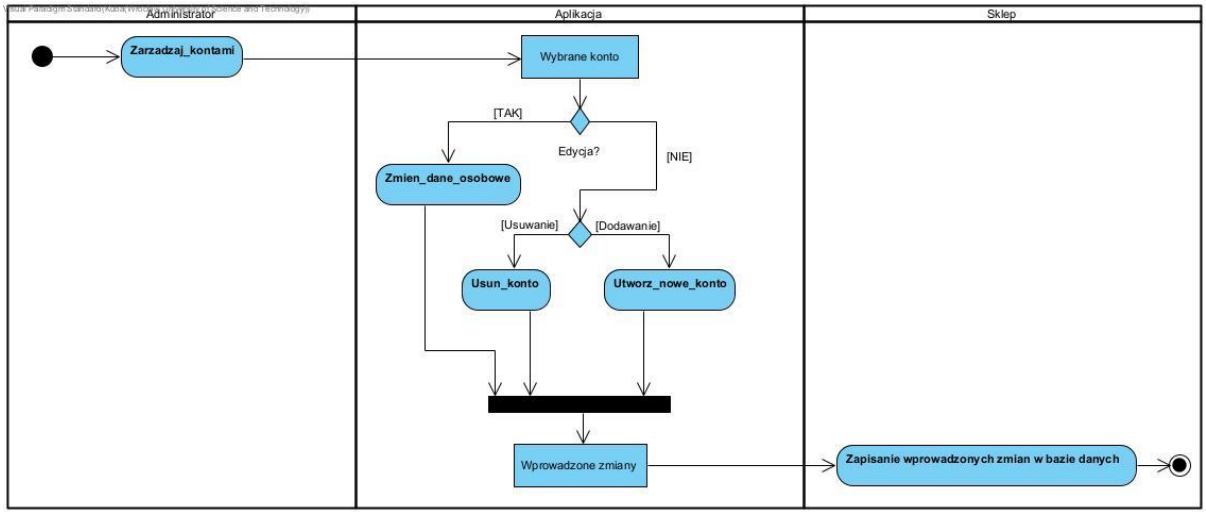
2.8 Anulowanie zamówienia



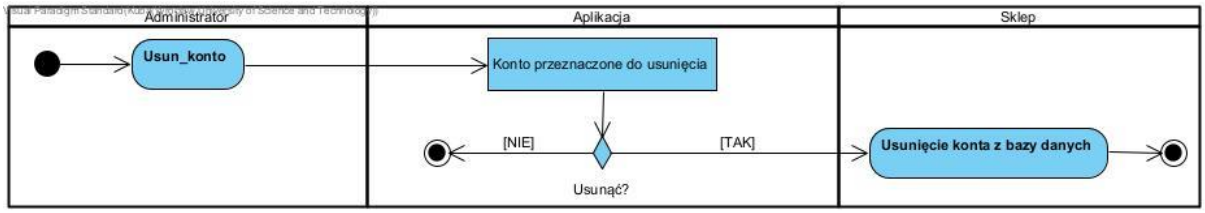
2.9 Akceptacja zamówienia



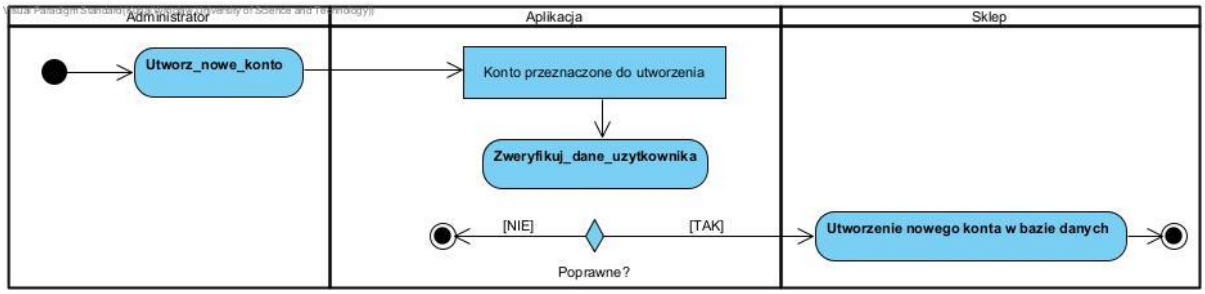
2.10 Zarządzanie kontami



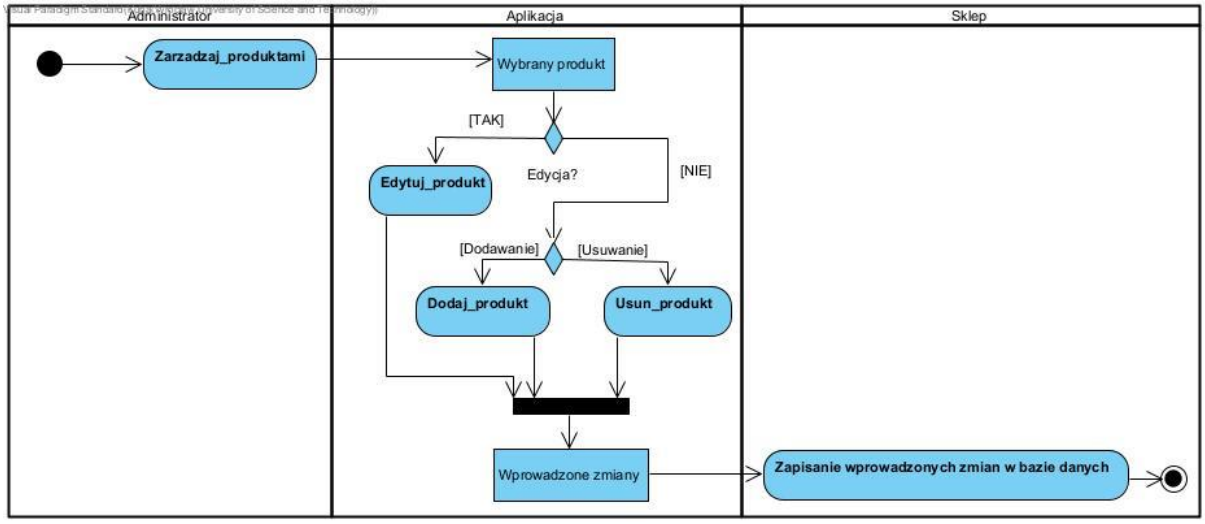
2.11 Usuwanie konta



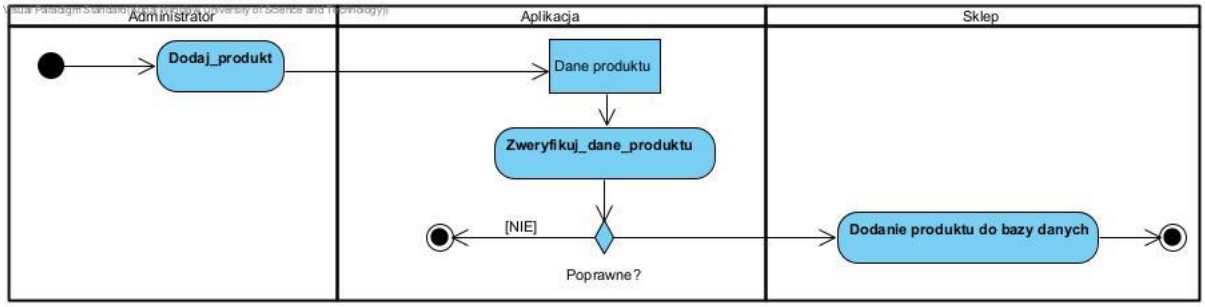
2.12 Tworzenie nowego konta



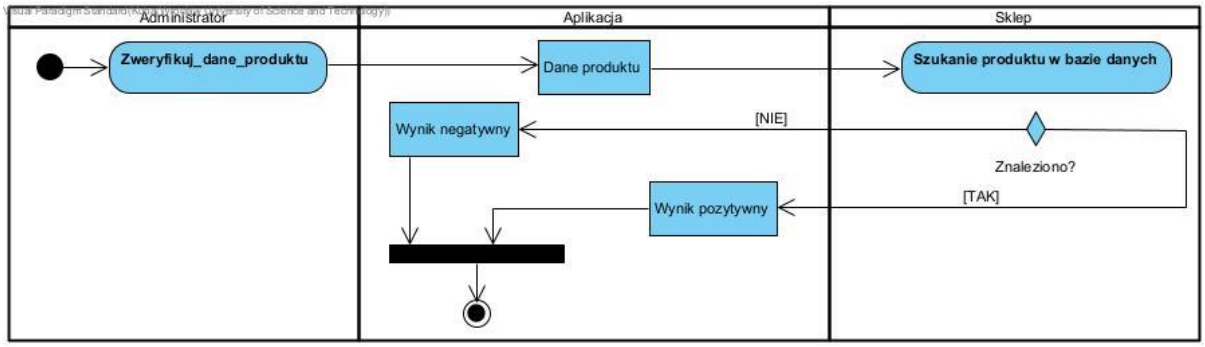
2.13 Zarządzanie produktami



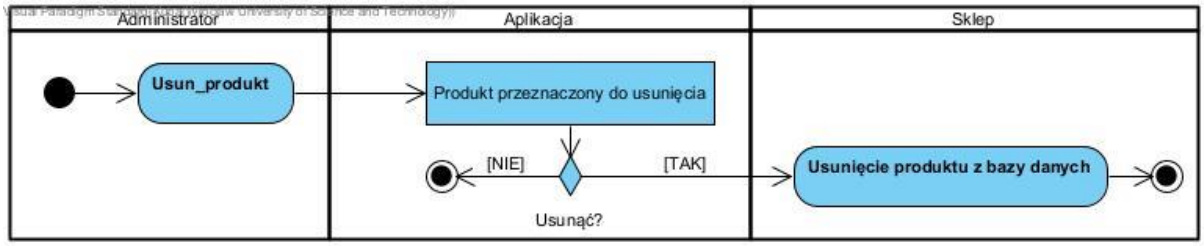
2.14 Dodawanie nowego produktu



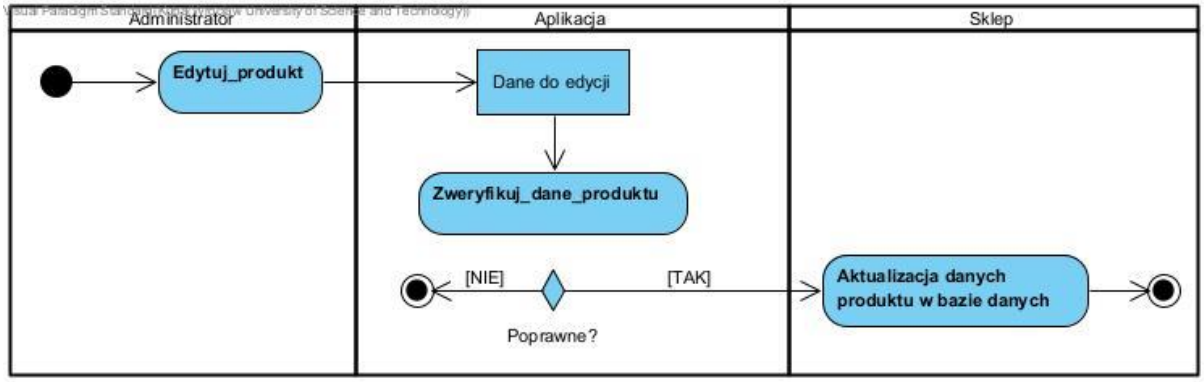
2.15 Weryfikacja danych produktu



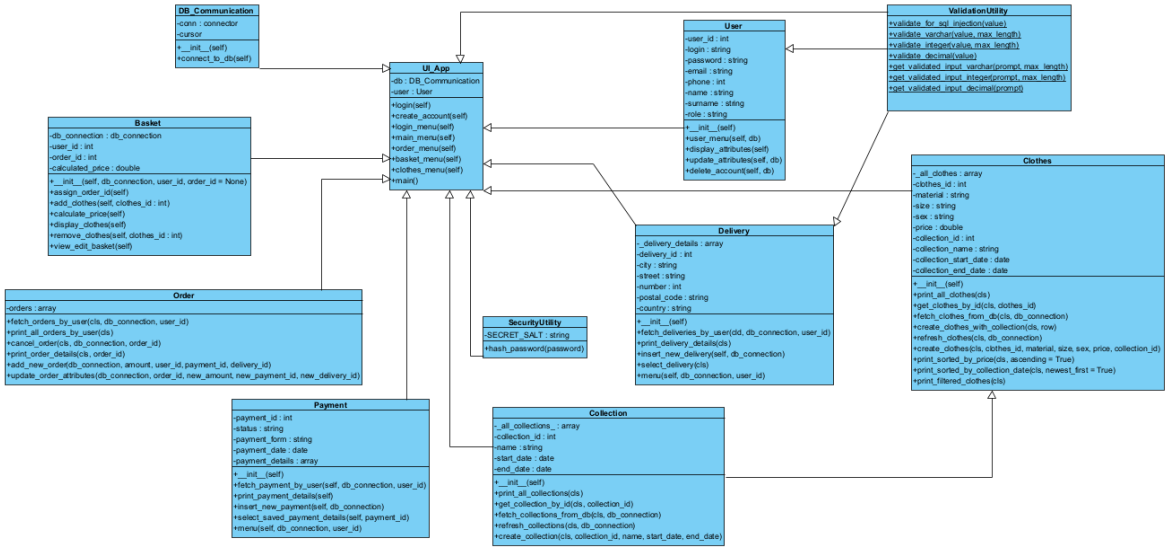
2.16 Usuwanie produktu



2.17 Edycja danych produktu



3. Diagram klas



4. Wykonanie i wdrożenie aplikacji

Aplikacja została wykonana i wdrożona w języku Pyhon. Poniżej zostały przedstawione i omówione kody klas, z których się składa:

- **Basket.py** – klasa reprezentuje koszyk zakupowy;
- **Clothes.py** – klasa reprezentuje ubrania;
- **Collection.py** – klasa reprezentuje kolekcje ubrań;
- **Db_communication.py** - klasa odpowiada za komunikację z bazą danych MySQL;
- **Delivery.py** – klasa obsługuje informacje dotyczące dostaw zamówień;
- **Order.py** – Klasa odpowiada za zarządzanie zamówieniami użytkowników;
- **Payment.py** – Klasa odpowiada za zarządzanie płatnościami;
- **Security_utility.py** – Klasa odpowiedzialna za zabezpieczenie hasła;
- **Ui_app.py** – Klasa implementuje interfejs odpowiedzialny za komunikację z użytkownikiem;
- **User.py** – Klasa reprezentuje użytkownika w systemie sklepu;
- **Validation_utility.py** – Klasa zawiera zestaw narzędzi do walidacji danych wejściowych.

- **Basket.py**:

```python
class Basket:

    def __init__(self, db_connection, user_id, order_id=None):

        self.db_connection = db_connection
        db_connection.cursor.callproc("addDefaultOrder", [user_id, 0])
        self.db_connection.conn.commit()
        db_connection.cursor.execute("SELECT MAX(order_id) FROM Orders")
        self.order_id = db_connection.cursor.fetchone()[0]
        self.calculated_price = None

    def assign_order_id(self):

        query = "SELECT MAX(order_id) FROM Orders"
        self.db_connection.cursor.execute(query)
        result = self.db_connection.cursor.fetchone()
        latest_order_id = result[0] if result[0] is not None else 0
        return latest_order_id + 1

    def add_clothes(self, clothes_id: int):
        print(f"order_id {self.order_id}")
