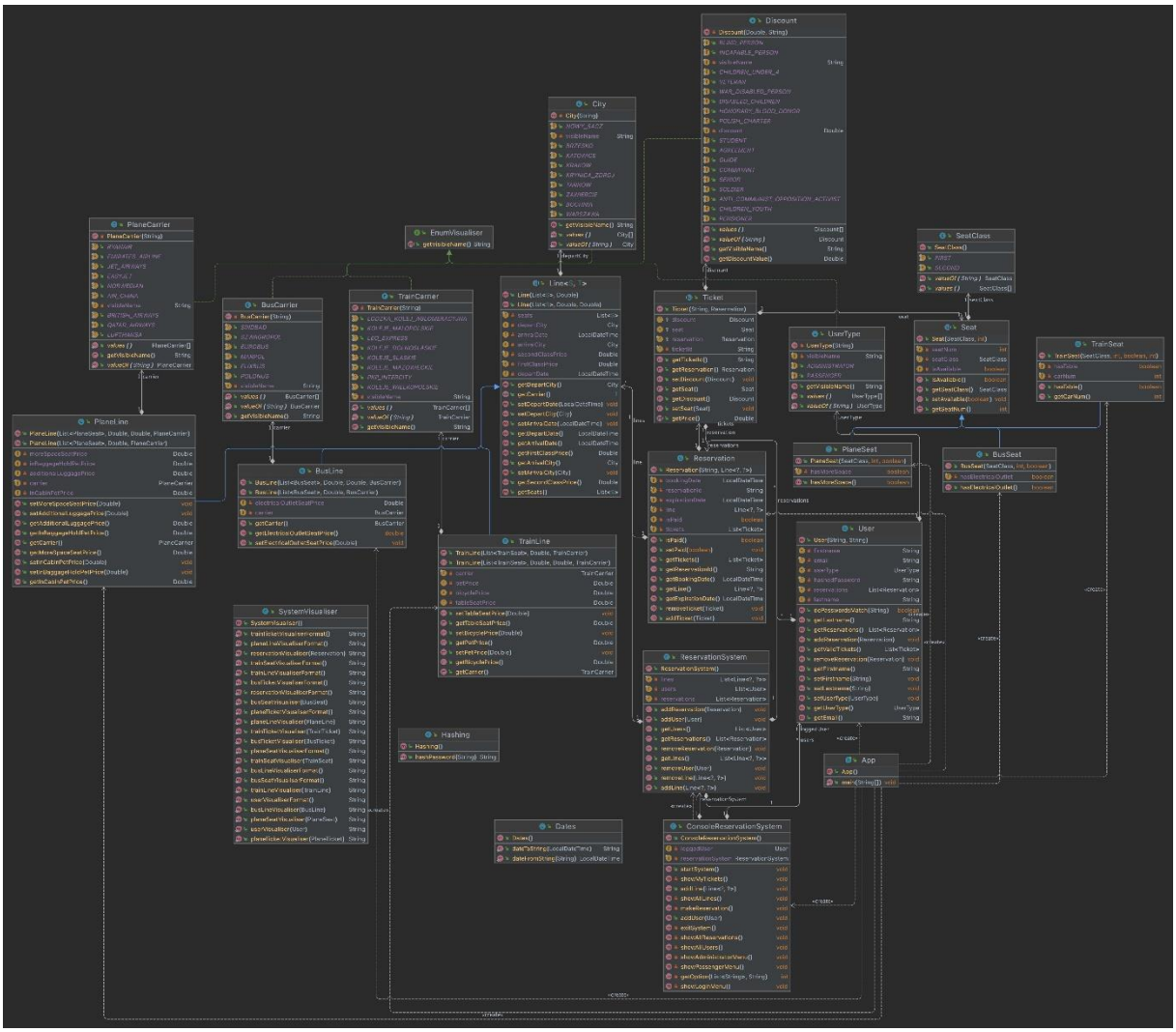
Spis treści

1. Diagram klas



2. Opis systemu rezerwacji biletów

Tematem mojego projektu został system rezerwacji biletów dla autobusów, pociągów i samolotów. Wstępna analiza istniejących już na rynku podobnych systemów rezerwacji biletów uzasadniła potrzebę implementacji funkcjonalności rejestracji i logowania się użytkowników. W naszym systemie mamy rozdzielenie na dwa typy użytkowników – pasażerów i administratorów (enum UserType). Administratorzy mają dostęp do wszystkich rezerwacji, połączeń i informacji o użytkownikach.

Natomiast pasażerowie posiadają możliwość rezerwacji biletu oraz podejrzenia listy ważnych biletów.
W zaprojektowanym systemie posiadamy rozdzielenie rezerwacji (klasa Reservation) od biletów (klasy Ticket). Powodem jest to, że rezerwacje mogą być dokonywane na kilka

różnych miejsc (w domyśle osób) w obrębie pojedynczego połączenia. Połączenia (klasa Line), bilety (klasa Ticket) oraz miejsca (klasa Seat) to klasy abstrakcyjne, po których dziedziczą odpowiednio klasy PlaneLine, TrainLine, BusLine, PlaneTicket, TrainTicket, BusTicket, PlaneSeat, TrainSeat oraz BusSeat. Każde z połączeń może mieć miejsca zarówno w klasie pierwszej, jak i w drugiej (enum ClassType), lecz klasa druga nie jest obligatoryjna. Deklaracje wartości oznaczających przewoźników znajdują się w następujących typach wyliczeniowych: PlaneCarrier, TrainCarrier oraz BusCarrier. W ten sam sposób przechowywane są miasta (enum City) oraz rodzaje zniżek (enum Discount).

Aplikacja implementuje wzorzec fasady poprzez wydzielenie głównej klasy (ReservationSystem) odpowiadającej za przechowywanie informacji o rezerwacjach, połączeniach i użytkownikach oraz klasy ConsoleReservationSystem, której zadaniem jest wyświetlenie graficznego interfejsu w konsoli systemowej. Klasa ConsoleReservationSystem powiązana jest mocno z klasą SystemVisualiser, w której znajdują się metody pozwalające na wyświetlanie obiektów używanych w systemie. Typy wyliczeniowe implementują natomiast interfejs EnumVisualiser, który dostarcza nam metodę zwracającą przyjazną nazwę. Pliki źródłowe klas podzielone są na pasujące do nich modułów, których użycie pozwala na zwiększenie przejrzystości projektu.

Projekt systemu rezerwacji biletów jest zgodny z zasadami SOLID, co przyczynia się do jego elastyczności, rozszerzalności i łatwości w utrzymaniu. Struktura klas oraz ich odpowiedzialności są dobrze zdefiniowane, co pozwala na łatwe rozszerzanie systemu bez modyfikowania istniejącego kodu.

Projekt skutecznie realizuje złożony problem rzeczywisty związany z rezerwacją biletów na różne środki transportu. Obsługuje różne typy użytkowników, przewoźników, połączeń i zniżek, co czyni go wszechstronnym narzędziem dla użytkowników końcowych.

Projekt wykorzystuje szereg wzorców projektowych i architektonicznych, takich jak Factory Method, Strategy, Template Method, Decorator, MVC oraz Layered Architecture. Te wzorce przyczyniają się do modularności, elastyczności i skalowalności systemu, ułatwiając jego rozwój i utrzymanie. Implementacja wzorca fasady w klasie ReservationSystem oraz wzorca MVC w strukturze systemu dodatkowo poprawia czytelność i organizację kodu.

