Untold Ticketing App
Analysis and Design Document

Student:Bar Luca-Narcis
Group:30431/1

Table of Contents

1. Requirements Analysis 3
- 1.1 Assignment Specification 3
- 1.2 Functional Requirements 3
- 1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 4

4. UML Sequence Diagrams 7

5. Class Design 8

6. Data Model 10

7. System Testing 11

8. Bibliography 11

1. Requirements Analysis

1.1 Assignment Specification

The application is a Java Desktop app designed for selling tickets to the Untold festival. It has 2 types of users(cashier and admin) who must provide their credentials when opening the app, before being able to use it. The data that they work with is stored in a relational database implemented using PostgreSQL. The tables are for artists, users, concerts, tickets and an intermediary table used to link multiple artists to a single concert.

1.2 Functional Requirements

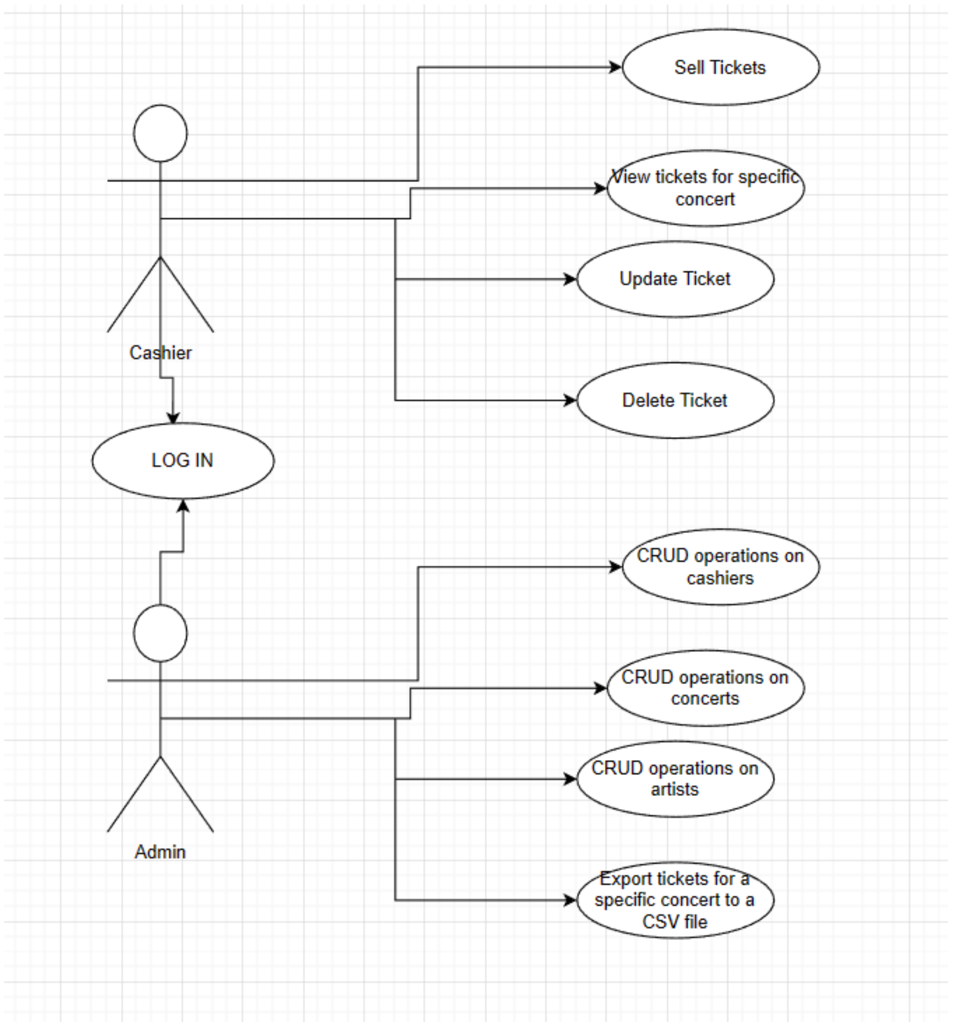
- The admin user can perform the following operations:
    - CRUD on cashiers' information
    - CRUD on the concerts at Untold
    - Export all the tickets that were sold for a certain show
- The cashier user can perform the following operations:
    - Sell tickets to a concert
    - CRUD on tickets
- The system notifies the users when:
    - The number of tickets per concert was exceeded
- Data is stored in a relational database
- Layers architectural pattern is used to organize the application
- User interface

