Untold ticketing application

Student: Cornea Mihai

**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 3

3. System Architectural Design 4

4. UML Sequence Diagrams 6

5. Class Design 6

6. Data Model 8

7. System Testing 8

8. Bibliography 9

1. Requirements Analysis

# Assignment Specification

This application is a Java desktop app used for selling tickets to the Untold festival. It allows a user to log into one of 3 account types: a buyer (who at the moment cannot do anything), a cashier who can sell(create) tickets to a concert and see a full list of thickets to all the concerts, and an admin who has full control over the concerts and users table in the database. The app is connected to a SQLite database which has tables for bands, concerts, tickets, users as well as a ‘concertbands’ table which connects the bands and concerts tables.

# Functional Requirements

* Admin can perform:

CRUD operations on cashier data(modify the users table);

CRUD operations on concerts;

Export all tickets sold for a show(not implemented as of this moment).

* Cashier can create tickets and see all the sold tickets
* Use of a relational database
* Layered architectural pattern

# Non-functional Requirements

* Fast
* Easy to use
* Password encryption for users

2. Use-Case Model

Diagram

Description automatically generated

Use case: Admin authentication and operations

Level: user-goal level

Primary actor: Administrator

Main success scenario: Enter correct authentication data (username and password) for the associated administrator account

Extensions: Login error message, SQL error message

Diagram

Description automatically generated

Use case: Cashier authentication and operations

Level: user-goal level

Primary actor: Cashier

Main success scenario: Enter correct username and password associated with the cashier account

Extensions: Login error message, SQL error message

3. System Architectural Design

**3.1 Architectural Pattern Description**

The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern. This pattern is the de facto standard for most Java EE applications and therefore is widely known by most architects, designers, and developers. The layered architecture pattern closely matches the traditional IT communication and organizational structures found in most companies, making it a natural choice for most business application development efforts.

Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database ([Figure 1-1](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html#sapr_0101_img)). In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (e.g., SQL or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn’t need to know or worry about how to get customer data; it only needs to display that information on a screen in particular format. Similarly, the business layer doesn’t need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.

**3.2 Diagrams**

4. UML Sequence Diagrams

*Diagram

Description automatically generated*

5. Class Design

**5.1 Design Patterns Description**

*[Describe briefly the used design patterns.]*

**5.2 UML Class Diagram**

Graphical user interface, application

Description automatically generated

6. Data Model

A screenshot of a video game

Description automatically generated

7. System Testing

All the testing has been done progressively when creating each class. The Main class has a demo which showcases some of the functionalities of the application as there is no user interface yet.

public static void main(String[] args) throws SQLException {  
 *conn* = new Connect();  
 UserDAO userDAO = new UserDAO(*conn*);  
 //userDAO.insert(new User(3, "cashier1", "1234", 2));  
 System.*out*.println(userDAO.findById(1).getPassword());  
  
 Scanner scan= new Scanner(System.*in*);  
 Cashier cashierBL = new Cashier(*conn*);  
 System.*out*.println("Enter username: ");  
 String text= scan.nextLine();  
  
 while(userDAO.findUser(text) == -1){  
 System.*out*.println("User not found");  
 System.*out*.println("Enter username: ");  
 text= scan.nextLine();  
 }  
  
 User user = userDAO.findById(userDAO.findUser(text));  
 System.*out*.println("Enter password: ");  
 String password = scan.nextLine();  
  
 while (password.equals(user.getPassword()) == false)  
 {  
 System.*out*.println("Wrong password");  
 System.*out*.println("Enter password: ");  
 password = scan.nextLine();  
 }  
 System.*out*.println("Welcome " + user.getUserName());  
  
 if(user.getType() == 2){  
 Cashier cashier = new Cashier(*conn*);  
 cashier.sellTicket(1,1,1,30, "Random user");  
 }  
 else if(user.getType() == 3){  
 Admin admin = new Admin(*conn*);  
 admin.addConcert(4, "Concert4", "Description4", new Timestamp(System.*currentTimeMillis*()), 100);  
 System.*out*.println(admin.seeAllConcerts());  
 admin.deleteConcert(4, "Concert4", "Description4", new Timestamp(System.*currentTimeMillis*()), 100);  
 }  
}

The results of these commands can be seen printed in the console, as well as in the modified tables in the database.

8. Bibliography

<https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html>

<https://www.w3schools.com/sql/>