        query = "INSERT INTO Basket (order_id, clothes_id) VALUES (%s, %s)"
        self.db_connection.cursor.execute(query, (self.order_id, clothes_id))
        self.db_connection.conn.commit()

    def calculate_price(self):

        query = """
            SELECT SUM(Clothes.price)
            FROM Clothes
            JOIN Basket ON Clothes.clothes_id = Basket.clothes_id
            WHERE Basket.order_id = %s
        """
        self.db_connection.cursor.execute(query, (self.order_id,))
        self.calculated_price = self.db_connection.cursor.fetchone()[0]

    def display_clothes(self):

        query = """
            SELECT Clothes.material, Clothes.size, Clothes.sex, Clothes.price
            FROM Basket
            JOIN Clothes ON Basket.clothes_id = Clothes.clothes_id
            WHERE Basket.order_id = %s
        """
        self.db_connection.cursor.execute(query, (self.order_id,))
        for row in self.db_connection.cursor:
            print(f"Material: {row[0]}, Size: {row[1]}, Sex: {row[2]}, Price: {Decimal(row[3])}")

    def remove_clothes(self, clothes_id: int):
        # Remove the specified clothes from the basket
        query = "DELETE FROM Basket WHERE order_id = %s AND clothes_id = %s"
        self.db_connection.cursor.execute(query, (self.order_id, clothes_id))
        self.db_connection.conn.commit()

    def view_edit_basket(self):
        while True:
            print("\nCurrent basket:")
            self.display_clothes()
            self.calculate_price()
            print(f"Price of the basket: {self.calculated_price}")

            print("\n1: Add Clothes")
            print("2: Remove Clothes")
            print("3: Exit")
            choice = input("Enter your choice: ")

            if choice == '1':
                clothes_id = int(input("Enter Clothes ID to add: "))
                self.add_clothes(clothes_id)
            elif choice == '2':
                clothes_id = int(input("Enter Clothes ID to remove: "))
                self.remove_clothes(clothes_id)
            elif choice == '3':
                return self.order_id, self.calculated_price
            else:
                print("Invalid choice. Please try again.")
```

*Funkcja **__init__** :*

Inicjalizuje nową instancję koszyka. Ustala połączenie z bazą danych, tworzy nowe zamówienie dla użytkownika, pobiera jego identyfikator i inicjalizuje zmienną `calculated_price` jako `None`.