1.3 Non-functional Requirements

-*Responsive GUI*
-*Fast access time*
-*Password encryption for extra security*
-*Portability(can be ran on any device)*
-*Scalable*
-*Local, for now.*

2. Use-Case Model

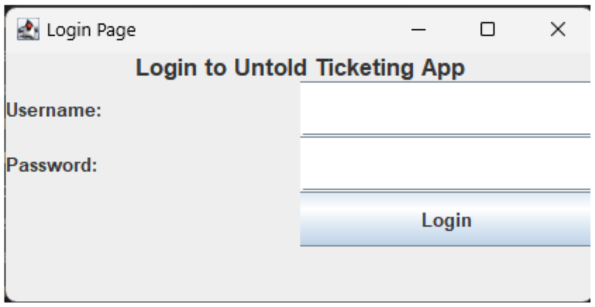
General Use-Cases for the actors of the application:


Sell Tickets
View tickets for specific concert
Update Ticket
Delete Ticket
Cashier
LOG IN
CRUD operations on cashiers
CRUD operations on concerts
CRUD operations on artists
Export tickets for a specific concert to a CSV file
Admin

Use case: Creating a Ticket for a concert/show
Primary actor: Cashier(user)
Main success scenario:
User enters their credentials- > credentials are checked and if they are correct, the loginFrame will open un a CashierFrame since it will get its role from the UserBL method-> User (cashier) can select from the available operations, and in this case

they will select the “Sell Ticket” option and will be prompted to insert the details for the ticket(s). If the operation is executed successfully(i.e. entered data is correct and there are enough tickets), the tickets will be created and added in the tickets table.
Extensions:
-should the user provide some incorrect credentials, they will receive an error
-should the user provide the incorrect name of the concert, they will receive an error
-should there be too few tickets for the operation, the user will again receive an error.

Screenshots for the successful case:
Login screen:


Login Page

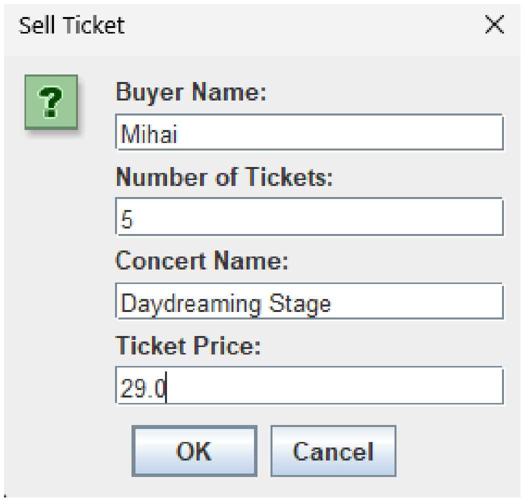
Login to Untold Ticketing App

Username:

Password:

Login

Interface for providing ticket details:


Sell Ticket

Buyer Name: Mihai

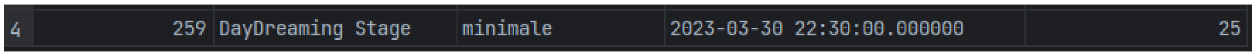
Number of Tickets: 5

Concert Name: Daydreaming Stage

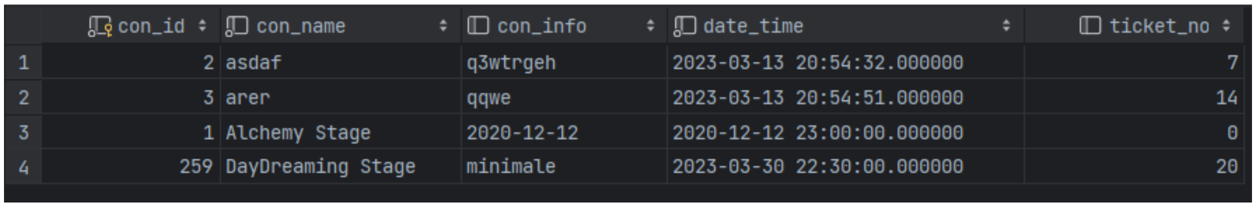
Ticket Price: 29.0

OK Cancel

Before confirming ticket sale:

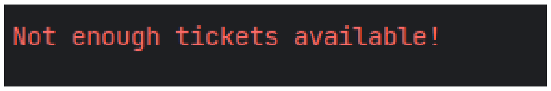

```
4      259 DayDreaming Stage   minimale        2023-03-30 22:30:00.000000        25
```

After confirming ticket sale(we see that the tickets for DayDreaming Stage have gone down):



	con_id	con_name	con_info	date_time	ticket_no
1	2	asdaf	q3wtrgeh	2023-03-13 20:54:32.000000	7
2	3	arer	qqwe	2023-03-13 20:54:51.000000	14
3	1	Alchemy Stage	2020-12-12	2020-12-12 23:00:00.000000	0
4	259	DayDreaming Stage	minimale	2023-03-30 22:30:00.000000	20

Example of error, if we try to sell too many tickets(currently pops up only in the terminal, and no tickets will be sold in this case):



```
Not enough tickets available!
```

3. System Architectural Design

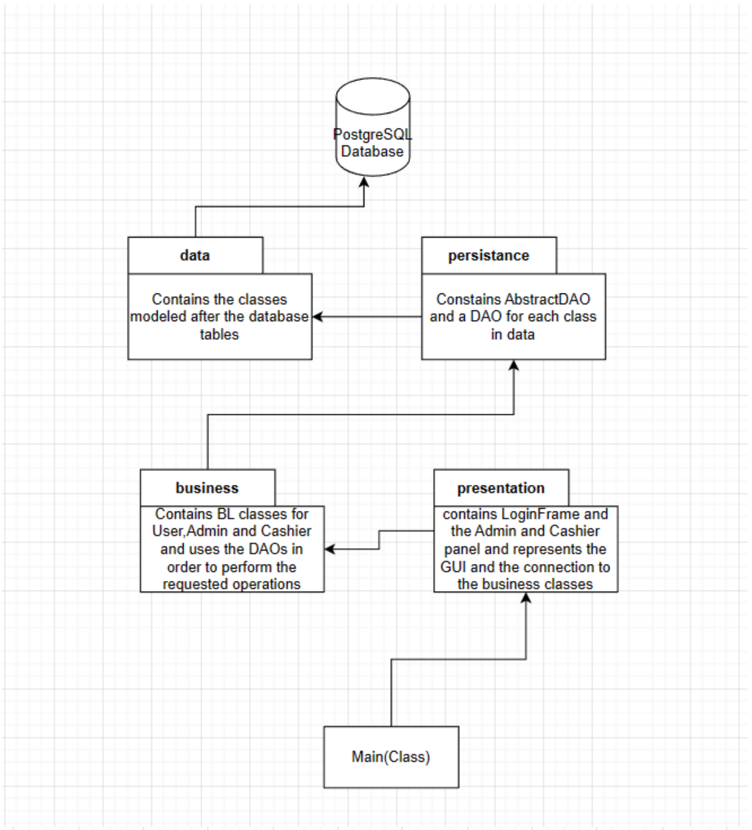
For the project I've used the Layered architecture(i.e. Data layer, business layer and presentation layer).