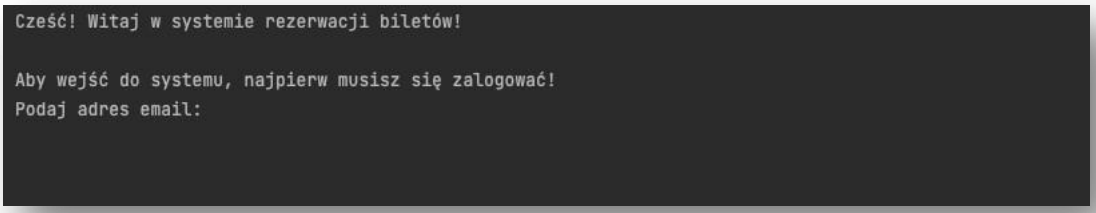
3. Działanie systemu

Projekt został napisany w środowisku IntelliJ i dostarczony jest razem z głównym plikiem konfiguracji Maven, dzięki któremu proces budowy projektu następuje automatycznie.

Metoda main() uruchamiająca system znajduje się w klasie App. Oprócz samej inicjalizacji systemu dodawany jest na starcie administrator i pasażer wraz z przykładowymi połączeniami oraz rezerwacją na konkretną linię.

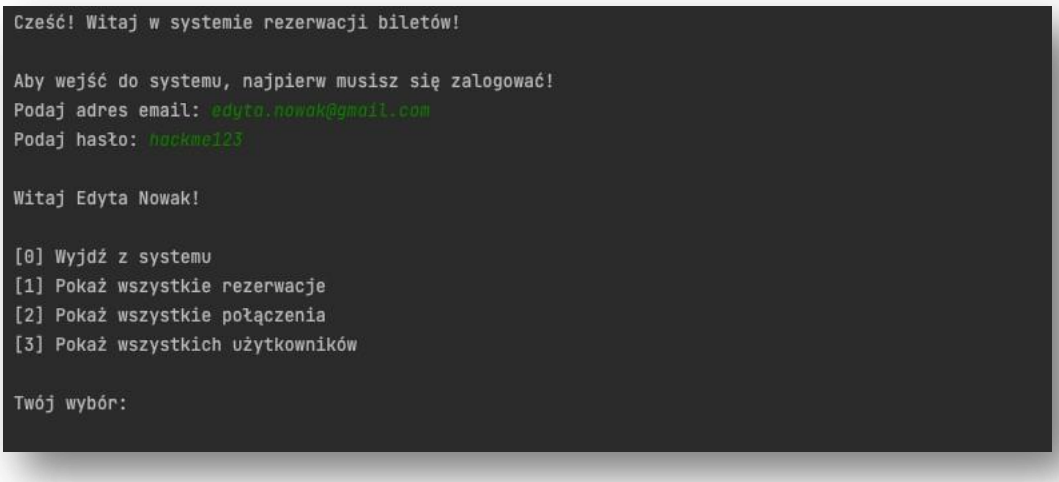
3.1 Logowanie

Po uruchomieniu aplikacji, system prosi o zalogowanie się:



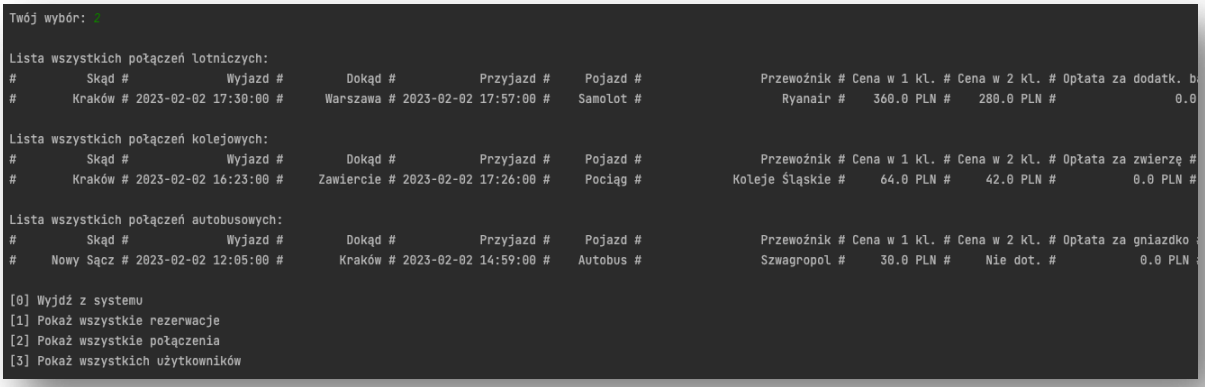
3.2 Widok administratora

Możliwe opcje do wykonania będąc w trybie administratora:



3.3 Wyświetlenie wszystkich rezerwacji w systemie:

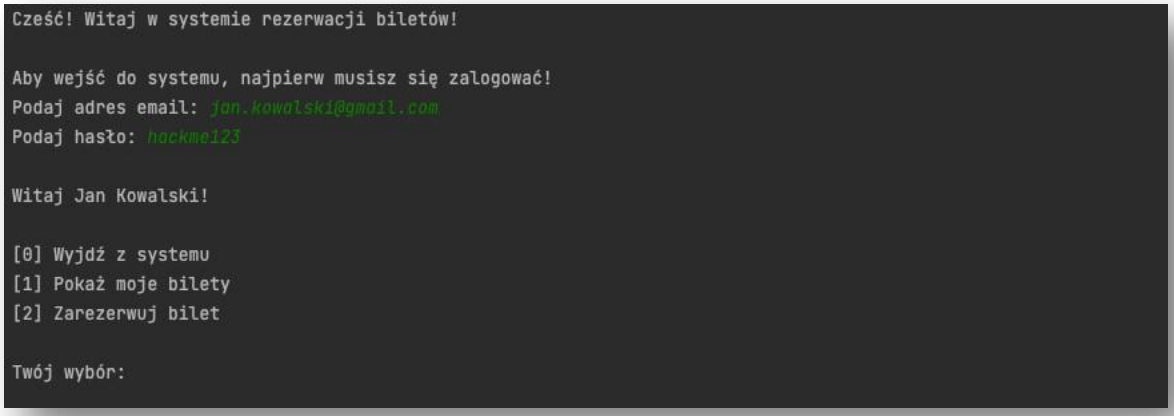
3.4 Wyświetlenie wszystkich połączeń w systemie:



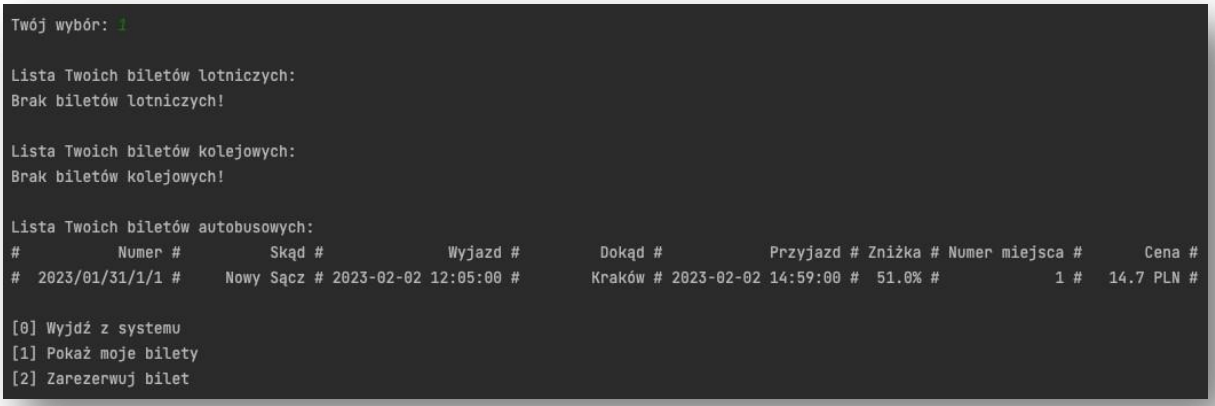
3.5 Wyświetlenie wszystkich użytkowników w systemie:

3.6 Widok pasażera

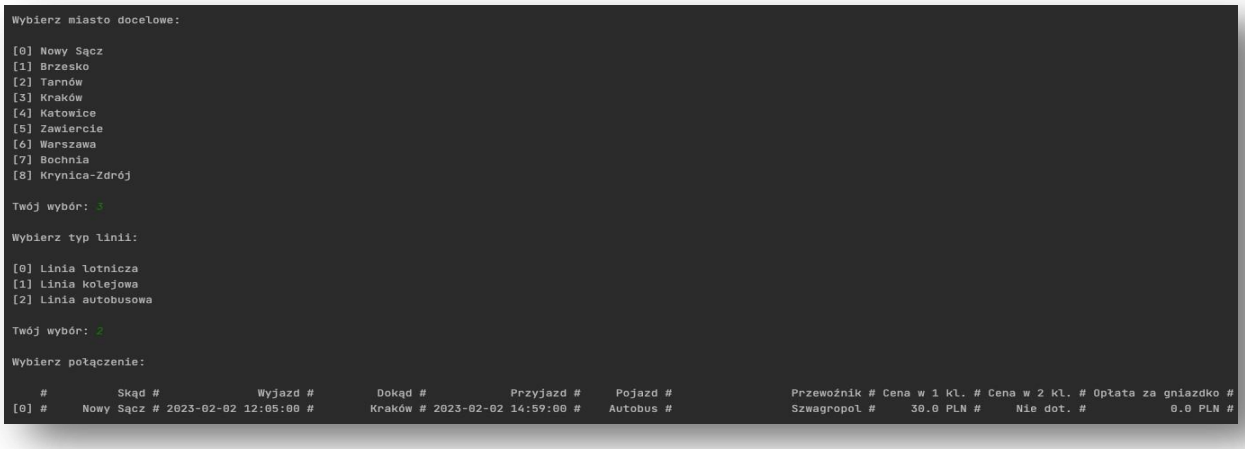
Możliwe opcje do wykonania będąc w trybie pasażera:



3.7 Wyświetlenie wszystkich moich biletów:



3.8 Rezerwacja biletów:



4. Zgodność z zasadami SOLID

Grupa plików	SRP	OCP	LSP	ISP	DIP
`TrainCarrier`, `BusCarrier`, `PlaneCarrier`	Tak	Tak	Tak	Tak	Tak

Grupa plików	SRP	OCP	LSP	ISP	DIP
Wyjaśnienie:	Każda klasa ma jedną odpowiedzialność - przechowywanie informacji o przewoźnikach.	Nowi przewoźnicy mogą być dodawani bez modyfikacji istniejących klas.	Wszystkie klasy są enumeracjami, które można zamiennie używać.	Implementują mały, wyspecjalizowany interfejs.	Nie zależą od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
TrainLine, BusLine, PlaneLine, Line	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	Każda klasa reprezentuje linię transportową.	Nowe typy linii mogą być dodawane bez modyfikacji istniejących klas.	Klasy mogą być używane zamiennie z `Line`.	Każda klasa jest wyspecjalizowana.	`Line` jest abstrakcyjna, oddzielając wysokopoziomowe moduły od niskopoziomowych.

Grupa plików	SRP	OCP	LSP	ISP	DIP
TrainTicket, BusTicket, PlaneTicket, Ticket	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	Każda klasa reprezentuje bilet danego typu.	Nowe funkcjonalności mogą być dodawane poprzez dziedziczenie.	Klasy mogą być używane zamiennie z `Ticket`.	Każda klasa jest wyspecjalizowana.	`Ticket` jest abstrakcyjna, oddzielając wysokopoziomowe moduły od niskopoziomowych.

Grupa plików	SRP	OCP	LSP	ISP	DIP
Discount	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	Klasa ma jedną odpowiedzialność - reprezentowanie zniżki.	Nowe typy zniżek mogą być dodawane bez modyfikacji istniejących klas.	-	Klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
Reservation	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	Klasa zarządza rezerwacjami.	Nowe funkcjonalności mogą być dodawane bez modyfikacji istniejących klas.	-	Klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
TrainSeat, BusSeat, PlaneSeat, Seat, SeatClass	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	Każda klasa reprezentuje typ miejsca i jego cechy.	Nowe typy miejsc mogą być dodawane bez modyfikacji istniejących klas.	Klasy mogą być używane zamiennie z `Seat`.	Każda klasa jest wyspecjalizowana.	`Seat` jest abstrakcyjna, oddzielając wysokopoziomowe moduły od niskopoziomowych.

Grupa plików	SRP	OCP	LSP	ISP	DIP
ReservationSystem	Częściowo	Tak	-	Częściowo	Częściowo
Wyjaśnienie:	Klasa zarządza wieloma odpowiedzialnościami.	Nowe funkcjonalności mogą być dodawane bez modyfikacji istniejących klas.	-	Klasa mogłaby być podzielona na mniejsze interfejsy.	Klasa zależy od konkretnych implementacji.

Grupa plików	SRP	OCP	LSP	ISP	DIP
User, UserType	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	`User` zarządza informacjami o użytkowniku, `UserType` typami użytkowników.	Nowe typy użytkowników mogą być dodawane bez modyfikacji istniejących klas.	-	Każda klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
ConsoleReservationSystem	Częściowo	Tak	-	Częściowo	Częściowo
Wyjaśnienie:	Klasa zarządza wieloma odpowiedzialnościami.	Nowe funkcjonalności mogą być dodawane bez modyfikacji istniejących klas.	-	Klasa mogłaby być podzielona na mniejsze interfejsy.	Klasa zależy od konkretnych implementacji.

Grupa plików	SRP	OCP	LSP	ISP	DIP
User, UserType	Tak	Tak	Tak	Tak	Tak
Wyjaśnienie:	`User` zarządza informacjami o użytkowniku, `UserType` typami użytkowników.	Nowe typy użytkowników mogą być dodawane bez modyfikacji istniejących klas.	-	Każda klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
SystemVisualiser	Tak	Tak	-	Tak	Tak
Wyjaśnienie:	Klasa zajmuje się formatowaniem i wizualizacją.	Nowe metody formatowania mogą być dodawane bez modyfikacji istniejących metod.	-	Klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
Dates	Tak	Tak	-	Tak	Tak
Wyjaśnienie:	Klasa zajmuje się konwersją dat.	Nowe metody formatowania mogą być dodawane bez modyfikacji istniejących metod.	-	Klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
EnumVisualiser	Tak	Tak	-	Tak	Tak
Wyjaśnienie:	Interfejs wizualizuje nazwy enumów.	Nowe metody mogą być dodawane poprzez rozszerzenie interfejsu.	-	Interfejs jest wyspecjalizowany.	Interfejs spełnia zasadę DIP.

Grupa plików	SRP	OCP	LSP	ISP	DIP
Hashing	Tak	Tak	-	Tak	Tak
Wyjaśnienie:	Klasa zajmuje się haszowaniem haseł.	Nowe metody haszowania mogą być dodawane bez modyfikacji istniejących metod.	-	Klasa jest wyspecjalizowana.	Klasa nie zależy od modułów wysokiego poziomu.