*Funkcja **assign_order_id** :*

Zwraca najnowszy identyfikator zamówienia z bazy danych, inkrementując go o 1. Wykorzystywane jest to, aby przypisać unikalny identyfikator zamówienia do koszyka.

*Funkcja **add_clothes** :*

Dodaje ubranie o określonym identyfikatorze do koszyka przypisanego bieżącemu zamówieniu.

*Funkcja **calculate_price** :*

Oblicza łączną cenę ubrań w koszyku na podstawie cen ubrań z bazy danych.

*Funkcja **display_clothes** :*

Wyświetla szczegóły dotyczące materiału, rozmiaru, płci i ceny ubrań znajdujących się w koszyku.

*Funkcja **remove_clothes** :*

Usuwa określone ubranie z koszyka przypisanego bieżącemu zamówieniu.

*Funkcja **view_edit_basket** :*

Wyświetla zawartość koszyka w pętli, umożliwia dodawanie lub usuwanie ubrań oraz wyjście z interaktywnego widoku koszyka.

- **Clothes.py**:

```
class Clothes:
    _all_clothes = []

    def __init__(self):
        self.clothes_id = None
        self.material = None
        self.size = None
        self.sex = None
        self.price = None
        self.collection_id = None
        self.collection_name = None
        self.collection_start_date = None
        self.collection_end_date = None
        Clothes._all_clothes.append(self)

    @classmethod
    def print_all_clothes(cls):
        for clothes in cls._all_clothes:
            print(
                f"ID: {clothes.clothes_id}, Material: {clothes.material}, Size: {clothes.size}, "
                f"Sex: {clothes.sex}, Price: {clothes.price}, Collection: {clothes.collection_name}, "
                f"Start Date: {clothes.collection_start_date}, End Date: {clothes.collection_end_date}")

    @classmethod
    def get_clothes_by_id(cls, clothes_id):
        for clothes in cls._all_clothes:
            if clothes.clothes_id == clothes_id:
                return clothes
        return None

    @classmethod
    def fetch_clothes_from_db(cls, db_connection):
        cls._all_clothes.clear()
        query = """
            SELECT c.clothes_id, c.material, c.size, c.sex, c.price, col.name, col.start_date, col.end_date
            FROM Clothes c
            JOIN Collections col ON c.collection_id = col.collection_id
            """
        db_connection.cursor.execute(query)
        rows = db_connection.cursor.fetchall()
        for row in rows:
            cls.create_clothes_with_collection(row)  # Modified to handle collection details

    @classmethod
    def create_clothes_with_collection(cls, row):
        clothes = cls()
        clothes.clothes_id, clothes.material, clothes.size, clothes.sex, clothes.price, \
            clothes.collection_name, clothes.collection_start_date, clothes.collection_end_date = row

    @classmethod
    def refresh_clothes(cls, db_connection):
        cls.fetch_clothes_from_db(db_connection)

    @classmethod
    def create_clothes(cls, clothes_id, material, size, sex, price, collection_id):
        clothes = cls()
        clothes.clothes_id = clothes_id
        clothes.material = material
        clothes.size = size
        clothes.sex = sex
        clothes.price = price
        clothes.collection_id = collection_id
        return clothes

    @classmethod
    def print_sorted_by_price(cls, ascending=True):
        sorted_clothes = sorted(cls._all_clothes, key=lambda x: x.price, reverse=not ascending)
        for clothes in sorted_clothes:
            print(
                f"ID: {clothes.clothes_id}, Material: {clothes.material}, Size: {clothes.size}, "
                f"Sex: {clothes.sex}, Price: {clothes.price}, Collection: {clothes.collection_name}, "
                f"Start Date: {clothes.collection_start_date}, End Date: {clothes.collection_end_date}")

    @classmethod
    def print_sorted_by_collection_date(cls, newest_first=True):
        # Sort based on collection's start_date
        sorted_clothes = sorted(cls._all_clothes, key=lambda x: x.collection_start_date, reverse=newest_first)

        for clothes in sorted_clothes:
            print(
                f"ID: {clothes.clothes_id}, Material: {clothes.material}, Size: {clothes.size}, "
                f"Sex: {clothes.sex}, Price: {clothes.price}, Collection: {clothes.collection_name}, "
                f"Start Date: {clothes.collection_start_date}, End Date: {clothes.collection_end_date}")

    @classmethod
    def print_filtered_clothes(cls):
        material = input("Enter material (or leave blank): ").strip().lower()
        size = input("Enter size (or leave blank): ").strip().lower()
        min_price = input("Enter minimum price (or leave blank): ").strip()
        max_price = input("Enter maximum price (or leave blank): ").strip()
        sex = input("Enter sex (male/female/unisex or leave blank): ").strip().lower()

        # Convert min_price and max_price to float, handle empty inputs
        min_price = float(min_price) if min_price else None
        max_price = float(max_price) if max_price else None

        for clothes in cls._all_clothes:
            if (not material or clothes.material.lower() == material) and \
                    (not size or clothes.size.lower() == size) and \
                    (not sex or clothes.sex.lower() == sex) and \
                    (min_price is None or clothes.price >= min_price) and \
                    (max_price is None or clothes.price <= max_price):
                print(
                    f"ID: {clothes.clothes_id}, Material: {clothes.material}, Size: {clothes.size}, "
                    f"Sex: {clothes.sex}, Price: {clothes.price}, Collection: {clothes.collection_name}, "
                    f"Start Date: {clothes.collection_start_date}, End Date: {clothes.collection_end_date}")
```

Funkcja **__init__** :

Inicjalizuje nowy obiekt klasy **Clothes**. Przypisuje wartości atrybutom, takim jak identyfikator ubrania, materiał, rozmiar, płeć, cena, identyfikator kolekcji oraz daty rozpoczęcia i zakończenia kolekcji. Dodaje utworzone ubranie do listy wszystkich ubrań **_all_clothes**.

Funkcja **print_all_clothes** :

Wyświetla szczegóły wszystkich ubrań przechowywanych w liście **_all_clothes**, takie jak identyfikator, materiał, rozmiar, płeć, cena, nazwa kolekcji oraz daty rozpoczęcia i zakończenia kolekcji.

Funkcja **get_clothes_by_id** :

Znajduje i zwraca ubranie o określonym identyfikatorze.

Funkcja **fetch_clothes_from_db** :

Pobiera informacje o ubraniach z bazy danych i aktualizuje listę **_all_clothes**.

Funkcja **create_clothes_with_collection** :

Tworzy obiekt ubrania z przypisanymi informacjami o kolekcji.

Funkcja **refresh_clothes** :

Aktualizuje listę **_all_clothes**, pobierając najnowsze informacje z bazy danych.

Funkcja **create_clothes** :

Tworzy i zwraca ubranie z podanymi danymi.

Funkcja **print_sorted_by_price** :

Wyświetla posortowane ubrania według ceny, z możliwością ustalenia rosnącego lub malejącego porządku.

Funkcja **print_sorted_by_collection_date** :

Wyświetla posortowane ubrania według daty rozpoczęcia kolekcji, z możliwością ustalenia kolejności od najnowszych do najstarszych.

Funkcja **print_filtered_clothes** :

Pozwala na filtrowanie ubrań na podstawie różnych kryteriów, takich jak materiał, rozmiar, cena czy płeć. Wyświetla ubrania spełniające kryteria użytkownika.