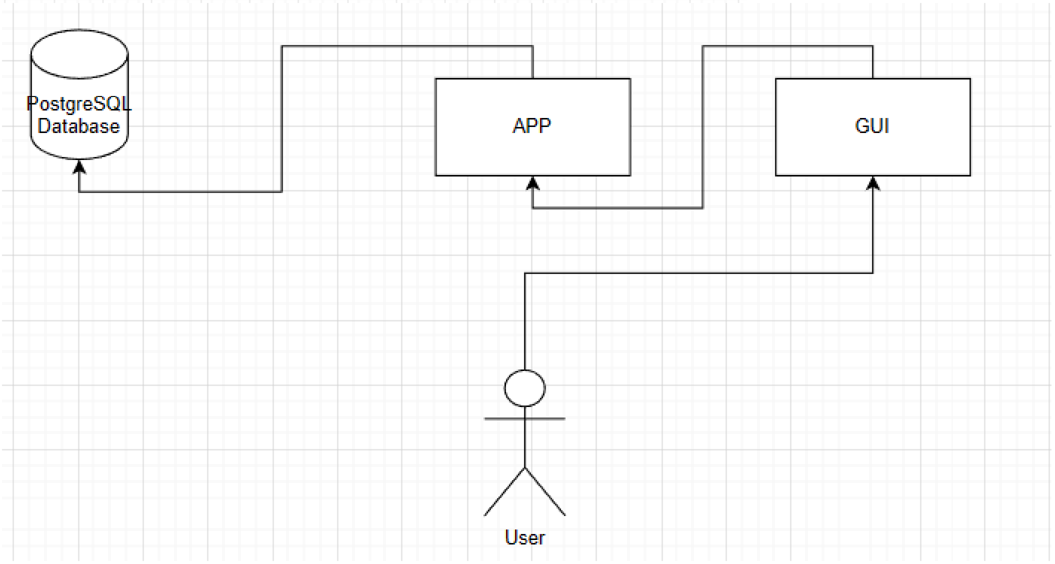
3.1 Architectural Pattern Description

The Layered architecture pattern is one of the most common patter in Java development and, in essence, it implies organizing the data into multiple horizontal layers, each performing a specific role within the application.
In my app, I've split the layers into 3(or 4 rather): The data package is used to create the model classes after the database tables, the persistence layer consists of an AbstractDAO class, which is extended by the DAO classes for each of the classes in the data package and are used to represent the way operations on the database(i.e. the CRUD operations) are performed. The business layer is used to build effective roles within the app, such as the administrator and the cashier BL classes, which are used to represent the available operations for each user type and also a UserBL class which is used for authentication and user type retrieval. And finally, the presentation class is the one responsible for the interface of the app and its functionality(i.e. the action listeners for the buttons).And the DBCon class is responsible with connecting to my PostgreSQL database, using the JDBC driver.(included in the app are the libraries needed for jdbc and for the openCSV library).

