Progetto gestionale: CarSharing

Martina Roscica
0124001503

Richiesta

Customer

- Prenotare un’auto
- Ritornare un’auto
- Pagare per il servizio di cui si usufruisce
- EXTRA: visualizzare lo storico delle prenotazioni e dei pagamenti

---

Admin

- Inserire nuova auto
- Inserire nuovo parcheggio
- Visualizzare i ritardi nelle riconsegne

Strumenti

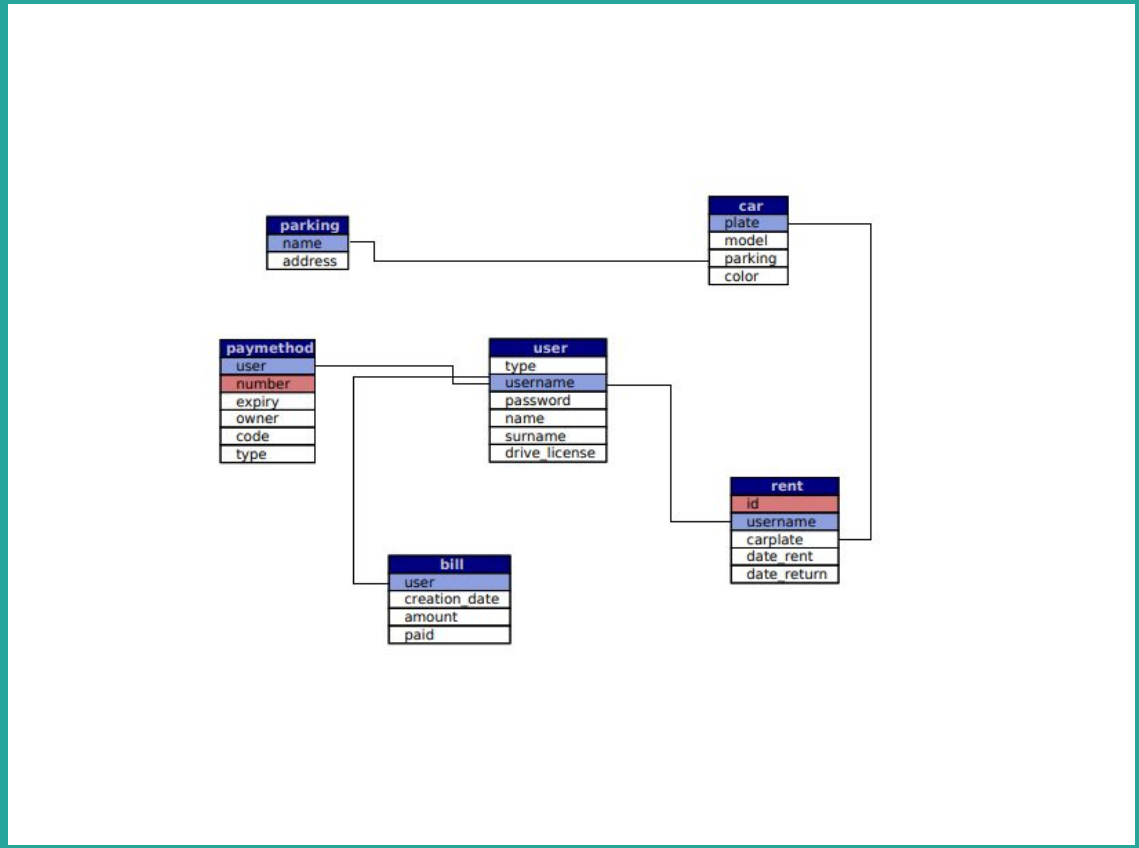
Eclipse - XAMPP MySQL - WindowBuilder

Collegamento al DB

Mediante file JAR, XAMPP e MySQL.
Le operazioni di apertura e chiusura connessione vengono realizzate nella classe DBConnection.java presente nel package Connection come segue:

```java
import java.sql.*;

public class DBConnection {

    public DBConnection() {

    }

    public static Connection openConnection()  {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Connection c=null;
        try {
            c = DriverManager.getConnection("jdbc:mysql://localhost:3306/db_rentit2?user=root&password=");
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return c;
    }
```

UML - DataBase



Autenticazione e registrazione utente

Packages: Entities, Log
Classes: Customer, Admin, Login

La finestra iniziale mostra all'utente la possibilità di iscriversi al servizio o di effettuare l'accesso. In ambedue i casi è coinvolta la classe login che, con l'ausilio di due metodi log() e register(), concedono l'accesso all'utente se ha compilato correttamente i campi di LoginForm e ne inizializzano lo stato di prenotazione con initializeCustomerCar().

