EECS 3311 Project 2 Report

Group 6

Kaibin Liang
Kevin Wang
Chen Yu
Steven Chen

March 25, 2025

Part A: Justification for choosing these six designs patterns.

1. **Users : Factory pattern**

In this system, the Users class has multiple subclasses (such as Student, Faculty Member, Visitor, etc.), and each user type may have different behaviors and properties. Using the Factory Pattern, the user object creation process can be encapsulated in a factory class. In addition, in this system, we may need to constantly add new customers, so using the factory pattern will facilitate subsequent maintenance and expansion.

2. **Booking : Command pattern**

In the Booking module of the parking system, users may perform various operations on reservations, such as editing a reservation, canceling a reservation, extending a reservation, or querying reservation details. These operations are essentially "behavior requests" with clear execution targets (Booking objects) and operation methods. Using the Command Pattern, each operation is encapsulated into a specific command class, which uniformly implements a common Command interface (such as the execute() method), so that the "initiator of the request" (such as the user interface or controller) and the "executor of the operation" (that is, the method of the Booking class) can be decoupled.

3. **Parking :Observer pattern**

The Observer Pattern is used in ParkingFrame because it is necessary to implement a real-time interface update mechanism after the parking space data changes. In this pattern, ParkingModel plays the role of "Subject", responsible for maintaining the status data of the parking lot and parking spaces (such as availability, occupancy status, license plate number), while ParkingFrame, as an "Observer", registers with the model to receive notifications. When the sensor detects a change in the vehicle or the administrator manually updates the status, the system reloads the information from the data source (such as CSV or database), and ParkingModel calls the function to automatically notify all registered views to update their own content.

4. **Manager - singleton pattern**

In the parking management system, the Manager is responsible for user approval, parking lot configuration, and system permission control, all of which are global behaviors at the system level. Therefore, the Singleton Pattern ensures that there is only one Manager instance in the entire system, avoiding permission conflicts and inconsistent states, while providing a unified and controlled access entry for other modules. Through private constructors and global access methods, the Singleton Pattern effectively limits the way Managers are created, which not only improves the security of the system, but also simplifies resource synchronization and global management logic.

5. **Payment - strategy pattern**

In the parking management system, the Payment module involves a variety of different payment methods, each of which corresponds to different payment logic and processing rules. Using the Strategy Pattern, each payment method can be encapsulated into an independent strategy class and the PaymentStrategy interface can be uniformly implemented, so that the system can dynamically switch payment methods according to user type or payment selection at runtime.

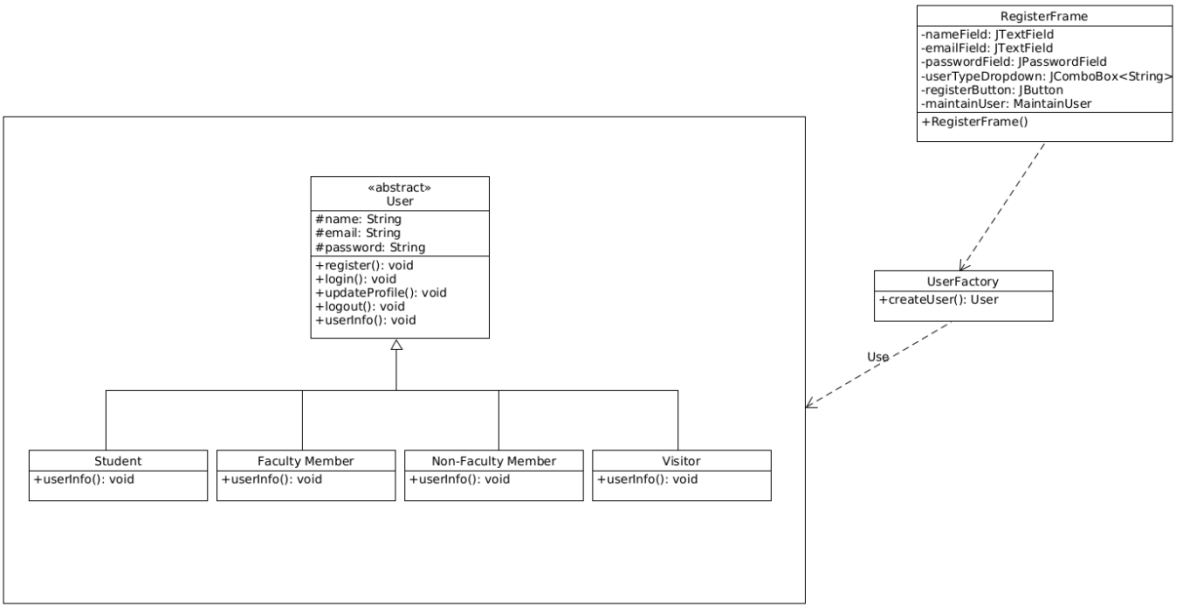
This design pattern facilitates the subsequent addition of new payment methods (such as Apple Pay, campus card, etc.).