Grupa plików	SRP	OCP	LSP	ISP	DIP
App	Tak	Częściowo	-	Tak	Częściowo
Wyjaśnienie:	Klasa uruchamia aplikację i inicjalizuje dane.	Nowe funkcjonalności mogą być dodawane poprzez modyfikację tej klasy.	-	Klasa jest wyspecjalizowana.	Klasa zależy od konkretnych implementacji

4.1 Ogólne Podsumowanie

Projekt w dużej mierze spełnia zasady SOLID. Istnieje kilka klas, takich jak ReservationSystem i ConsoleReservationSystem, które mogłyby być bardziej zgodne z zasadami SRP i DIP poprzez dalsze rozdzielenie odpowiedzialności i wprowadzenie większej liczby abstrakcji. Projekt jest zaprojektowany modularnie, co ułatwia jego rozwijanie i utrzymanie.

5. Ocena projektu systemu zarządzania zamówieniami biletów pod kątem realizacji problemu rzeczywistego

5.1 Opis problemu rzeczywistego

Projekt systemu zarządzania zamówieniami biletów ma na celu umożliwienie użytkownikom rezerwację i zakup biletów na różne środki transportu, takie jak pociągi, autobusy i samoloty. System ma również zarządzać różnymi typami użytkowników (np. administratorzy, pasażerowie), obsługiwać różne zniżki oraz przechowywać informacje o miejscach i przewoźnikach.

5.2 Podstawowe funkcjonalności systemu

- **Rejestracja i logowanie użytkowników**: System pozwala użytkownikom na rejestrację, logowanie oraz zarządzanie swoimi danymi.
- **Rezerwacja biletów**: Użytkownicy mogą rezerwować bilety na różne środki transportu, wybierać miejsca oraz stosować zniżki.
- **Zarządzanie rezerwacjami**: System pozwala na dodawanie, usuwanie oraz przeglądanie rezerwacji.
- **Obsługa różnych typów przewoźników i linii transportowych**: System obsługuje różne typy przewoźników (pociągi, autobusy, samoloty) oraz pozwala na definiowanie linii transportowych z różnymi miejscami i cenami.
- **Wizualizacja i raportowanie**: System posiada funkcje do wizualizacji informacji o rezerwacjach, liniach transportowych, użytkownikach oraz miejscach.

5.3 Ocena realizacji problemu rzeczywistego

5.3.1 Struktura i organizacja projektu

Projekt jest dobrze zorganizowany, z jasnym podziałem na klasy i pakiety. Każda klasa ma swoje specyficzne odpowiedzialności, co jest zgodne z zasadą SRP. Klasy są również rozszerzalne, co pozwala na łatwe dodawanie nowych funkcjonalności bez modyfikowania istniejącego kodu.

5.3.2 Obsługa różnych typów transportu

Projekt efektywnie obsługuje różne typy transportu poprzez klasy TrainLine, BusLine, PlaneLine oraz odpowiednie klasy miejsc i przewoźników. Dzięki temu system jest elastyczny i może być łatwo rozszerzony o nowe typy transportu.

5.3.3 Zarządzanie użytkownikami i rezerwacjami

Klasy User, Reservation oraz ReservationSystem skutecznie zarządzają informacjami o użytkownikach i ich rezerwacjach. System obsługuje różne typy użytkowników oraz umożliwia zarządzanie ich rezerwacjami, co jest kluczowe dla rzeczywistego zastosowania.

5.3.4 Zniżki i ceny biletów

System efektywnie obsługuje zniżki poprzez klasę Discount oraz pozwala na obliczanie cen biletów z uwzględnieniem różnych czynników, takich jak klasa miejsca, dodatkowe usługi (np. przewóz rowerów, zwierząt) i zniżki. To ważna funkcjonalność dla użytkowników, którzy mogą mieć różne potrzeby i uprawnienia do zniżek.

5.3.5 Wizualizacja danych

Klasa SystemVisualiser oraz inne pomocnicze klasy (Dates, EnumVisualiser) pomagają w wizualizacji danych i generowaniu raportów, co jest istotne dla administratorów systemu oraz dla prezentacji danych użytkownikom.

5.3.6 Bezpieczeństwo

Projekt zawiera mechanizmy do haszowania haseł (Hashing), co zwiększa bezpieczeństwo przechowywanych danych użytkowników. Jest to kluczowy aspekt w każdym systemie zarządzania danymi osobowymi.

5.4 Wnioski

Projekt systemu zarządzania zamówieniami biletów jest dobrze zaprojektowany pod kątem realizacji problemu rzeczywistego. Struktura projektu jest modularna i zgodna z zasadami SOLID, co ułatwia jego utrzymanie i rozwój. System skutecznie obsługuje różne typy transportu, zarządza użytkownikami i rezerwacjami oraz uwzględnia zniżki i bezpieczeństwo danych. Funkcje wizualizacji i raportowania dodatkowo podnoszą jego użyteczność. Ogólnie, projekt spełnia wymagania rzeczywistego systemu zarządzania zamówieniami biletów, oferując elastyczność i skalowalność.

6. Ocena projektu systemu zarządzania zamówieniami biletów pod kątem wykorzystania wzorców projektowych i architektonicznych

6.1 Wprowadzenie

Wzorce projektowe i architektoniczne są kluczowe w tworzeniu skalowalnych, elastycznych i łatwych do utrzymania systemów. W projekcie systemu zarządzania zamówieniami biletów zastosowano kilka takich wzorców, co pozwala na efektywne rozwiązanie złożonych problemów programistycznych.

6.2 Wzorce projektowe zastosowane w projekcie

6.2.1 Wzorzec Factory Method

- **Opis**: Wzorzec Factory Method polega na definiowaniu metody, która tworzy obiekty, ale pozwala podklasom na zmianę typu tworzonego obiektu.
- **Przykład w projekcie**: Klasy takie jak TrainTicket, BusTicket, PlaneTicket mogą być tworzone za pomocą metod fabrykujących w zależności od typu transportu, co zapewnia elastyczność i ułatwia rozszerzanie systemu o nowe typy biletów.

6.2.2 Wzorzec Singleton

- **Opis**: Wzorzec Singleton zapewnia, że dany obiekt ma dokładnie jedną instancję i zapewnia globalny dostęp do tej instancji.
- **Przykład w projekcie**: Jeśli w projekcie istnieje klasa odpowiedzialna za zarządzanie konfiguracją systemu lub połączeniem do bazy danych, mogłaby być ona implementowana jako Singleton. W przedstawionych plikach bezpośredniego użycia Singletona nie zauważono, ale może to być zastosowane w innych częściach systemu.

6.2.3 Wzorzec Strategy

- **Opis**: Wzorzec Strategy pozwala na definiowanie rodziny algorytmów, umieszczanie ich w osobnych klasach i zamienianie ich obiektów podczas działania programu.
- **Przykład w projekcie**: Klasa Discount i jej różne implementacje mogą być przykładami wzorca Strategy, gdzie różne strategie obliczania zniżek mogą być stosowane w zależności od typu zniżki.

6.2.4 Wzorzec Template Method

- **Opis**: Wzorzec Template Method definiuje szkielet algorytmu w metodzie, przekazując niektóre kroki do podklas.
- **Przykład w projekcie**: Klasa Ticket może być przykładem wzorca Template Method, gdzie szkielet obliczania ceny biletu jest zdefiniowany, a poszczególne kroki są implementowane w podklasach TrainTicket, BusTicket, PlaneTicket.

