Object-Oriented Programming

Workshop No. 2 - Object-Oriented Design

Technical implementation plan

Diego Alejandro Yañes Zabala (20251020103)

Brayan Sebastian Diaz Ramirez (20251020150)

Oscar Javier Camargo Trujillo (20251020143)

Universidad Distrital Francisco José de Caldas

Faculty of Engineering

Systems Engineering Program

Teacher: Ing. Carlos Andrés Sierra, M.Sc

Bogotá D.C., Colombia

2025

1. Changes and Decisions from Workshop One

In this phase, new functional and non-functional requirements were added. User authentication with access based on roles (administrator and client) was implemented. For non-functional requirements, it was established that data must be securely stored.

Two new classes were created to specify the roles of *DistriCine* users:

- **CustomerUser:** Represents regular users who can make reservations and payments.
- **AdminUser:** Represents administrators who manage films, schedules, and venues.

This design decision was based on the Single Responsibility Principle, ensuring that each subclass has specific and independent responsibilities while inheriting correctly from the parent class.

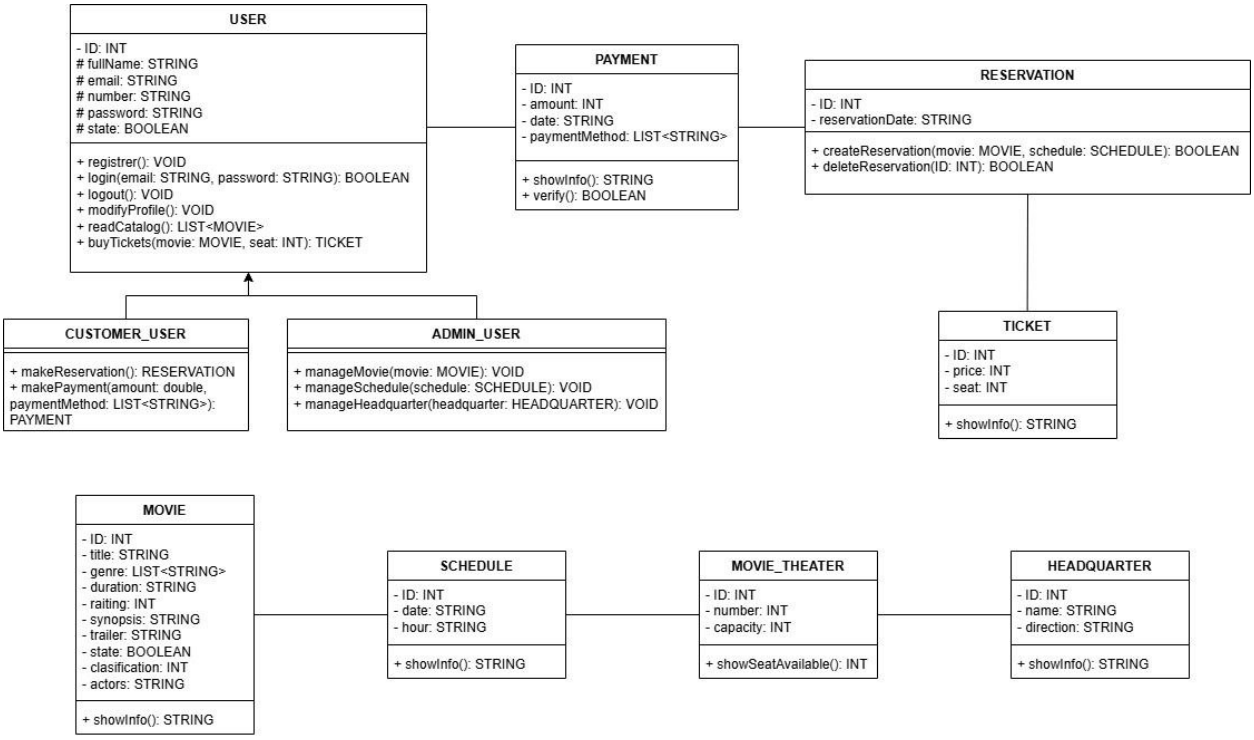
Regarding user stories, the administrator is responsible for managing the movie catalog and the locations where they will be shown. These changes promote modularity and scalability, organizing the project design using OOP principles and ensuring future development potential.

2. Technical Design: UML Diagrams

The UML diagrams were designed and refined to better represent the system's structure and relationships. In the class diagram, the `CustomerUser` and `AdminUser` classes inherit from `User`.

Interactions exist between `User` and `Reservation`, as well as between `Movie`, `Schedule`, `Theater`, and `Headquarters`. Attributes and behaviors are inherited from the superclass `User`, while subclasses extend or adapt methods according to their roles.