- **Collection.py** :

```python
class Collection:
    _all_collections = []

    def __init__(self):
        self.collection_id = None
        self.name = None
        self.start_date = None
        self.end_date = None
        Collection._all_collections.append(self)

    @classmethod
    def print_all_collections(cls):
        for collection in cls._all_collections:
            print(
                f"ID: {collection.collection_id}, Name: {collection.name}, Start Date: {collection.start_date}, "
                f"End Date: {collection.end_date}")

    @classmethod
    def get_collection_by_id(cls, collection_id):
        for collection in cls._all_collections:
            if collection.collection_id == collection_id:
                return collection
        return None

    @classmethod
    def fetch_collections_from_db(cls, db_connection):
        cls._all_collections.clear()

        db_connection.cursor.execute("SELECT collection_id, name, start_date, end_date FROM Collections")
        rows = db_connection.cursor.fetchall()
        for row in rows:
            cls.create_collection(row[0], row[1], row[2], row[3])

    @classmethod
    def refresh_collections(cls, db_connection):
        cls.fetch_collections_from_db(db_connection)

    @classmethod
    def create_collection(cls, collection_id, name, start_date, end_date):
        collection = cls()
        collection.collection_id = collection_id
        collection.name = name
        collection.start_date = start_date
        collection.end_date = end_date
        return collection
```

Funkcja **__init__** :

Inicjalizuje nowy obiekt klasy `Collection`. Przypisuje wartości atrybutom, takim jak identyfikator kolekcji, nazwa, data rozpoczęcia i zakończenia. Dodaje utworzoną kolekcję do listy wszystkich kolekcji **_all_collections**.

Funkcja **print_all_collections** :

Wyświetla szczegóły wszystkich kolekcji przechowywanych w liście **_all_collections**, takie jak identyfikator, nazwa, data rozpoczęcia i zakończenia.

Funkcja **get_collection_by_id** :

Znajduje i zwraca kolekcję o określonym identyfikatorze.

Funkcja **fetch_collections_from_db** :

Pobiera informacje o kolekcjach z bazy danych i aktualizuje listę `_all_collections`.

Funkcja **refresh_collections** :

Aktualizuje listę **_all_collections**, pobierając najnowsze informacje z bazy danych.

Funkcja create_collection :

Tworzy i zwraca nową kolekcję z podanymi danymi, dodając ją jednocześnie do listy **_all_collections**.

- **Db_communication** :

```
import mysql.connector


class DB_Communication:
    def __init__(self):
        self.conn = None
        self.cursor = None
        self.connect_to_db()

    def connect_to_db(self):
        try:
            self.conn = mysql.connector.connect(
                host='localhost',
                user='root',
                password='jkhfasddjk123',
                port='3306',
                database='sklep'
            )
            self.cursor = self.conn.cursor()
        except mysql.connector.Error as e:
            print(f"Error connecting to MySQL: {e}")
```

Funkcja **__init__** :

Inicjalizuje nowy obiekt klasy **DB_Communication**. Inicjalizuje atrybuty **conn** i **cursor** na wartość **None** oraz wykonuje połączenie z bazą danych poprzez wywołanie metody **connect_to_db**.

Funkcja **connect_to_db** :

Próbuje nawiązać połączenie z bazą danych MySQL, korzystając z podanych danych dostępowych (host, użytkownik, hasło, port, nazwa bazy danych). Jeśli połączenie się powiedzie, ustawia atrybuty **conn** i **cursor** na odpowiednie wartości. W przypadku błędu wypisuje komunikat o nieudanej próbie połączenia.

- **Delivery.py** :

```python
class Delivery:

    def __init__(self):
        self.delivery_id = None
        self.city = None
        self.street = None
        self.number = None
        self.postal_code = None
        self.country = None
        Delivery._delivery_details.append(self)

    @classmethod
    def fetch_deliveries_by_user(cls, db_connection, user_id):
        cls._delivery_details.clear()
        query = """
                SELECT DISTINCT Delivery.delivery_id, Delivery.city, Delivery.street, Delivery.number,
                Delivery.postal_code, Delivery.country FROM Delivery JOIN Orders ON
                Delivery.delivery_id = Orders.delivery_id WHERE Orders.user_id = %s
            """

        db_connection.cursor.execute(query, (user_id,))
        rows = db_connection.cursor.fetchall()
        for row in rows:
            delivery = cls()
            (delivery.delivery_id, delivery.city, delivery.street, delivery.number,
             delivery.postal_code, delivery.country) = row

    @classmethod
    def print_delivery_details(cls):
        for delivery in cls._delivery_details:
            print(f"Delivery ID: {delivery.delivery_id}, City: {delivery.city}, Street: {delivery.street}, "
                  f"Number: {delivery.number}, Postal Code: {delivery.postal_code}, Country: {delivery.country}")

    def insert_new_delivery(self, db_connection):

        print("New order will be created with this delivery details")
        self.city = ValidationUtility.get_validated_input_varchar("Enter city: ", 100)
        self.street = ValidationUtility.get_validated_input_varchar("Enter street: ", 100)
        self.number = ValidationUtility.get_validated_input_integer("Enter number: ", 10)
        self.postal_code = ValidationUtility.get_validated_input_integer("Enter postal code: ", 15)
        self.country = ValidationUtility.get_validated_input_varchar("Enter country: ", 100)

        try:
            db_connection.cursor.execute(
                "INSERT INTO Delivery (city, street, number, postal_code, country) VALUES (%s, %s, %s, %s, %s)",
                (self.city, self.street, self.number, self.postal_code, self.country))
            db_connection.conn.commit()
            self.delivery_id = db_connection.cursor.lastrowid
        except mysql.connector.Error as e:
            print(f"Error in database operation: {e}")
        finally:
            print("The delivery details has been added to the database and will be visible to the user "
                  "after finalizing the basket.")
            return self.delivery_id

    @classmethod
    def select_delivery(cls):
        cls.print_delivery_details()
        try:
            selected_id = int(input("Enter the ID of the delivery you want to select: "))
        except ValueError:
            print("Invalid input. Please enter a numeric ID.")
            return None

        for delivery in cls._delivery_details:
            if delivery.delivery_id == selected_id:
                return selected_id
        print("No delivery found with the provided ID.")
        return None

    def menu(self, db_connection, user_id):
        while True:
            print("\nDelivery Management Menu:")
            print("1. Print Delivery Details")
            print("2. Select Saved Delivery Details")
            print("3. Insert New Delivery")
            print("4. Exit")
            self.fetch_deliveries_by_user(db_connection, user_id)
            choice = input("Enter your choice: ")

            if choice == "1":
                self.print_delivery_details()
            elif choice == "2":
                selected_delivery = self.select_delivery()
                if selected_delivery:
                    print("Selected delivery details:")
                    print(selected_delivery)
                    return selected_delivery
            elif choice == "3":
                self.insert_new_delivery(db_connection)
            elif choice == "4":
                break
            else:
                print("Invalid choice. Please try again.")
```

Atrybut klasowy **_delivery_details** :

Lista przechowująca wszystkie dostępne informacje o dostawach.

Funkcja **__init__** :

Inicjalizuje nowy obiekt klasy **Delivery**. Przypisuje wartości atrybutom takim jak identyfikator dostawy (**delivery_id**), miasto (**city**), ulica (**street**), numer (**number**), kod pocztowy (**postal_code**), kraj (**country**). Dodaje utworzoną dostawę do listy **_delivery_details**.

Funkcja **fetch_deliveries_by_user** :

Pobiera informacje o dostawach związanych z danym użytkownikiem na podstawie identyfikatora użytkownika. Czyści wcześniejsze informacje o dostawach i aktualizuje listę **_delivery_details**.

Funkcja **print_delivery_details** :

Wyświetla szczegóły dostaw przechowywanych w liście **_delivery_details**, takie jak identyfikator, miasto, ulica, numer, kod pocztowy i kraj.

Funkcja **insert_new_delivery** :

Pozwala użytkownikowi wprowadzić nowe dane dostawy. Dodaje te dane do bazy danych, a następnie aktualizuje atrybut **delivery_id** utworzonego obiektu. Komunikuje użytkownikowi o pomyślnym dodaniu danych dostawy do bazy danych.

Funkcja **select_delivery** :

Wyświetla dostępne dostawy, a następnie pozwala użytkownikowi wybrać dostawę na podstawie identyfikatora. Zwraca identyfikator wybranej dostawy lub **None**, jeśli dostawa o podanym identyfikatorze nie istnieje.

Funkcja **menu** :

Zapewnia interaktywne menu zarządzania dostawami, umożliwiając użytkownikowi wybór opcji takich jak wyświetlanie, wybieranie lub dodawanie dostaw. Loop kończy się, gdy użytkownik wybierze opcję "**Exit**".