Per chi effettua registrazione, è necessario inserire tutte le info richieste e verrà creato l'oggetto Customer in questione e gli sarà concesso direttamente l'accesso al suo menu utente personale.

WindowBuilder

Con tale strumento è stato possibile creare le schermate in cui mostrare le richieste del progetto. Sono state organizzate le seguenti finestre suddividendole tra area utenti e area amministratore.

Area Amministratore

- Menu Principale(admin)
- Sezione di visualizzazione ritardi di riconsegne.
- Form in cui aggiungere auto
- Form per aggiungere parcheggi
- Area dedicata alla redistribuzione delle auto nei parcheggi con la possibilità di visualizzare in tempo reale auto, parcheggi e le eventuali modifiche.

Area Cliente

- Menu Principale(user)
- Sezione prenotazione auto con elenco aggiornabile delle disponibilità.
- Sezione per ritornare l'auto con riepilogo prenotazione correlata.
- Sezione pagamento da effettuare in due diverse modalità.
- Storico prenotazioni e pagamenti relativi.

Design Patterns

Singleton - gestione login/registrazione utente.
State - garantire corretta alternanza di stato dell'auto.
Strategy - implementazione di differenti metodi di pagamento.

Singleton

Classe: login
Utilizzo: LoginForm

Il Singleton è utilizzato per assicurare l'istanza unica di una classe, in questo caso login, e di definire e garantire un unico punto di accesso alla stessa.

```java
//Pattern in uso SINGLETON
public class login {
private String username;
private String password;
private static login instance;
/*
 * instance è static
 */

public login() {
    //this constructor is empty.
}
public static login getInstance() {
    if(instance==null) {
        instance=new login();
    }
    return instance;
}
```

State

Interfaccia: CarState
Classi concrete: UnavailableCarState, AvailableCarState
Context: Car

```
public interface CarState {
public void switchState(Car car);
}
```

Creata un'interfaccia CarState che non definisce il metodo switchState(Car car), le implementazioni di quest'ultima, ossia AvailableCarState e UnavailableCarState, definiscono tale metodo assumendo il comportamento di un interruttore, ossia commutando lo stato dell'auto passata come parametro nello stato opposto.

```
/**
 * Un'auto in questo stato è prenotata, dunque
 * non è disponibile per nuove prenotazioni finchè
 * non verrà restituita e cambiato il suo stato.
 */
public class UnavailableCarState implements CarState {

    static UnavailableCarState instance = new UnavailableCarState();
    public static UnavailableCarState instance() {
        return instance;
    }

    public void switchState(Car car) {
        car.setState(AvailableCarState.instance);
        }
}
```