For the administrator, three additional system actions are included, while the customer user adds the methods `makeReservation()` and `makePayment()`, tailored to their specific responsibilities.



3. Implementation of OOP Principles in DistriCine

The *DistriCine* project is based on Object-Oriented Programming (OOP) principles: encapsulation, inheritance, and polymorphism.

Encapsulation

Encapsulation is applied to all classes using private access modifiers for attributes and public `get` and `set` methods for controlled access. For example, in the `Movie` class, private attributes such as `title`, `genre`, `duration`, and `rating` can only be modified through public methods. This ensures data integrity and security.

Inheritance

Inheritance is implemented through a class hierarchy defined by the `User` superclass, which groups common attributes and methods such as `name`, `email`, `password`, `login()`, and `logout()`. The derived subclasses include:

- `CustomerUser`: Handles reservations, queries, and payments.
- `AdminUser`: Manages movies, schedules, venues, and users.

This structure promotes code reuse, a clear hierarchical organization, and easy system expansion when introducing new user types.

Polymorphism

Polymorphism is applied primarily through method overriding (`@Override`) in derived classes. For example, the `showMenu()` method defined in `User` is overridden in `CustomerUser` and `AdminUser` to display role-specific menus. Method overloading is also applied, as in `processPayment()`, allowing variations based on parameters.

Project Structure

```
DistriCine
          User.java              # Base class or superclass
          CustomerUser.java      # Subclass representing the
   customer
          AdminUser.java         # Subclass representing the
   administrator
          Movie.java             # Movie information
          Reservation.java       # Reservation management
          Payment.java           # Payment processing
          Schedule.java          # Screening schedules
          Headquarters.java      # Venue information
```

This modular structure facilitates maintenance, supports scalability, and ensures coherence with the UML design.

4. Work in Progress Code

At this stage of the project, the main base and derived classes have been defined according to the previously designed UML diagram.

Parent Class: `User`

```
public abstract class User {
```

```
  private String name;
  private String email;
  private String password;

  public User(String name, String email, String password) {
      this.name = name;
      this.email = email;
      this.password = password;
  }

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public String getEmail() { return email; }
  public void setEmail(String email) { this.email = email; }

  public String getPassword() { return password; }
  public void setPassword(String password) { this.password =
     password; }

  public abstract void showMenu();
}
```

Subclass: `CustomerUser`

```
public class CustomerUser extends User {
  public CustomerUser(String name, String email, String password) {
      super(name, email, password);
  }

  @Override
  public void showMenu() {
      System.out.println("Customer Menu:");
      System.out.println("1. View schedule");
      System.out.println("2. Make a reservation");
      System.out.println("3. Check history");
  }

  public void makeReservation() {
      System.out.println("Reservation in process...");
  }
}
```

Subclass: `AdminUser`

```
public class AdminUser extends User {
  public AdminUser(String name, String email, String password) {
      super(name, email, password);
  }

  @Override
  public void showMenu() {
      System.out.println("Administrator Menu:");
      System.out.println("1. Add Movie");
```

```
        System.out.println("2. Edit Schedule");
        System.out.println("3. Manage Venues");
    }

    public void addMovie() {
      System.out.println("New movie added to the catalog.");
    }
}
```

Complementary Classes

```
public class Movie {
    private String title;
    private String genre;
    private double duration;

    public Movie(String title, String genre, double duration) {
        this.title = title;
        this.genre = genre;
        this.duration = duration;
    }

    public void showInfo() {
        System.out.println("Movie: " + title + " | Genre: " + genre +
                           " | Duration: " + duration + " min");
    }
}

public class Reservation {
    private Movie movie;
```

```
  private String date;
  private String time;

  public Reservation(Movie movie, String date, String time) {
   this.movie = movie;
   this.date = date;
   this.time = time;
  }

  public void confirm() {
      System.out.println("Reservation confirmed for " +
                         movie.getTitle() + " on " + date + " at " +
                             time);
  }
}
```

These classes represent independent domain entities that support user interactions and apply encapsulation through private attributes and public methods.

Brief Reflection

In this second workshop, we experienced the challenge of moving from the design phase to real coding. At the beginning, it was not easy to connect all the classes and keep the relationships consistent, but as we applied encapsulation and inheritance, the logic became much clearer. We also discovered how polymorphism can simplify different behaviors in our system and make the code more flexible. Overall, it was very interesting to see how the theory we learned in class can actually be applied to a real project. Using tools like the UML diagram and having an initial design helped us understand the importance of planning — it made the implementation faster, more organized, and much easier to manage.