6. Admin - Proxy pattern

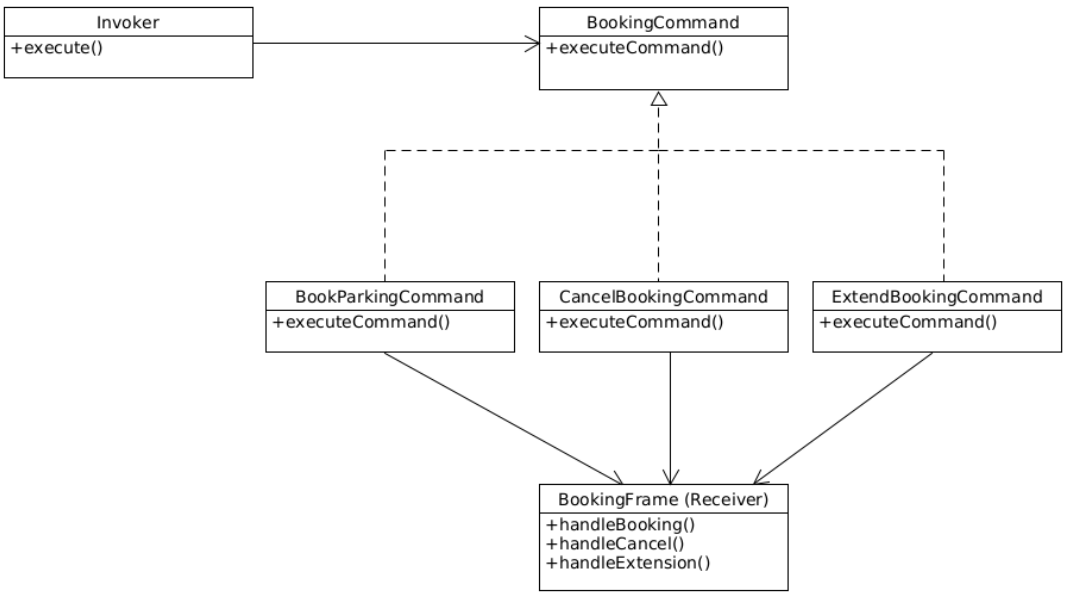
In the parking system, the Admin module is responsible for key functions such as enabling parking lots, approving users, and modifying system status. In order to protect these sensitive operations from being called arbitrarily by ordinary users, by introducing the Proxy Pattern, we can perform permission verification and operation log recording before each call, and even implement extended functions such as delayed loading of real objects.