- **Order.py** :

```python
import mysql.connector


class Order:
    orders = {}

    @classmethod
    def fetch_orders_by_user(cls, db_connection, user_id):
        query = """
                SELECT Orders.order_id, Orders.amount,
                       Payments.status, Payments.payment_form, Payments.date,
                       Delivery.city, Delivery.street, Delivery.number, Delivery.postal_code, Delivery.country,
                       Clothes.material, Clothes.size, Clothes.sex, Clothes.price,
                       Collections.name, Collections.start_date, Collections.end_date
                FROM Orders
                JOIN Payments ON Orders.payment_id = Payments.payment_id
                JOIN Delivery ON Orders.delivery_id = Delivery.delivery_id
                JOIN Basket ON Orders.order_id = Basket.order_id
                JOIN Clothes ON Basket.clothes_id = Clothes.clothes_id
                JOIN Collections ON Clothes.collection_id = Collections.collection_id
                WHERE Orders.user_id = %s
                """

        db_connection.cursor.execute(query, (user_id,))
        raw_orders = db_connection.cursor.fetchall()

        if not cls.orders:
            for row in raw_orders:
                order_id = row[0]
                if order_id not in cls.orders:
                    cls.orders[order_id] = {
                        'order_id': order_id,
                        'amount': row[1],
                        'payment_details': {'status': row[2], 'payment_form': row[3], 'date': row[4]},
                        'delivery_details': {'city': row[5], 'street': row[6], 'number': row[7], 'postal_code': row[8],
                                             'country': row[9]},
                        'clothes': [],
                        'collections': []
                    }
                clothes_details = {'material': row[10], 'size': row[11], 'sex': row[12], 'price': row[13]}
                collection_details = {'name': row[14], 'start_date': row[15], 'end_date': row[16]}
                cls.orders[order_id]['clothes'].append(clothes_details)
                cls.orders[order_id]['collections'].append(collection_details)
        else:
            updated_orders = {}

            for row in raw_orders:
                order_id = row[0]
                if order_id not in updated_orders:
                    updated_orders[order_id] = {
                        'order_id': order_id,
                        'amount': row[1],
                        'payment_details': {'status': row[2], 'payment_form': row[3], 'date': row[4]},
                        'delivery_details': {'city': row[5], 'street': row[6], 'number': row[7], 'postal_code': row[8],
                                             'country': row[9]},
                        'clothes': [],
                        'collections': []
                    }
                clothes_details = {'material': row[10], 'size': row[11], 'sex': row[12], 'price': row[13]}
                collection_details = {'name': row[14], 'start_date': row[15], 'end_date': row[16]}
                updated_orders[order_id]['clothes'].append(clothes_details)
                updated_orders[order_id]['collections'].append(collection_details)

            cls.orders = updated_orders
```

```python
    @classmethod
    def print_all_orders_by_user(cls):
        for order_id, order_details in cls.orders.items():
            print()
            print(f"Order ID: {order_id}, Amount: {order_details['amount']}")
            payment = order_details['payment_details']
            print(
                f"Payment Status: {payment['status']}, Payment Form: {payment['payment_form']}, "
                f"Payment Date: {payment['date']}")
            delivery = order_details['delivery_details']
            print(
                f"Delivery Address: {delivery['city']}, {delivery['street']} {delivery['number']}, "
                f"{delivery['postal_code']}, {delivery['country']}")
            clothes_index = 0
            collections_index = 0

            while clothes_index < len(order_details['clothes']) and collections_index < len(
                    order_details['collections']):
                clothes = order_details['clothes'][clothes_index]
                collection = order_details['collections'][collections_index]

                print(
                    f"Material: {clothes['material']}, Size: {clothes['size']}, Sex: {clothes['sex']}, "
                    f"Price: {clothes['price']}")

                print(
                    f"{collection['name']}, Collection Start Date: {collection['start_date']}, "
                    f"Collection End Date: {collection['end_date']}")

                clothes_index += 1
                collections_index += 1

    @classmethod
    def cancel_order(cls, db_connection, order_id):

        try:

            db_connection.cursor.callproc("CancelOrder", [order_id])
            db_connection.conn.commit()
            print("Order canceled successfully.")

        except mysql.connector.Error as e:
            print(f"Error during order cancellation: {e}")

    @classmethod
    def print_order_details(cls, order_id):
        int_order_id = int(order_id)
        order_details = cls.orders.get(int_order_id)
        counter = 0
        if order_details:
            print(f"Order ID: {order_id}, Amount: {order_details['amount']}")
            payment = order_details['payment_details']
            print(f"Payment Status: {payment['status']}, Payment Form: {payment['payment_form']}, Payment Date: "
                  f"{payment['date']}")
            delivery = order_details['delivery_details']
            print(f"Delivery Address: {delivery['city']}, {delivery['street']} {delivery['number']}, "
                  f"{delivery['postal_code']}, {delivery['country']}")
            for clothes in order_details['clothes']:
                counter += 1
                print(f"{counter}: Material: {clothes['material']}, Size: {clothes['size']}, Sex: {clothes['sex']}, "
                      f"Price: {clothes['price']}")
            counter = 0
            for collection in order_details['collections']:
                counter += 1
                print(f"{counter}: {collection['name']}, Collection Start Date: {collection['start_date']}, "
                      f"Collection End Date: {collection['end_date']}")
        else:
            print(f"No details found for Order ID: {order_id}")

    @staticmethod
    def add_new_order(db_connection, amount, user_id, payment_id, delivery_id):
        try:
            db_connection.cursor.callproc("addOrder", [amount, user_id, payment_id, delivery_id])
            db_connection.conn.commit()
            print("New order added successfully.")
        except mysql.connector.Error as e:
            print(f"Error during order addition: {e}")

    @staticmethod
    def update_order_attributes(db_connection, order_id, new_amount, new_payment_id, new_delivery_id):
        try:
            db_connection.cursor.callproc("updateOrder",
                                          [order_id, new_amount, new_payment_id, new_delivery_id])
            db_connection.conn.commit()
            print("Order attributes updated successfully.")
        except mysql.connector.Error as e:
            print(f"Error during order attribute update: {e}")
```

Funkcja **fetch_orders_by_user** :

Pobiera zamówienia użytkownika z bazy danych na podstawie identyfikatora użytkownika. Aktualizuje słownik **orders** przechowujący szczegóły zamówień, uwzględniając informacje o płatnościach, dostawach, ubraniach i kolekcjach.

Funkcja **print_all_orders_by_user** :

Wyświetla szczegóły wszystkich zamówień użytkownika, takie jak identyfikator zamówienia, kwota, status płatności, forma płatności, data płatności, adres dostawy oraz szczegóły ubrań i kolekcji w zamówieniu.

Funkcja **cancel_order** :

Anuluje zamówienie o podanym identyfikatorze. Wywołuje procedurę składowaną "**CancelOrder**" na podstawie identyfikatora zamówienia.

Funkcja **print_order_details** :

Wyświetla szczegóły zamówienia o podanym identyfikatorze, takie jak identyfikator zamówienia, kwota, status płatności, forma płatności, data płatności, adres dostawy oraz szczegóły ubrań i kolekcji w zamówieniu.

Funkcja **add_new_order** :

Dodaje nowe zamówienie do bazy danych na podstawie przekazanych informacji, takich jak kwota, identyfikator użytkownika, identyfikator płatności i identyfikator dostawy.

Funkcja **update_order_attributes** :

Aktualizuje atrybuty zamówienia o podanym identyfikatorze, takie jak kwota, identyfikator płatności i identyfikator dostawy.