3.2 Diagrams

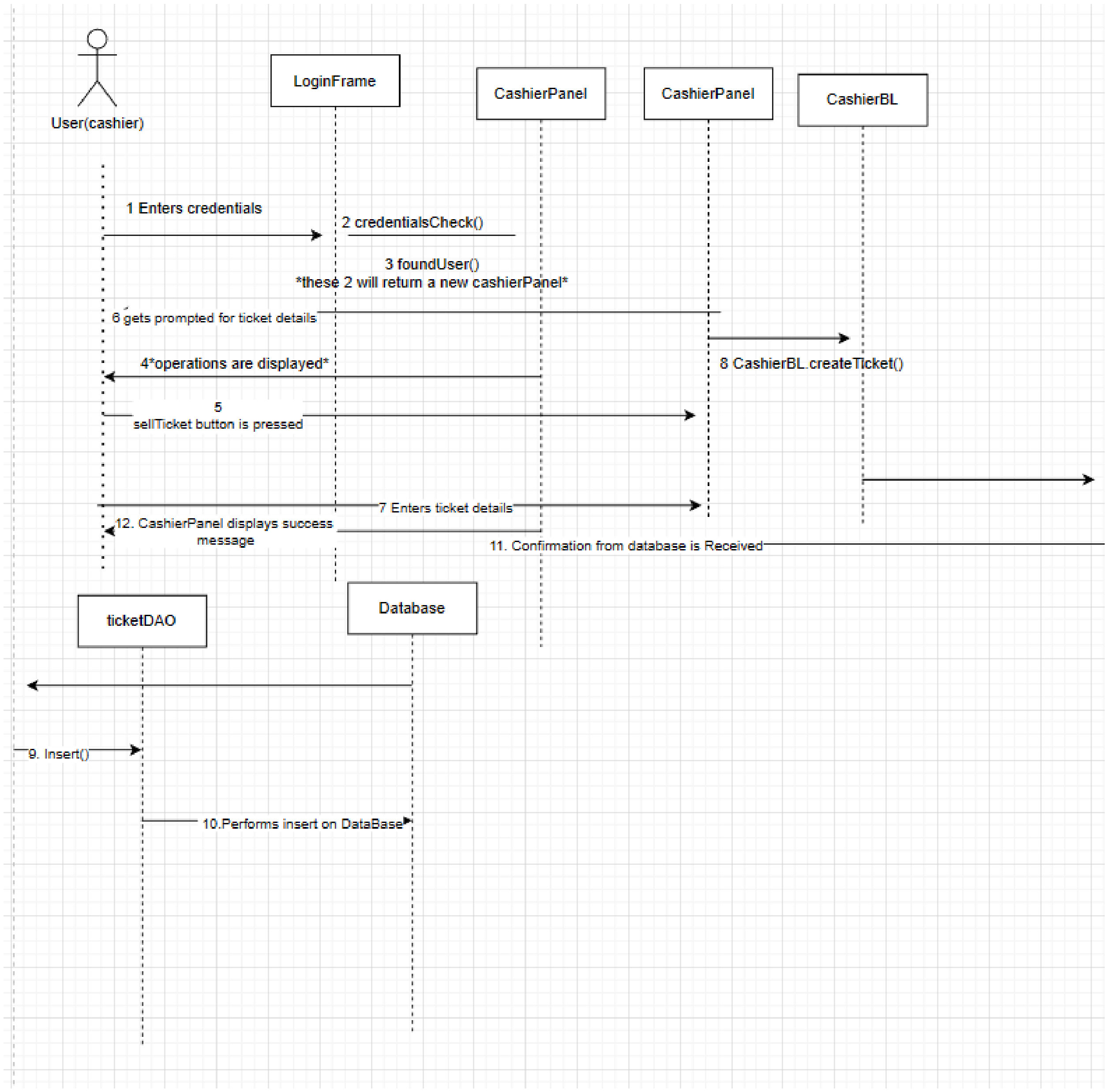
The package modelling of the app:


PostgreSQL Database
data
Contains the classes modeled after the database tables
persistance
Constains AbstractDAO and a DAO for each class in data
business
Contains BL classes for User,Admin and Cashier and uses the DAOs in order to perform the requested operations
presentation
contains LoginFrame and the Admin and Cashier panel and represents the GUI and the connection to the business classes
Main(Class)



4. UML Sequence Diagrams

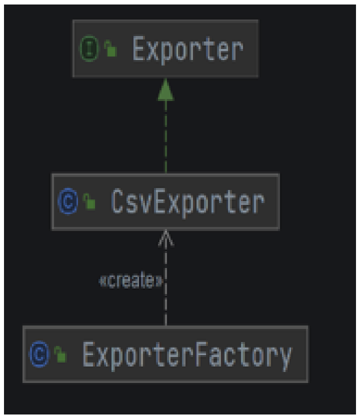
Sequence Diagram for the “Create Ticket” operation which was also described in the Use-Case section:


User(cashier)
LoginFrame
CashierPanel
CashierPanel
CashierBL
1 Enters credentials
2 credentialsCheck()
3 foundUser()
these 2 will return a new cashierPanel
6 gets prompted for ticket details
4*operations are displayed*
8 CashierBL.createTicket()
5
sellTicket button is pressed
7 Enters ticket details
12. CashierPanel displays success message
11. Confirmation from database is Received
ticketDAO
Database
9. Insert()
10.Performs insert on DataBase