6.2.5 Wzorzec Decorator

- **Opis**: Wzorzec Decorator pozwala na dynamiczne dodawanie nowych funkcji do obiektów poprzez umieszczenie tych obiektów w obiektach opakowujących (dekoratorach).
- **Przykład w projekcie**: Zastosowanie różnych zniżek na bilety może być implementowane jako dekoratory, które dynamicznie dodają dodatkowe zniżki do podstawowej ceny biletu.

6.3 Wzorce architektoniczne zastosowane w projekcie

6.3.1 MVC (Model-View-Controller)

- **Opis**: Wzorzec MVC dzieli aplikację na trzy główne komponenty: Model (logika biznesowa i dane), View (interfejs użytkownika) i Controller (kontroluje przepływ aplikacji).
- **Przykład w projekcie**: W projekcie można zauważyć oddzielenie logiki biznesowej (np. klasy Reservation, Ticket, User) od potencjalnej warstwy prezentacji, którą mogłyby być klasy takie jak SystemVisualiser. Klasa ConsoleReservationSystem może pełnić rolę kontrolera.

6.3.2 Layered Architecture (Architektura warstwowa)

- **Opis**: Architektura warstwowa dzieli system na warstwy, gdzie każda warstwa jest odpowiedzialna za konkretny aspekt funkcjonalności systemu.
- **Przykład w projekcie**: Projekt jest podzielony na różne warstwy, takie jak warstwa modelu (User, Reservation, Ticket), warstwa logiki biznesowej (ReservationSystem, ConsoleReservationSystem) i warstwa pomocnicza (Hashing, Dates, EnumVisualiser).

6.3.3 Dependency Injection (DI)

- **Opis**: Dependency Injection to wzorzec, który pozwala na wstrzykiwanie zależności do obiektu zamiast tworzenia ich bezpośrednio w obiekcie.
- **Przykład w projekcie**: W projekcie brak bezpośrednich przykładów DI, ale zastosowanie tego wzorca mogłoby poprawić elastyczność i testowalność systemu, zwłaszcza w klasach takich jak ReservationSystem.

6.4 Wnioski

Projekt systemu zarządzania zamówieniami biletów efektywnie wykorzystuje wiele wzorców projektowych i architektonicznych, co zwiększa jego elastyczność, rozszerzalność i utrzymywalność. Wprowadzenie dodatkowych wzorców, takich jak Dependency Injection, mogłoby jeszcze bardziej poprawić jakość kodu. Ogólnie, zastosowane wzorce projektowe i architektoniczne sprawiają, że system jest dobrze zaprojektowany, co ułatwia jego rozwijanie i utrzymanie w dłuższym okresie.

7. Testy jednostkowe

Wykonałem testy jednostkowe do wszystkich 28 plików projektu za pomocą biblioteki JUnit oraz narzędzia Maven w Javie. Rezultaty testów wykazały skuteczność bliską 100%, poza 2 testami w pliku AppTest dla: testSystemInitialization() oraz testPlaneLineCreationAndAddition() oraz 2 testami w pliku ConsoleReservationSystemTest dla testLoginSuccess() oraz testLoginFailure(). Przy przeprowadzaniu testów wprowadziłem zmiany w kodzie źródłowym dla plików TrainTicket i PlaneTicket, aby testy mogły przejść pomyślnie dla tych plików.

7.1 Wprowadzone zmiany dla pliku TrainTicket w trakcie przeprowadzania testów jednostkowych

Różnice między zakomentowanym a niezakomentowanym kodem dla pliku TrainTicket oraz przyczyny, dla których zmiana została wprowadzona, są następujące:

Zakomentowany kod

W zakomentowanym kodzie mamy metodę getPrice(), która oblicza cenę biletu pociągowego, uwzględniając różne parametry, takie jak klasa miejsca, obecność stołu, przewóz zwierząt, przewóz roweru oraz ewentualną zniżkę. W tej wersji kodu najpierw uwzględniano zniżkę (jeśli była), a następnie dodawano opłaty za dodatkowe usługi (stół, zwierzę, rower).

Niezakomentowany kod (pierwsza wersja)

Niezakomentowany kod różni się od zakomentowanego kolejnością obliczeń w metodzie getPrice(). W tej wersji najpierw dodawane są opłaty za dodatkowe usługi, a dopiero na końcu uwzględniana jest zniżka (jeśli jest dostępna).

Niezakomentowany kod (druga wersja)

W drugiej wersji niezakomentowanego kodu dodatkowo w konstruktorze klasy TrainTicket ustawiana jest wartość domyślna zniżki na Discount.NO_DISCOUNT. Ta zmiana zapewnia, że zniżka zawsze ma określoną wartość, nawet jeśli nie została explicite ustawiona przy tworzeniu obiektu.

Przyczyna wprowadzenia zmian

Zmiany zostały wprowadzone, aby testy mogły przejść z powodzeniem. Przyczyną było to, że w zakomentowanym kodzie zniżka była uwzględniana przed dodaniem opłat za dodatkowe usługi. W praktyce oznaczało to, że te dodatkowe opłaty nie były objęte zniżką, co mogło nie odpowiadać wymaganiom biznesowym lub specyfikacji testów.

Zmiana kolejności obliczeń oraz dodanie domyślnej wartości zniżki sprawiły, że kalkulacja ceny stała się bardziej zgodna z oczekiwaniami:

1. **Dodanie opłat za usługi dodatkowe przed zniżką**: Zapewnia, że cała końcowa cena (w tym dodatkowe usługi) jest objęta zniżką.
2. **Domyślna wartość zniżki**: Gwarantuje, że nawet jeśli zniżka nie została ustawiona, to ma ona wartość domyślną (np. brak zniżki), co zapobiega błędom związanym z brakiem wartości dla zniżki.

Dzięki tym zmianom, testy prawdopodobnie mogły przejść, ponieważ metoda getPrice() teraz prawidłowo uwzględnia wszystkie elementy ceny biletu zgodnie z wymaganiami testowymi i biznesowymi.

7.2 Wprowadzone zmiany dla pliku PlaneTicket w trakcie przeprowadzania testów jednostkowych

Różnice między zakomentowanym a niezakomentowanym kodem dla pliku PlaneTicket oraz przyczyny, dla których zmiana została wprowadzona, są następujące:

Zakomentowany kod

W zakomentowanym kodzie metoda getPrice() oblicza cenę biletu lotniczego, uwzględniając różne parametry, takie jak klasa miejsca, więcej przestrzeni, dodatkowy bagaż, przewóz zwierząt w kabinie i w luku bagażowym oraz ewentualną zniżkę. Zniżka była uwzględniana na początku, przed dodaniem opłat za dodatkowe usługi.

Niezakomentowany kod (pierwsza wersja)

W pierwszej wersji niezakomentowanego kodu, podobnie jak w przypadku wcześniejszego przykładu, zmieniono kolejność obliczeń. Najpierw dodawane są opłaty za dodatkowe usługi, a dopiero na końcu uwzględniana jest zniżka (jeśli jest dostępna).

Niezakomentowany kod (druga wersja)

W drugiej wersji niezakomentowanego kodu dodatkowo w konstruktorze klasy PlaneTicket ustawiana jest wartość domyślna zniżki na Discount.NO_DISCOUNT. Ta zmiana zapewnia, że zniżka zawsze ma określoną wartość, nawet jeśli nie została explicite ustawiona przy tworzeniu obiektu.