- **Payment.py** :

```python
class Payment:
    def __init__(self):
        self.payment_id = None
        self.status = None
        self.payment_form = None
        self.payment_date = None
        self.payment_details = []

    def fetch_payments_by_user(self, db_connection, user_id):
        self.payment_details.clear()
        query = """
            SELECT DISTINCT Payments.payment_id, Payments.status, Payments.payment_form, Payments.date
            FROM Payments
            JOIN Orders ON Payments.payment_id = Orders.payment_id
            WHERE Orders.user_id = %s
        """
        db_connection.cursor.execute(query, (user_id,))
        payments = db_connection.cursor.fetchall()

        for payment in payments:
            payment_detail = {
                'payment_id': payment[0],
                'status': payment[1],
                'payment_form': payment[2],
                'payment_date': payment[3]
            }
            self.payment_details.append(payment_detail)

    def print_payment_details(self):
        for detail in self.payment_details:
            print(detail)

    def insert_new_payment(self, db_connection):
        self.status = "unpaid"

        while True:
            payment_form = input("Enter payment form (card, blik, transfer): ").lower()
            if payment_form in ["card", "blik", "transfer"]:
                self.payment_form = payment_form
                break
            else:
                print("Invalid input. Please enter card, blik, or transfer.")

        db_connection.cursor.execute(
            "INSERT INTO Payments (status, payment_form, date) VALUES (%s, %s, %s)",
            (self.status, self.payment_form, self.payment_date))
        db_connection.conn.commit()
        self.payment_id = db_connection.cursor.lastrowid
        print("The payment form has been added to the database and will be visible to the user after finalizing the "
              "basket.")
        return self.payment_id

    def select_saved_payment_details(self, payment_id):
        found_payment = None

        for payment_detail in self.payment_details:
            if payment_detail['payment_id'] == payment_id:
                found_payment = payment_detail
                break

        if found_payment:
            self.status = found_payment['status']
            self.payment_form = found_payment['payment_form']
            self.payment_date = found_payment['payment_date']
            return payment_id
        else:
            print("Payment details not found.")
            return None

    def menu(self, db_connection, user_id):
        while True:
            print("\n--- Payment options menu ----")
            print("1: Print Payment Details")
            print("2: Insert New Payment")
            print("3: Select Saved Payment Details")
            print("4: Exit")
            self.fetch_payments_by_user(db_connection, user_id)
            choice = input("Enter your choice: ")

            if choice == '1':
                self.print_payment_details()
            elif choice == '2':
                payment_id = self.insert_new_payment(db_connection)
                return payment_id
            elif choice == '3':
                payment_id = input("Enter payment ID: ")
                selected_payment = self.select_saved_payment_details(payment_id)
                if selected_payment:
                    print("Selected payment details:")
                    print(selected_payment)
                    return selected_payment
            elif choice == '4':
                print("Exiting the payment menu.")
                break
            else:
                print("Invalid choice. Please select a valid option (1-4).")
```

Funkcja **__init__** :

Inicjalizuje nowy obiekt klasy **Payment**, ustawiając atrybuty na wartości początkowe.

Funkcja **fetch_payments_by_user** :

Pobiera informacje o płatnościach użytkownika z bazy danych na podstawie identyfikatora użytkownika. Aktualizuje listę **payment_details**, przechowującą szczegóły płatności.

Funkcja **print_payment_details** :

Wyświetla szczegóły wszystkich dostępnych płatności przechowywanych w liście **payment_details**.

Funkcja **insert_new_payment** :

Pozwala użytkownikowi wprowadzić nowe dane płatności, takie jak forma płatności (karta, blik, przelew). Dodaje nową płatność do bazy danych i zwraca jej identyfikator.

Funkcja **select_saved_payment_details** :

Pozwala na wybór zapisanych szczegółów płatności na podstawie identyfikatora płatności. Ustawia atrybuty obiektu na wybrane szczegóły płatności.

Funkcja **menu** :

Obsługuje menu opcji płatności, umożliwiając użytkownikowi wybór działań takich jak wyświetlanie, dodawanie nowej płatności oraz wybór zapisanych szczegółów płatności.

- **Security_utility.py** :

```python
import hashlib


class SecurityUtility:
    SECRET_SALT = b'ad'

    @staticmethod
    def hash_password(password):
        hashed_password = hashlib.sha256(SecurityUtility.SECRET_SALT + password.encode()).hexdigest()
        return hashed_password
```

Funkcja **hash_password** :

Przyjmuje hasło jako argument, a następnie używa algorytmu SHA-256 do wygenerowania skrótu (hashu) tego hasła. Do hashowania dodawany jest tajny sól (SECRET_SALT), co zwiększa bezpieczeństwo operacji. Ostateczny zhashowany wynik jest reprezentowany jako szesnastkowy ciąg znaków i zwracany jako wynik funkcji. Wprowadzenie tajnej soli jest praktyką stosowaną w celu utrudnienia ataków typu "**rainbow table**" oraz zwiększenia odporności na ataki bruteforce.

- **Ui_app.py :**

```python
from validation_utility import ValidationUtility
from security_utility import SecurityUtility
from db_communication import DB_Communication
from user import User
from payment import Payment
from delivery import Delivery
from basket import Basket
from order import Order
from clothes import Clothes
import mysql.connector


class UI_App:
    def __init__(self):
        self.db = DB_Communication()
        self.user = None

    def login(self):
        user_login = ValidationUtility.get_validated_input_varchar("Enter login: ", 20)
        user_password = ValidationUtility.get_validated_input_varchar("Enter password: ", 20)
        hashed_password = SecurityUtility.hash_password(user_password)

        try:
            self.db.cursor.execute("SELECT * FROM Users WHERE login = %s AND password = %s", (user_login,
                                                                                               hashed_password))
            user_data = self.db.cursor.fetchone()
            if user_data:
                current_user = User(*user_data)
                self.user = current_user
                print(f"Welcome, {current_user.name}")
                return True
            else:
                print("Login failed. Incorrect username or password.")
                return False
        except mysql.connector.Error as e:
            print(f"Error during login: {e}")
            return False

    def create_account(self):
        while True:
            user_login = ValidationUtility.get_validated_input_varchar("Enter login: ", 20)
            self.db.cursor.execute("SELECT login FROM Users WHERE login = %s", (user_login,))
            if self.db.cursor.fetchone():
                print("Login already exists. Please choose a different login.")
            else:
                break
        user_password = ValidationUtility.get_validated_input_varchar("Enter password: ", 20)
        hashed_password = SecurityUtility.hash_password(user_password)
        user_email = ValidationUtility.get_validated_input_varchar("Enter email: ", 255)
        user_phone = ValidationUtility.get_validated_input_varchar("Enter phone: ", 15)
        user_name = ValidationUtility.get_validated_input_varchar("Enter name: ", 45)
        user_surname = ValidationUtility.get_validated_input_varchar("Enter surname: ", 45)
        user_role = "customer"
        # user_role = ValidationUtility.get_validated_input_varchar("Enter role (admin, employee, customer): ", 20)

        if user_role not in ['admin', 'employee', 'customer']:
            print("Invalid role. Please enter 'admin', 'employee', or 'customer'.")
            return


        try:
            self.db.cursor.callproc('CreateUserAccount',
                                    [user_login, hashed_password, user_email, user_phone, user_name, user_surname,
                                     user_role])
            self.db.conn.commit()
            print("Account created successfully.")
            return True
        except mysql.connector.Error as e:
            print(f"Error in account creation: {e}")
            return False
```

```
71
72      def login_menu(self):
73          while True:
74              print("\n--- Login Menu ---")
75              print("1: Create an Account")
76              print("2: Login to an Account")
77              print("3: Exit")
78              choice = input("Enter your choice: ")
79
80              if choice == '1':
81                  self.create_account()
82              elif choice == '2':
83                  is_success = self.login()
84                  if is_success:
85                      return True
86              elif choice == '3':
87                  return False
88              else:
89                  print("Invalid choice. Please try again.")
90
91      def main_menu(self):
92
93          while True:
94              print("\n--- Main Menu ---")
95              print("1) Account")
96              print("2) Orders")
97              print("3) Basket")
98              print("4) Clothes")
99              print("5) Log out")
100
101             choice = input("Select an option: ")
102
103             if choice == "1":
104                 print("Account menu selected")
105                 self.user.user_menu(self.db)
106             elif choice == "2":
107                 print("Order menu selected")
108                 self.order_menu()
109             elif choice == "3":
110                 self.basket_menu()
111             elif choice == "4":
112                 print("Clothes menu selected")
113                 self.clothes_menu()
114             elif choice == "5":
115                 print("Goodbye!")
116                 break
117             else:
118                 print("Invalid choice. Please select a valid option.")
119
120     def order_menu(self):
121         orders = Order()
122         while True:
123             print("\n--- Order Menu ---")
124             print("1: View all Orders")
125             print("2: Cancel Selected Order")
126             print("3: Display Selected Order Details")
127             print("4: Exit")
128             orders.fetch_orders_by_user(self.db, self.user.user_id)
129             choice = input("Enter your choice: ")
130
131             if choice == '1':
132                 orders.print_all_orders_by_user()
133             elif choice == '2':
134                 order_id = input("Enter the Order ID to cancel: ")
135                 orders.cancel_order(self.db, order_id)
136             elif choice == '3':
137                 order_id = input("Enter the Order ID to view details: ")
138                 orders.print_order_details(order_id)
139             elif choice == '4':
140                 break
141             else:
142                 print("Invalid choice. Please try again.")
143
```

```
    def basket_menu(self):

        basket = Basket(self.db, self.user.user_id)
        delivery = Delivery()
        payment = Payment()
        selected_payment = {}
        selected_delivery = {}
        order_id = 0
        total_cost = 0

        while True:

            print("\n--- Basket Menu ----")
            print("1: Edit the basket")
            print("2: Edit the shipping options")
            print("3: Edit the payment options")
            print("4: Finalize the order")
            print("5: Exit")

            choice = input("Enter your choice: ")

            if choice == '1':
                order_id, total_cost = basket.view_edit_basket()

            elif choice == '2':
                selected_delivery = delivery.menu(self.db, self.user.user_id)

            elif choice == '3':
                selected_payment = payment.menu(self.db, self.user.user_id)

            elif choice == '4':
                Order.update_order_attributes(self.db, order_id, total_cost, selected_payment, selected_delivery)

            elif choice == '5':
                print("Exiting the menu.")
                break
            else:
                print("Invalid choice. Please select a valid option (1-5).")

    def clothes_menu(self):
        clothes = Clothes()
        clothes.fetch_clothes_from_db(self.db)
        while True:
            print("\n--- Clothes Menu ----")
            print("1: View all Clothes")
            print("2. Display Clothes by Price")
            print("3. Display Clothes by Collection")
            print("4. Search clothes with filters")
            print("5. Exit")
            clothes.refresh_clothes(self.db)
            choice = input("Enter your choice: ")

            if choice == '1':
                clothes.print_all_clothes()
            elif choice == '2':
                order = input("Enter 'asc' for ascending or 'desc' for descending order of price: ").strip().lower()
                ascending = True if order == 'asc' else False
                clothes.print_sorted_by_price(ascending)
            elif choice == '3':
                order = input("Enter 'newest' for newest first or "
                              "'oldest' for oldest first in collection date: ").strip().lower()
                newest_first = True if order == 'newest' else False
                clothes.print_sorted_by_collection_date(newest_first)
            elif choice == '4':
                clothes.print_filtered_clothes()
            elif choice == '5':
                print("Exiting the program. Goodbye!")
                break
            else:
                print("Invalid choice. Please select a valid option (1-6).")


def main():
    is_logged = True
    app = UI_App()
    while is_logged:
        is_logged = app.login_menu()
        if is_logged:
            app.main_menu()


if __name__ == "__main__":
    main()
```