Part B: Class diagrams

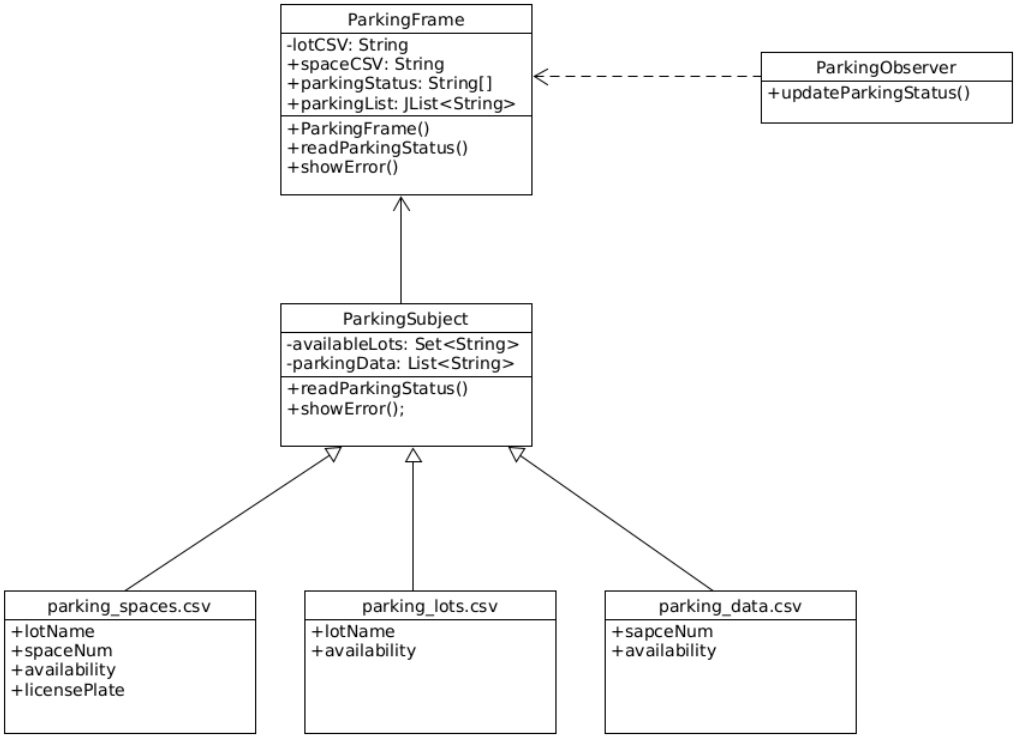
1. Factory:



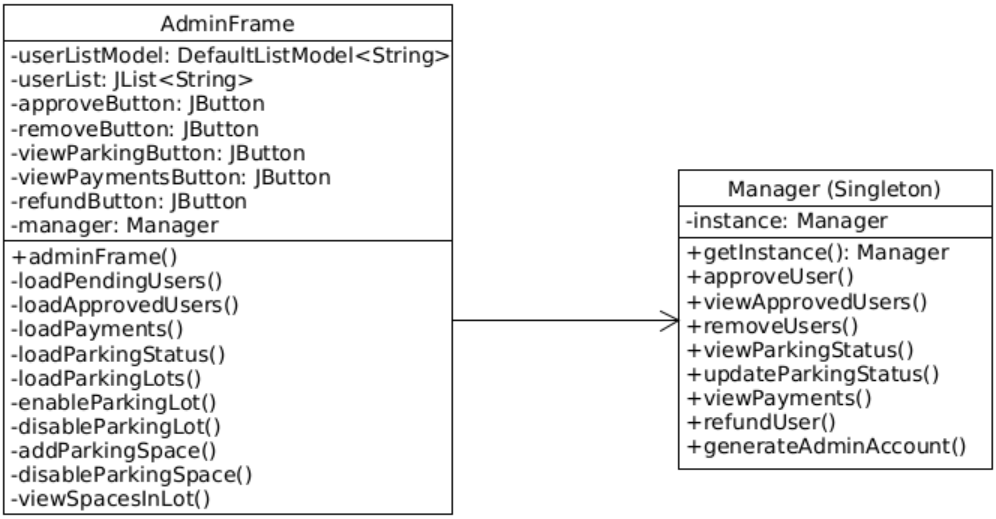
2. Command:



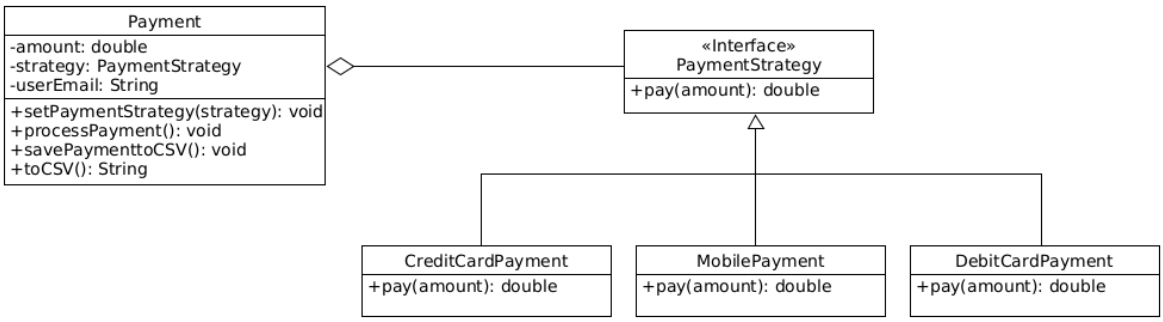
3. Observer:



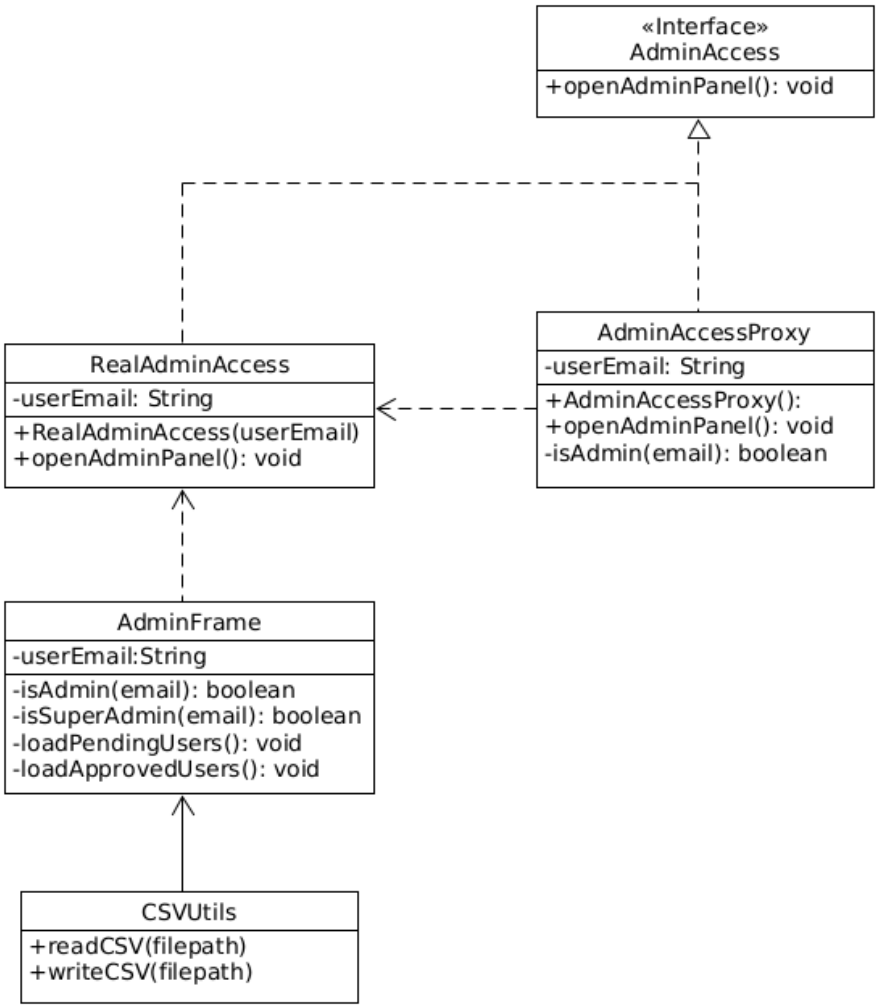
4. Singleton:



5. Strategy:



6. Proxy:



Part C：Discussion about how each requirement can be achieved.