5. Class Design

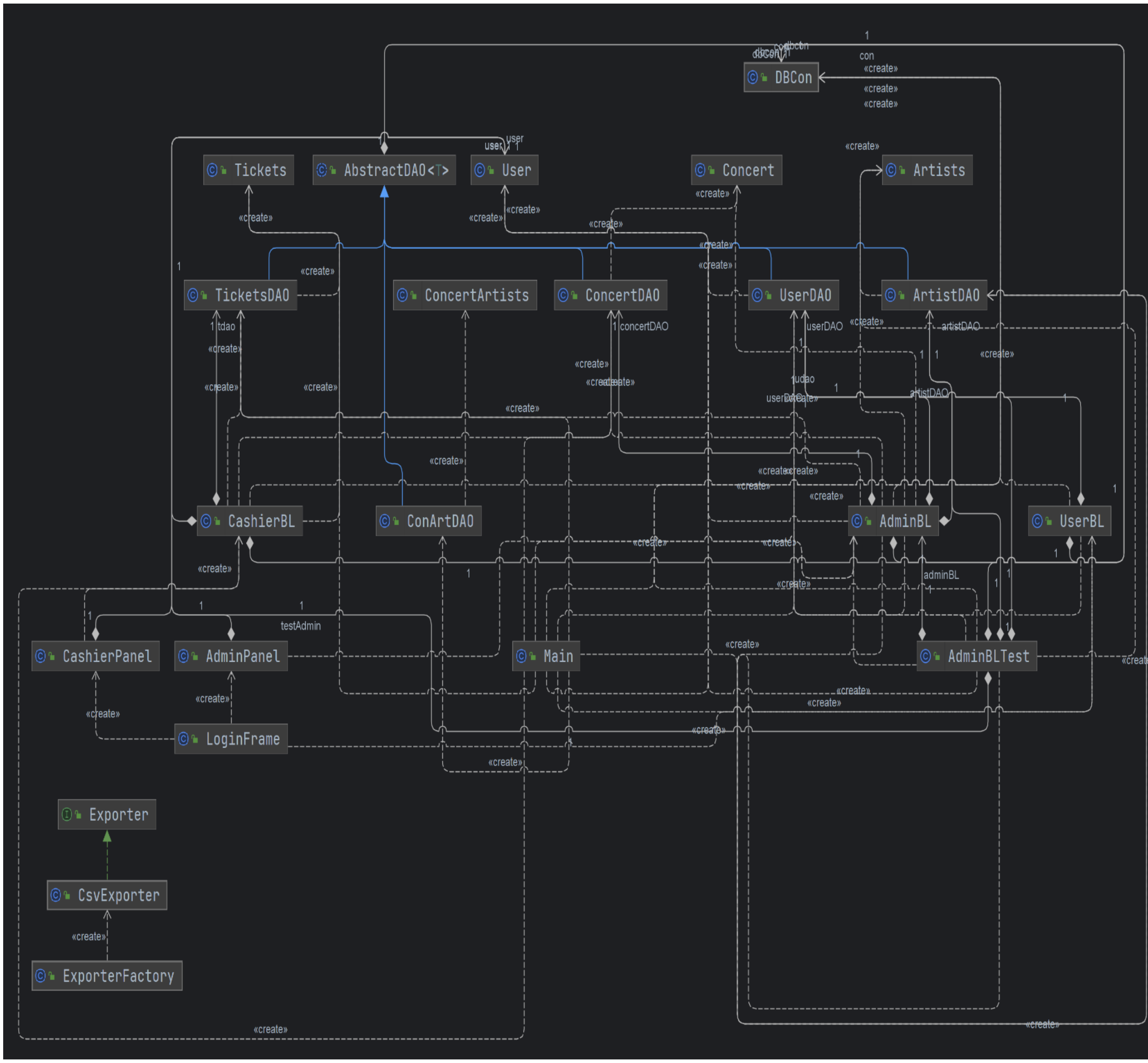
5.1 Design Patterns Description

As a design pattern, the one I've got to use was for the CSV exporting, and I've used the Factory design pattern, which implies that we create object without exposing the creation logic to the client and refer to newly created object using a common interface