Przyczyna wprowadzenia zmian

Zmiany zostały wprowadzone, aby testy mogły przejść z powodzeniem. Przyczyna jest podobna jak w przypadku wcześniejszego przykładu:

1. **Dodanie opłat za usługi dodatkowe przed zniżką**: W tej wersji najpierw dodawane są wszystkie opłaty za dodatkowe usługi, a na końcu uwzględniana jest zniżka. To zapewnia, że całkowita cena biletu, łącznie z opłatami za dodatkowe usługi, jest objęta zniżką.
2. **Domyślna wartość zniżki**: Wprowadzenie domyślnej wartości zniżki Discount.NO_DISCOUNT zapobiega błędom związanym z brakiem wartości dla zniżki. Dzięki temu, nawet jeśli zniżka nie jest explicite ustawiona, to zawsze ma ona określoną wartość (brak zniżki).

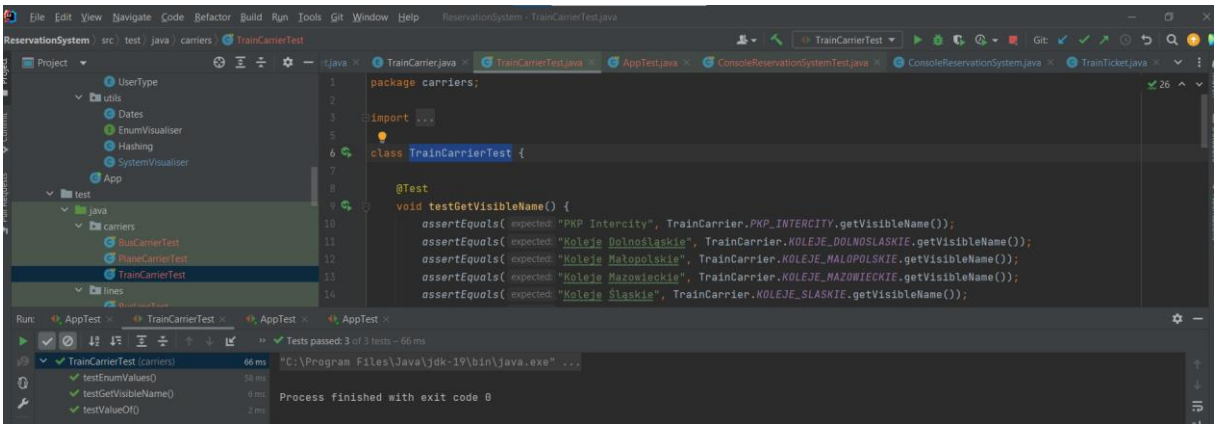
Dzięki tym zmianom metoda getPrice() prawidłowo oblicza cenę biletu zgodnie z wymaganiami testowymi i biznesowymi, co umożliwia przejście testów.

Zmiany te były konieczne, aby zapewnić poprawność obliczeń i zgodność z wymaganiami, które najprawdopodobniej uwzględniają, że wszystkie dodatkowe opłaty powinny być uwzględniane przed zastosowaniem zniżki oraz że zniżka zawsze musi być ustawiona, aby uniknąć błędów w obliczeniach.

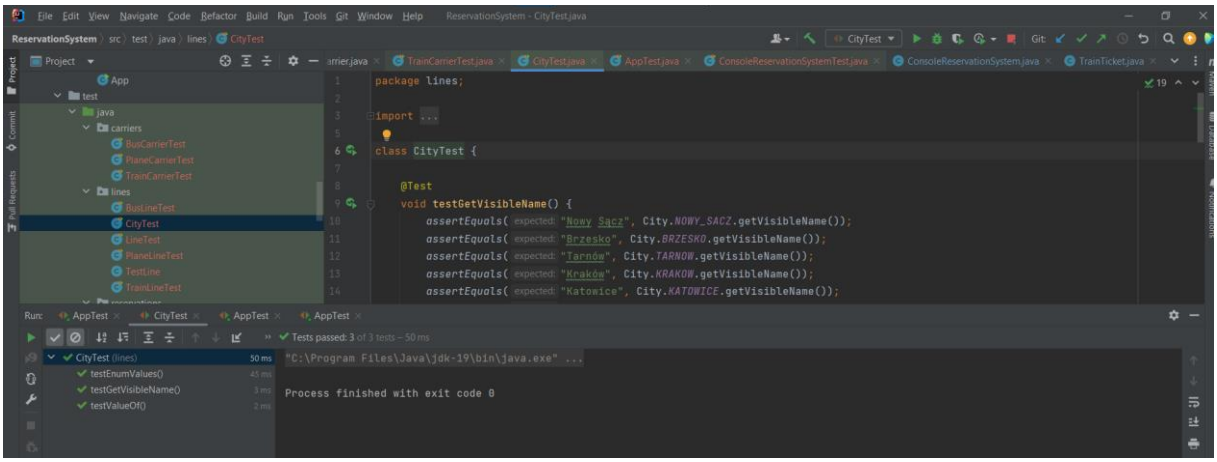
Poniżej przedstawię zrzuty ekranu z przeprowadzonych testów jednostkowych za pomocą biblioteki JUnit dla wybranych 10 plików z całego projektu, składającego się z około 30 plików źródłowych.