Req1:

RegisterFrame.java specifies that there are only four options: "Student", "Faculty", "Non-Faculty", and "Visitor". The factory pattern is used to create different subclass user objects such as Student, FacultyMember, etc., and write them to users.csv. The registration information is then written to the data/users.csv file through the addUser() and update() methods in

MaintainUser.java, which serves as a cache for users to be approved. After logging in, the administrator can manage all new registration requests in a unified manner. The administrator uses the interface operation in AdminFrame.java to call the Manager.approveUser(email) method to transfer the selected user from users.csv to approved_users.csv. The password and email fields are collected in the registration page and saved as strings. The implementation allows passwords to have a combination of uppercase letters, lowercase letters, numbers, and symbols.

```
panel.add(new JLabel("User Type:"));
userTypeDropdown = new JComboBox<>(new String[]{"Student", "Faculty", "Non-Faculty", "Visitor"});
panel.add(userTypeDropdown);
```

```
public void update(String path) throws Exception {
    CsvWriter csvOutput = new CsvWriter(new FileWriter(path, false), ',');
    csvOutput.write("type");
    csvOutput.write("name");
    csvOutput.write("email");
    csvOutput.write("password");
    csvOutput.endRecord();

    for (User u : users) {
        csvOutput.write(u.getClass().getSimpleName());
        csvOutput.write(u.getName());
        csvOutput.write(u.getEmail());
        csvOutput.write(u.getPassword());
        csvOutput.endRecord();
    }
    csvOutput.close();
}
```

```
// Add new user
public void addUser(User user) {
    users.add(user);
}
```

```
public static User createUser(String type, String name, String email, String password) {
```

Req2:

In the users package, the Superadmin.java class is defined to represent the super administrator. In AdminFrame.java, the isSuperAdmin(email) method is used to determine whether the current logged-in user is a super administrator. In AdminFrame.java, determine whether the user is a superadmin. If so, the admin account can be generated through generateAdminAccount() in manager.java.

```java
package user;

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

    @Override
    public void displayInfo() {
        System.out.println("Superadmin: " + name + " " + email);
    }
}
```

```java
    private boolean isSuperAdmin(String email) {
        List<String[]> users = CSVUtils.readCSV("data/approved_users.csv");
        for (String[] row : users) {
            if (row.length >= 4) {
                String type = row[0];
                String rowEmail = row[2];
                if (email.equalsIgnoreCase(rowEmail) && type.equalsIgnoreCase("SuperAdmin")) {
                    return true;
                }
            }
        }
        return false;
    }


    // Generate new admin account (SuperAdmin only)
    public void generateAdminAccount() {
        String username = "admin" + UUID.randomUUID().toString().substring(0, 5);
        String email = username + "@yorku.ca";
        String password = UUID.randomUUID().toString().substring(0, 8) + "#A";


        CSVUtils.writeCSV(APPROVED_USERS_FILE, new String[]{"Admin", username, email, password});

        JOptionPane.showMessageDialog(null,
                "New admin account generated:\nUsername: " + username +
                        "\nEmail: " + email +
                        "\nPassword: " + password,
                "Success", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Req3:

In PaymentFrame.java, the corresponding parking rate is assigned by the email address of the currently logged in user. The payment strategy pattern PaymentStrategy is used to support multiple payment methods.

```
        try {
            maintainUser.load();
            type = maintainUser.getUserType(userEmail);
            rate = switch (type.toLowerCase()) {
                case "student" -> 5.0;
                case "faculty" -> 8.0;
                case "non-faculty" -> 10.0;
                case "visitor" -> 15.0;
                default -> 10.0;
            };
        } catch (Exception ex) {
            ex.printStackTrace();
        }
```

```
        payButton.addActionListener(e -> {
            Payment payment = new Payment(finalRate, userEmail);