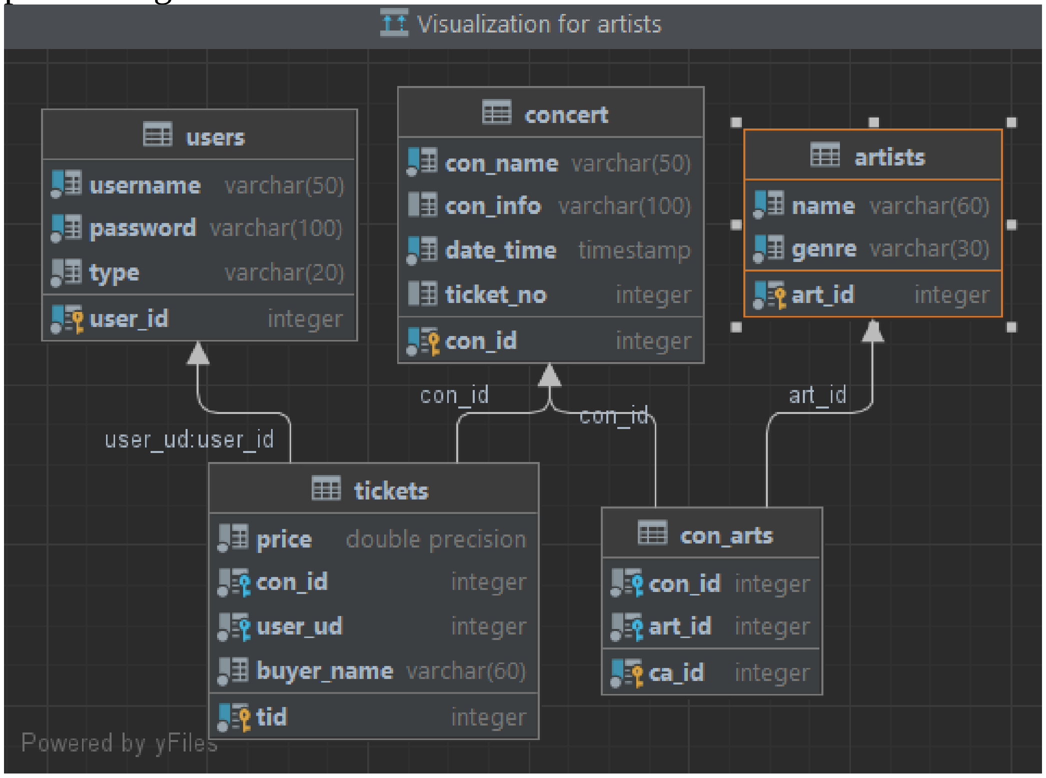
And I used it to create my CSV exporting methods which was then used in my AdminPanel class in order to export the list of tickets for a specific concert to a .csv format file.

5.2 UML Class Diagram



6. Data Model

For this project, I've structured my database using PostgreSQL, and separated my data into different tables. The users table contains the data for the admins and cashiers, concerts represent the performances with their dates and a short description, artists are the singers/bands performing, tickets are created by the cashiers for a specific concert and contain the id of the cashier that created them and the name of the buyer, and con_arts is used to represent a relation between concerts and artists, meaning that a single concert may have multiple artists performing at it.



7. System Testing

For testing, I've created a JUnitTest class which tests the method that updates a specific artist, which only an Admin can perform:

```
public void testUpdateArtist() throws SQLException {
    // Insert a new artist into the database
    Artists artist = new Artists(220, "Test Artist", "Test Genre");
    artistDAO.insert(artist);


    // Update the artist's name and genre
    boolean result = adminBL.updateArtist(artist.getArtistId(), "New Artist
Name", "New Artist Genre");

    // Verify that the artist was updated successfully
    assertTrue(result);

    Artists updatedArtist = artistDAO.findbyId(artist.getArtistId());
    assertEquals("New Artist Name", updatedArtist.getArtistName());
    assertEquals("New Artist Genre", updatedArtist.getArtistGenre());
    // Clean up test data
    artistDAO.delete(artist);
    userDAO.delete(testAdmin);
}
```

here we create a new artist with some random values that are not present in the table already, then we use the updateArtist method in the adminBL class, and user assertTrue(result) to make sure that the update is performed successfully.
Then we retrieve the updated artist and user assertEquals in order to check if the modification was performed successfully. And finally we delete the created artist and admin for the test, in order to avoid unwanted errors in our app.

8. Bibliography

1. Layered Architecture - Software Architecture Patterns [Book] (oreilly.com)

https://docs.oracle.com/middleware/1212/jdev/OJDUG/java_swing.htm

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

https://www.geeksforgeeks.org/data-access-layer/

https://www.postgresqltutorial.com/postgresql-jdbc/connecting-to-postgresql-database/