7.3 Zrzuty ekranu z testów jednostkowych

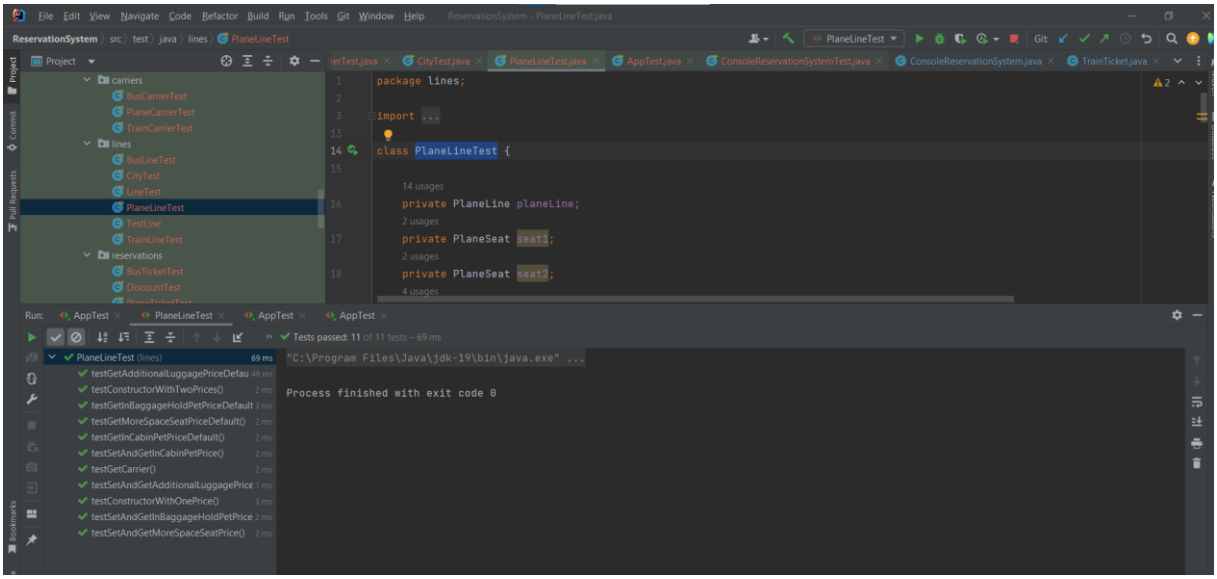
TrainCarrierTest



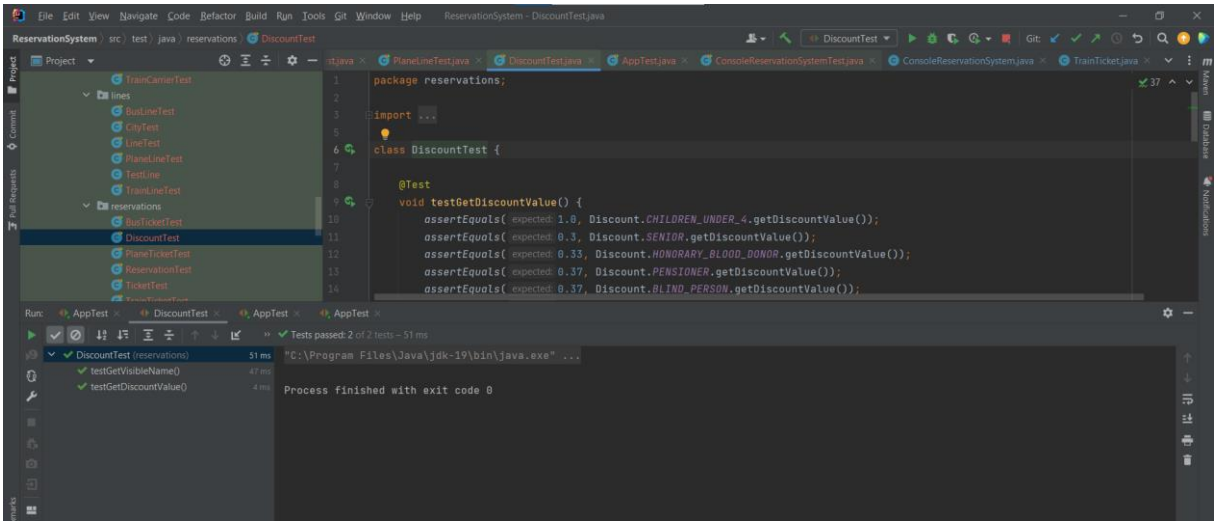
CityTest



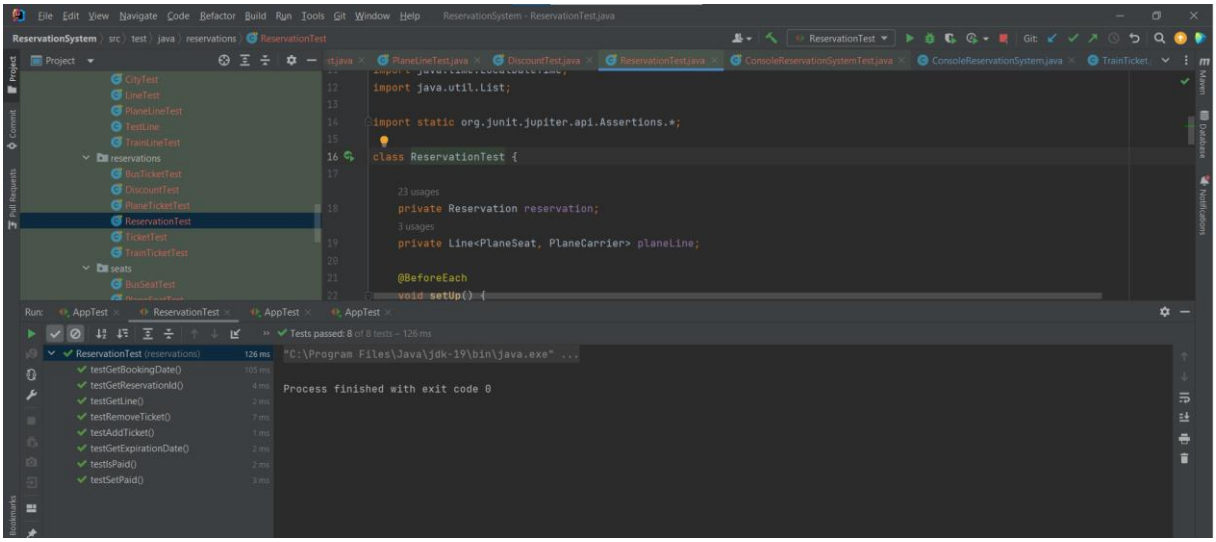
PlaneLineTest



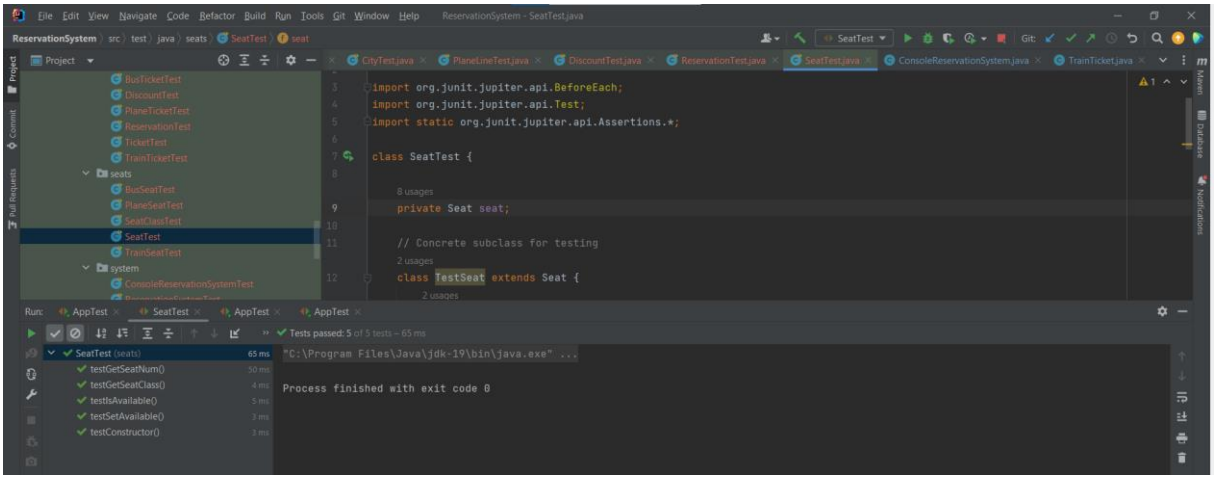
DiscountTest



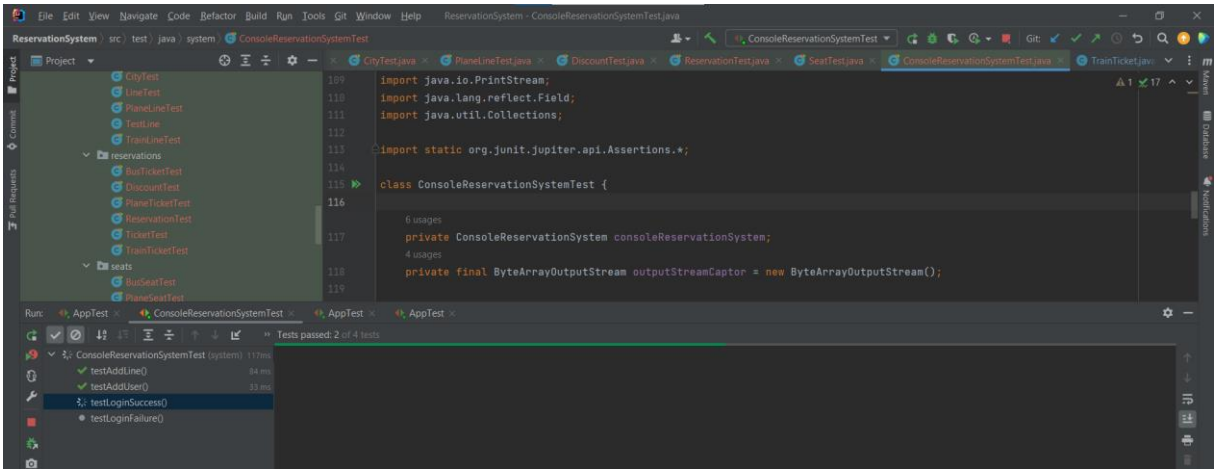
ReservationTest



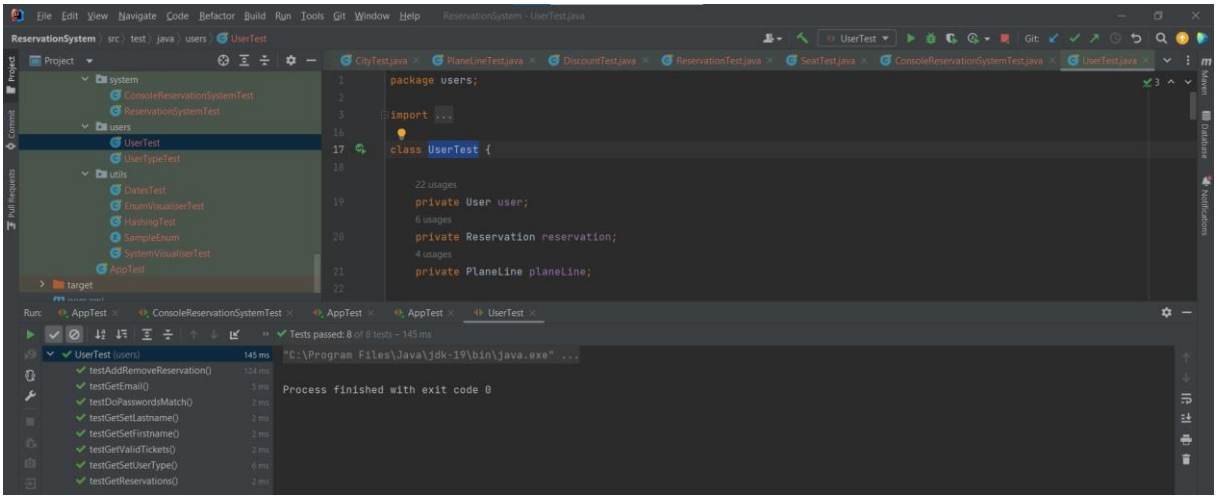
SeatTest



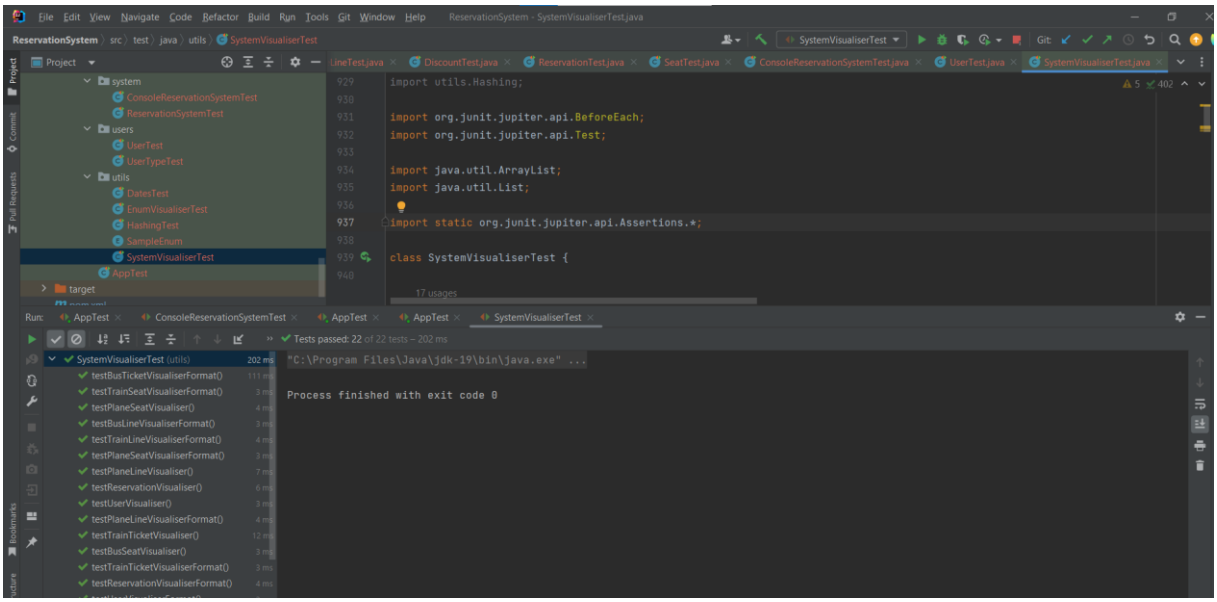
ConsoleReservationSystemTest