            String selected = (String) paymentDropdown.getSelectedItem();
            if (selected.equals("Credit Card")) {
                payment.setPaymentStrategy(new CreditCardPayment());
            } else if (selected.equals("Mobile Payment")) {
                payment.setPaymentStrategy(new MobilePayment());
            } else {
                payment.setPaymentStrategy(new DebitCardPayment());
            }

            payment.processPayment();
            JOptionPane.showMessageDialog(null, "Payment successful!");
        });
```

Req4

After the user enters the information, the system displays a pop-up window explaining the deposit policy, which requires the user to confirm. The system also records licensePlate, startTime, and endTime. In cancelParkingBooking(), if the current time exceeds the booking start time + 1 hour, it is considered a No-show and no refund is given; otherwise, it is considered an early cancellation and the deposit is refunded; the parking space status is reset to Available, and the parking space is released.

```
        // Show deposit information
        int confirm = JOptionPane.showConfirmDialog(this,
            "A deposit of $" + HOURLY_DEPOSIT + " will be charged.\n" +
            "If you don't arrive within 1 hour of your booking time,\n" +
            "this deposit will not be refunded.",
            "Deposit Information",
            JOptionPane.OK_CANCEL_OPTION,
            JOptionPane.INFORMATION_MESSAGE);
```

```
if (now.isAfter(startTime)) {
    // This is now acting as check-out
    if (now.isAfter(startTime.plusHours(1))) {
        showMessage("Checked out. Deposit of $" + HOURLY_DEPOSIT + " not refunded (no-show).");
    } else {
        showMessage("Checked out. Your $" + HOURLY_DEPOSIT + " deposit has been refunded.");
    }
} else {
    // Standard cancellation before start time
    showMessage("Booking cancelled. Your $" + HOURLY_DEPOSIT + " deposit has been refunded.");
}
```

Req5:

The parking space status is detected by the sensor. If the parking space is occupied, the license plate number is read and the csv file is updated.

```
// 读取停车位状态，新增支持车牌扫描显示
try (BufferedReader br = new BufferedReader(new FileReader(spaceFile))) {
    String line;
    while ((line = br.readLine()) != null) {
        String[] parts = line.split(",");
        if (parts.length >= 3) {
            String lotName = parts[0].trim();
            String spaceNumber = parts[1].trim();
            String status = parts[2].trim();
            String plate = (parts.length >= 4) ? parts[3].trim() : "";

            if (availableLots.contains(lotName)) {
                String entry = "Parking space " + spaceNumber + ": " + status;
                if (status.equalsIgnoreCase("Occupied") && !plate.isEmpty()) {
                    entry += " - Plate: " + plate;
                }
                result.add(entry);
            }
        }
    }
```

Req6:

By constructing four methods: enableParkingLot(), disableParkingLot(), addParkingSpace(), and disableParkingSpace(), the admin can modify the status of parking lots and parking spaces in the CSV file, and when parkinglot is closed, all parking spaces in the lot will also be closed.

```java
// Enable parking lot
private void enableParkingLot() {
    String lotId = JOptionPane.showInputDialog("Enter the lot ID to enable:");
    if (lotId != null && !lotId.trim().isEmpty()) {
        List<String[]> lots = CSVUtils.readCSV("data/parking_lots.csv");
        List<String[]> spaces = CSVUtils.readCSV("data/parking_spaces.csv");
        boolean found = false;

        for (String[] lot : lots) {
            if (lot[0].equalsIgnoreCase(lotId.trim())) {
                lot[1] = "Available";
                found = true;
            }
        }

        for (String[] space : spaces) {
            if (space[0].equalsIgnoreCase(lotId.trim())) {
                space[2] = "Available";
            }
        }

        if (found) {
            CSVUtils.writeCSV("data/parking_lots.csv", lots);
            CSVUtils.writeCSV("data/parking_spaces.csv", spaces);
            JOptionPane.showMessageDialog(null, "Lot enabled: " + lotId);
            loadParkingLots();
        } else {
            JOptionPane.showMessageDialog(null, "Lot not found.");
        }
    }
}