Funkcja **login_menu** :

Obsługuje proces logowania i tworzenia nowego konta. W zależności od wyboru użytkownika może prowadzić do utworzenia nowego konta, zalogowania się lub wyjścia z programu.

Funkcja **main_menu** :

Główna pętla interfejsu, gdzie użytkownik ma dostęp do różnych funkcji, takich jak zarządzanie kontem, zamówieniami, koszykiem, ubraniami. Umożliwia również wylogowanie.

Funkcja **order_menu** :

Zarządza opcjami związanymi z zamówieniami, takimi jak przeglądanie wszystkich zamówień, anulowanie wybranego zamówienia czy wyświetlanie szczegółów zamówienia.

Funkcja **basket_menu** :

Odpowiada za zarządzanie koszykiem użytkownika, umożliwia edycję zawartości koszyka, wybór opcji dostawy i płatności oraz finalizację zamówienia.

Funkcja **clothes_menu**:

Obsługuje funkcje związane z ubraniami, takie jak przeglądanie dostępnych ubrań, sortowanie według ceny czy kolekcji, oraz filtrowanie ubrań.

Funkcja **create_account** :

Obsługuje proces tworzenia nowego konta, sprawdzając unikalność loginu oraz dodając użytkownika do bazy danych.

Funkcja **login** :

Obsługuje proces logowania, sprawdzając poprawność wprowadzonych danych w porównaniu do informacji w bazie danych.

Funkcja **main** :

Funkcja główna programu, inicjuje obiekt klasy **UI_App** i uruchamia interfejs użytkownika, prowadzący przez proces logowania i obsługujący główne funkcje programu.

- **User.py :**

```python
from validation_utility import ValidationUtility
import sys
import mysql.connector


class User:
    def __init__(self, user_id: int, login: str, password: str, email: str, phone: str, name: str, surname: str,
                 role: str):
        self.user_id = user_id
        self.login = login
        self.password = password  # Note: Storing passwords in plain text is not secure
        self.email = email
        self.phone = phone
        self.name = name
        self.surname = surname
        self.role = role

    def user_menu(self, db):
        while True:
            print("\n--- User Menu ---")
            print("1: Display Attributes")
            print("2: Update Attributes")
            print("3: Delete Account")
            print("4: Exit")
            choice = input("Enter your choice: ")

            if choice == '1':
                self.display_attributes()
            elif choice == '2':
                self.update_attributes(db)
            elif choice == '3':
                self.delete_account(db)
            elif choice == '4':
                return 4
            else:
                print("Invalid choice. Please try again.")

    def display_attributes(self):
        print(f"Login: {self.login}")
        print(f"Email: {self.email}")
        print(f"Phone: {self.phone}")
        print(f"Name: {self.name}")
        print(f"Surname: {self.surname}")
        print(f"Role: {self.role}")

    def update_attributes(self, db):

        new_email = ValidationUtility.get_validated_input_varchar("Enter new email (or press Enter to skip): ", 255)
        new_phone = ValidationUtility.get_validated_input_varchar("Enter new phone (or press Enter to skip): ", 15)
        new_name = ValidationUtility.get_validated_input_varchar("Enter new name (or press Enter to skip): ", 45)
        new_surname = ValidationUtility.get_validated_input_varchar("Enter new surname (or press Enter to skip): ", 45)

        db.cursor.callproc('UpdateUserDetails', [self.user_id, new_email or self.email, new_phone or
                                                 self.phone, new_name or self.name, new_surname or self.surname])
        db.conn.commit()

        self.email = new_email if new_email else self.email
        self.phone = new_phone if new_phone else self.phone
        self.name = new_name if new_name else self.name
        self.surname = new_surname if new_surname else self.surname
        print("User details updated successfully.")

    def delete_account(self, db):
        confirmation = input("Are you sure you want to delete your account? (yes/no): ")
        if confirmation.lower() == 'yes':
            try:

                db.cursor.callproc('DeleteUserAccount', [self.user_id])
                db.conn.commit()
                print("Account deleted successfully.")
                return 3

            except mysql.connector.Error as e:
                print(f"Error during account deletion: {e}")
                sys.exit()
        else:
            print("Account deletion cancelled.")
```

Funkcja **__init__** :

Inicjalizuje obiekt User z podanymi atrybutami.

Funkcja **user_menu** :

Odpowiada za interakcję z użytkownikiem poprzez wyświetlanie menu z opcjami związanymi z zarządzaniem kontem. Umożliwia użytkownikowi wybór między wyświetlaniem, aktualizacją atrybutów, usunięciem konta lub wyjściem z menu.

Funkcja **display_attributes** :

Wyświetla atrybuty użytkownika, takie jak login, e-mail, numer telefonu, imię, nazwisko i rolę.

Funkcja **update_attributes** :

Pozwala użytkownikowi na aktualizację wybranych atrybutów, takich jak e-mail, numer telefonu, imię i nazwisko. Przyjmuje nowe dane od użytkownika i aktualizuje informacje w bazie danych.

Funkcja **delete_account** :

Obsługuje proces usuwania konta użytkownika. Weryfikuje potwierdzenie użytkownika przed usunięciem konta, a następnie wykonuje procedurę usuwania konta w bazie danych.