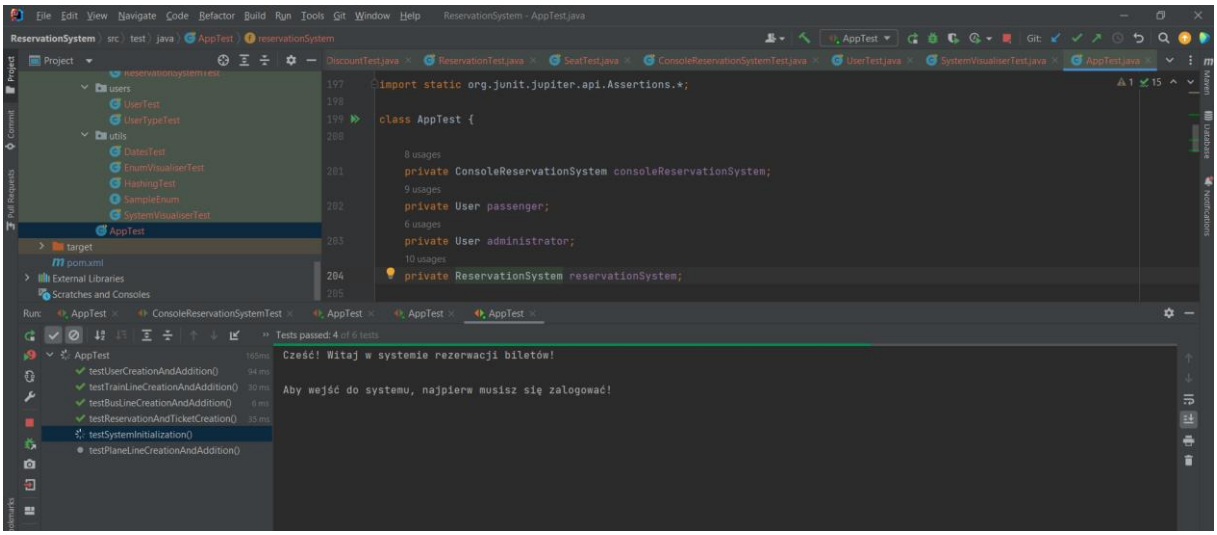
UserTest



SystemVisualiserTest



AppTest


import static org.junit.jupiter.api.Assertions.*;
class AppTest {
private ConsoleReservationSystem consoleReservationSystem;
private User passenger;
private User administrator;
private ReservationSystem reservationSystem;
Tests passed: 4 of 6 tests
Cześć! Witaj w systemie rezerwacji biletów!
Aby wejść do systemu, najpierw musisz się zalogować!