Strategy

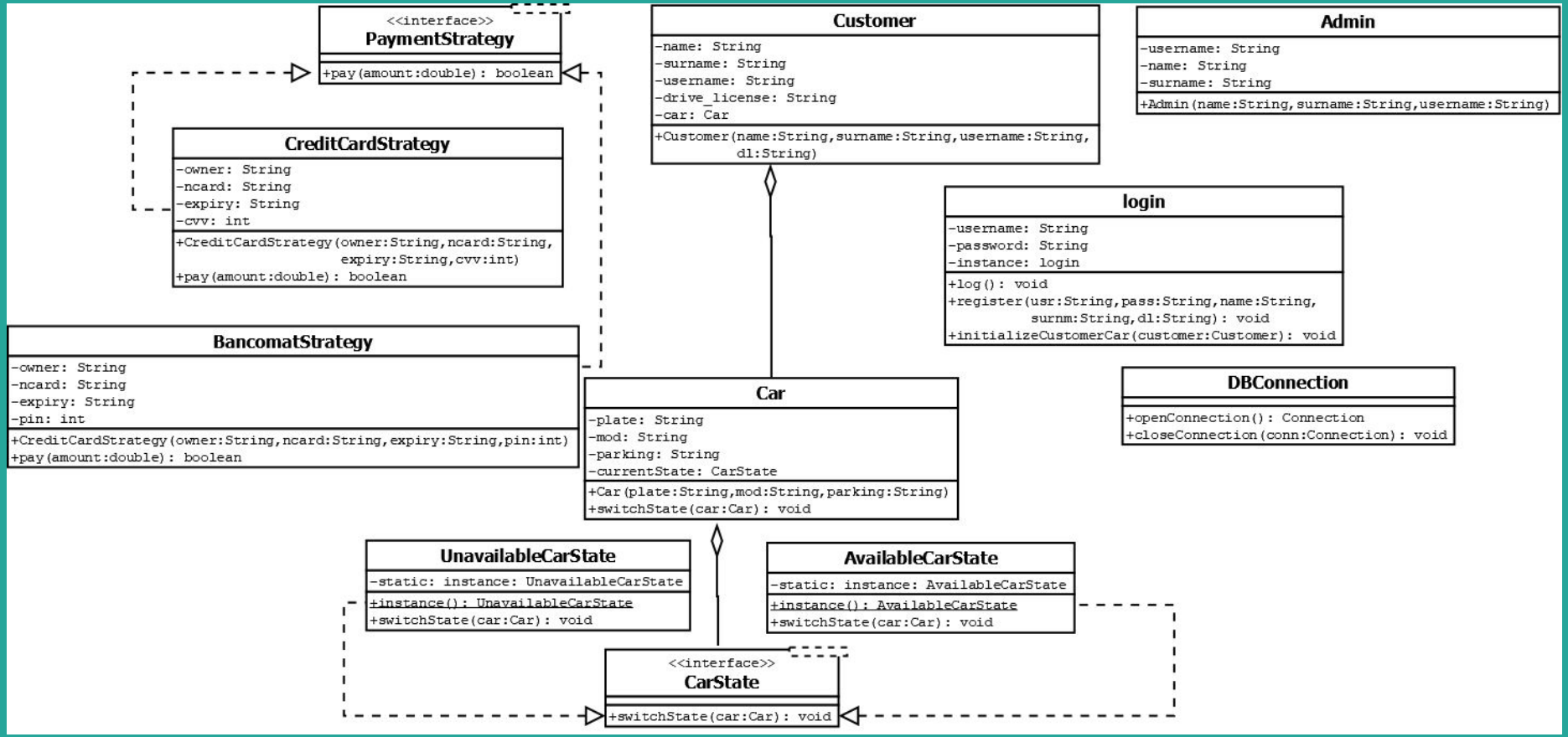
Interfaccia: PaymentStrategy
Classi concrete: BancomatStrategy, CreditCardStrategy
Context: PaymentsFrame

```java
public interface PaymentStrategy {
public void pay(double amount) throws ParseException;
}
```

Lo Strategy viene utilizzato in questo caso per definire due differenti metodi di pagamento BancomatStrategy e CreditCardStrategy, implementazioni dell'interfaccia PaymentStrategy, nelle quali è definito il metodo pay() che controlla le coordinate bancarie inserite se valide o meno.

```java
public boolean pay(double amount)  throws ParseException {
    //Prima di ammettere un pagamento, viene effettuato un controllo sulla data
    Date today = new Date();
    SimpleDateFormat format = new SimpleDateFormat("dd-MM-yyyy");
    Date data = format.parse(this.expiry);
    if(today.compareTo(data)>0) {
        System.out.println("Your credit card expired.");
        return false;
    }
    else {
        System.out.println("Payment submitted. Thank you, "+this.owner+"!");
        return true;
    }
}
```

UML - Classi


<<interface>>
PaymentStrategy
+pay(amount:double): boolean
Customer
-name: String
-surname: String
-username: String
-drive_license: String
-car: Car
+Customer(name:String,surname:String,username:String,
dl:String)
Admin
-username: String
-name: String
-surname: String
+Admin(name:String,surname:String,username:String)
CreditCardStrategy
-owner: String
-ncard: String
-expiry: String
-cvv: int
+CreditCardStrategy(owner:String,ncard:String,
expiry:String,cvv:int)
+pay(amount:double): boolean
login
-username: String
-password: String
-instance: login
+log(): void
+register(usr:String,pass:String,name:String,
surnm:String,dl:String): void
+initializeCustomerCar(customer:Customer): void
BancomatStrategy
-owner: String
-ncard: String
-expiry: String
-pin: int
+CreditCardStrategy(owner:String,ncard:String,expiry:String,pin:int)
+pay(amount:double): boolean
Car
-plate: String
-mod: String
-parking: String
-currentState: CarState
+Car(plate:String,mod:String,parking:String)
+switchState(car:Car): void
DBConnection
+openConnection(): Connection
+closeConnection(conn:Connection): void
UnavailableCarState
-static: instance: UnavailableCarState
+instance(): UnavailableCarState
+switchState(car:Car): void
AvailableCarState
-static: instance: AvailableCarState
+instance(): AvailableCarState
+switchState(car:Car): void
<<interface>>
CarState
+switchState(car:Car): void

Grazie per l’attenzione!