- **Validation_utlity.py** :

```python
import re
from decimal import Decimal, InvalidOperation


class ValidationUtility:

    @staticmethod
    def validate_for_sql_injection(value):

        patterns = [
            r'--',  # SQL comment
            r';',  # Statement separator
            r'\/\*',  # Multi-line comment
            r'\bOR\b', '\bAND\b',  # Logical operators
            r'[\s\r\n]*\bDROP\b', r'[\s\r\n]*\bTABLE\b', r'[\s\r\n]*\bINSERT\b',  # DDL/DML keywords
            r'[\s\r\n]*\bDELETE\b', r'[\s\r\n]*\bUPDATE\b',
            r'[\s\r\n]*\bSELECT\b', r'[\s\r\n]*\bFROM\b',
            r'EXEC\(', r'EXECUTE\('  # Executing stored procedures/functions
        ]

        if any(re.search(pattern, value, re.IGNORECASE) for pattern in patterns):
            print(f"Potentially harmful input detected: '{value}'. Please try again.")
            return False
        return True

    @staticmethod
    def validate_varchar(value, max_length):
        if not isinstance(value, str) or len(value) > max_length:
            print(f"String value exceeds the max allowed length of {max_length}. Please try again.")
            return False
        return True

    @staticmethod
    def validate_integer(value, max_length):
        try:

            int_value = int(value)

            if len(str(abs(int_value))) > max_length:
                print(f"Integer value exceeds the max allowed length of {max_length} digits. Please try again.")
                return False

            return True
        except ValueError:
            print(f"Value '{value}' is not a valid integer. Please try again.")
            return False

    @staticmethod
    def validate_decimal(value):
        try:

            decimal_value = Decimal(value)
            decimal_value = decimal_value.quantize(Decimal('0.00'))

            sign, digits, exponent = decimal_value.as_tuple()

            digits_before_decimal = len(digits) + exponent if exponent < 0 else len(digits)
            digits_after_decimal = -exponent if exponent < 0 else 0

            if digits_before_decimal > 6 or digits_after_decimal > 2:
                print(
                    "Value must be a decimal number with up to 6 digits before and 2 digits after the decimal point. "
                    "Please try again.")
                return False

            return True
        except (InvalidOperation, ValueError):
            print(f"Value '{value}' is not a valid decimal number. Please try again.")
            return False

    @staticmethod
    def get_validated_input_varchar(prompt, max_length):
        while True:
            user_input = input(prompt)
            if (ValidationUtility.validate_varchar(user_input, max_length) and
                    ValidationUtility.validate_for_sql_injection(user_input)):
                return user_input

    @staticmethod
    def get_validated_input_integer(prompt, max_length):
        while True:
            user_input = input(prompt)
            if ValidationUtility.validate_integer(user_input, max_length):
                return int(user_input)

    @staticmethod
    def get_validated_input_decimal(prompt):
        while True:
            user_input = input(prompt)
            if ValidationUtility.validate_decimal(user_input):
                return Decimal(user_input).quantize(Decimal('0.00'))
```

Funkcja validate_for_sql_injection :

Sprawdza, czy wartość zawiera potencjalnie szkodliwe frazy związane z atakami SQL injection. W przypadku wykrycia takich fraz, wyświetla komunikat ostrzegawczy.

Funkcja validate_varchar :

Waliduje ciąg znaków (varchar) pod kątem długości, upewniając się, że nie przekracza maksymalnej długości. Zwraca wartość logiczną w zależności od wyniku walidacji.

Funkcja validate_integer :

Waliduje liczbę całkowitą pod kątem długości, upewniając się, że nie przekracza maksymalnej liczby cyfr. Również sprawdza, czy wartość jest poprawną liczbą całkowitą. Zwraca wartość logiczną w zależności od wyniku walidacji.

Funkcja validate_decimal :

Waliduje liczbę dziesiętną pod kątem liczby cyfr przed i po przecinku. Również sprawdza, czy wartość jest poprawną liczbą dziesiętną. Zwraca wartość logiczną w zależności od wyniku walidacji.

Funkcja get_validated_input_varchar :

Pobiera wejściowy ciąg znaków od użytkownika, a następnie stosuje walidację dla wartości varchar i SQL injection. Zwraca wprowadzony ciąg znaków, jeśli jest poprawny.

Funkcja get_validated_input_integer :

Pobiera wejściową liczbę całkowitą od użytkownika, a następnie stosuje walidację dla wartości całkowitej. Zwraca wprowadzoną liczbę całkowitą, jeśli jest poprawna.

Funkcja get_validated_input_decimal :

Pobiera wejściową liczbę dziesiętną od użytkownika, a następnie stosuje walidację dla wartości dziesiętnej. Zwraca wprowadzoną liczbę dziesiętną, jeśli jest poprawna.

5. Przetestowanie aplikacji

W celu przetestowania aplikacji napisaliśmy testy. Testy są napisane przy użyciu frameworka pytest oraz unittest.mock do tworzenia atrap obiektów. Testy sprawdzają różne aspekty funkcji i metod w aplikacji, takie jak sprawdzanie poprawności danych, wyświetlanie informacji, aktualizacja atrybutów użytkownika, obsługa zamówień i dostaw, oraz interakcje z bazą danych. Parametryzacja testów pozwala na łatwe dodawanie nowych przypadków testowych.

Poniżej znajduje się kod testów w Pythonie:

```
import pytest
from unittest.mock import patch, Mock, MagicMock

from decimal import Decimal
from db_communication import DB_Communication
from ui_app import UI_App
from user import User
from validation_utility import ValidationUtility
from order import Order
from delivery import Delivery


@pytest.fixture
def mock_db():
    db = Mock(spec=DB_Communication)
    db.cursor = Mock()
    db.conn = Mock()
    return db


@pytest.fixture
def user():
    return User(1, 'test_login', 'password123', 'test@example.com',
'1234567890', 'John', 'Doe', 'user')


@pytest.mark.parametrize("value, expected", [
    ("safe_input", True),
    ("SELECT * FROM users", False),
    ("pass1; DROP TABLE users", False),
    ("sdffsdfsd-- comment", False),
])
def test_validate_for_sql_injection(value, expected):
    assert ValidationUtility.validate_for_sql_injection(value) == expected


@pytest.mark.parametrize("value, max_length, expected", [
    ("short_string", 20, True),
    ("a" * 21, 20, False),
    (123, 20, False),
])
def test_validate_varchar(value, max_length, expected):
    assert ValidationUtility.validate_varchar(value, max_length) ==
expected


@pytest.mark.parametrize("value, max_length, expected", [
    ("12345", 5, True),
```

```
    ("123456", 5, False),
    ("abc", 5, False),
])
def test_validate_integer(value, max_length, expected):
    assert ValidationUtility.validate_integer(value, max_length) ==
expected


@pytest.mark.parametrize("value, expected", [
    ("12.34", True),
    ("5232132.45", False),
    ("abc", False),
])
def test_validate_decimal(value, expected):
    assert ValidationUtility.validate_decimal(value) == expected


def test_display_attributes(capsys, user):
    user.display_attributes()
    captured = capsys.readouterr()
    assert (
                'Login: test_login\nEmail: test@example.com\nPhone:
1234567890\nName: John\nSurname: Doe\nRole: user\n') in captured.out


@patch('builtins.input', side_effect=["newemail@example.com", "", "John",
""])
def test_update_attributes(mock_input):
    db_mock = MagicMock()
    user = User(1, 'testuser', 'password', 'test@example.com',
'1234567890', 'Jane', 'Doe', 'customer')
    user.update_attributes(db_mock)
    assert user.email == "newemail@example.com"
    assert user.name == "John"



@patch('builtins.input', side_effect=["Test City", "Test Street", "123",
"12345", "Test Country"])
def test_insert_new_delivery(mock_input):
    db_mock = MagicMock()
    delivery = Delivery()
    delivery.insert_new_delivery(db_mock)


def test_fetch_orders_by_user():
    db_mock = MagicMock()
    user_id = 1
    expected_orders = [
        (1, 100, 'completed', 'card', '2024-01-01', 'City', 'Street', 123,
'12345', 'Country', 'Cotton', 'M', 'Male',
         50.0, 'Summer Collection', '2024-01-01', '2024-06-01')
    ]
    db_mock.cursor.fetchall.return_value = expected_orders
    Order.fetch_orders_by_user(db_mock, user_id)
    assert user_id in Order.orders


def test_cancel_order_success():
    db_mock = MagicMock()
    order_id = 1
```

```
    Order.cancel_order(db_mock, order_id)
    db_mock.cursor.callproc.assert_called_with("CancelOrder", [order_id])
    db_mock.conn.commit.assert_called_once()


@patch('builtins.input', side_effect=["new_user33", "password123"])
def test_complete_order_workflow(mock_input):
    db_mock = MagicMock()
    db_mock.cursor.fetchone.side_effect = [
        ("3", "new_user33", "password123", "new.user@example.com", "123-
456-7890", "Bob", "Doe", "customer"),
    ]

    app = UI_App()
    app.db = db_mock
    assert app.login()
    db_mock.cursor.fetchone.side_effect = [
        (Decimal("60.00"), 3, 3, 3),
    ]
    Order.add_new_order(db_mock, Decimal("60.00"), 3, 3, 3)
    db_mock.cursor.callproc.assert_called_with("addOrder",
[Decimal("60.00"), 3, 3, 3])
```