// Add parking space
private void addParkingSpace() {
    String lotId = JOptionPane.showInputDialog("Enter lot ID (e.g., Lot1):");
    String spaceId = JOptionPane.showInputDialog("Enter space ID (e.g., P1):");
    if (lotId != null && spaceId != null &&
            !lotId.trim().isEmpty() && !spaceId.trim().isEmpty()) {
        CSVUtils.writeCSV("data/parking_spaces.csv",
                new String[]{lotId.trim(), spaceId.trim(), "Available"});
        JOptionPane.showMessageDialog(null, "Added space: " + lotId + "-" + spaceId);
    }
}

// Disable parking space
private void disableParkingSpace() {
    String fullId = JOptionPane.showInputDialog("Enter space to disable (format Lot1-P1):");
    if (fullId != null && fullId.contains("-")) {
        String[] parts = fullId.split("-");
        String lotId = parts[0].trim();
        String spaceId = parts[1].trim();
        List<String[]> spaces = CSVUtils.readCSV("data/parking_spaces.csv");
        boolean found = false;

        for (String[] space : spaces) {
            if (space[0].equalsIgnoreCase(lotId) && space[1].equalsIgnoreCase(spaceId)) {
                space[2] = "Unavailable";
                found = true;
            }
        }

        if (found) {
            CSVUtils.writeCSV("data/parking_spaces.csv", spaces);
            JOptionPane.showMessageDialog(null, "Space disabled: " + fullId);
            loadParkingStatus();
        } else {
            JOptionPane.showMessageDialog(null, "Specified space not found.");
        }
    }
}
```

Req7:
The uniqueness of the parking space is achieved by assigning "parking lot number + parking space number" to each parking space, and the status of the parking space can be obtained by reading parking_spaces.csv.

```
Lot1,P1,Occupied,ADD23,null,null
Lot1,P2,Occupied,GKD16,null,null
Lot1,P3,Occupied,TYY79,null,null
Lot1,P4,Available,,,
Lot1,P5,Available,null,null,null
Lot1,P6,Available,null,null,null
Lot1,P7,Available,null,null,null
```

Req8:
In BookingFrame, users must enter license plate information when making a reservation, and update it to parking_spaces.csv. In handleCancel(), read the current time and compare it with the reservation start time. If the current time is earlier than the reservation start time → cancel successfully.

```
// License plate input
add(new JLabel("License Plate Number:"));
licensePlateField = new JTextField();
add(licensePlateField);

private void handleCancel() {
    String licensePlate = licensePlateField.getText().trim();
    if (licensePlate.isEmpty()) {
        showError("Please enter your license plate number");
        return;
    }
```

Req9: In handleExtension(), find the existing reservation based on the license plate number + parking space location; compare whether the time to be changed is valid; if successful, update the information in parking_spaces.csv.

```
    private void handleExtension() {
        String licensePlate = licensePlateField.getText().trim();
        if (licensePlate.isEmpty()) {
            showError("Please enter your license plate number");
            return;
        }

        String spot = (String) parkingSpotDropdown.getSelectedItem();
        String[] booking = findBooking(spot, licensePlate);

        if (booking == null) {
            showError("No active booking found");
            return;
        }

        try {
            LocalTime currentEnd = LocalTime.parse(booking[5], TIME_FORMATTER);
            LocalTime now = LocalTime.now();

            if (now.isAfter(currentEnd)) {
                showError("Cannot extend booking after it has ended");
                return;
            }

            String newEndTime = JOptionPane.showInputDialog(this,
                "Current end time: " + booking[5] + "\nEnter new end time (HH:mm):",
                currentEnd.plusHours(1).format(TIME_FORMATTER));

            if (newEndTime == null || newEndTime.trim().isEmpty()) return;

            LocalTime newEnd = LocalTime.parse(newEndTime.trim(), TIME_FORMATTER);
            if (!newEnd.isAfter(currentEnd)) {
                showError("New end time must be after current end time");
                return;
            }

            if (extendParkingBooking(spot, licensePlate, newEndTime)) {
                showSuccess("Booking extended successfully!\nNew end time: " + newEndTime);
                endTimeField.setText(newEndTime);
            } else {
                showError("Failed to extend booking");
            }
```

Req10:

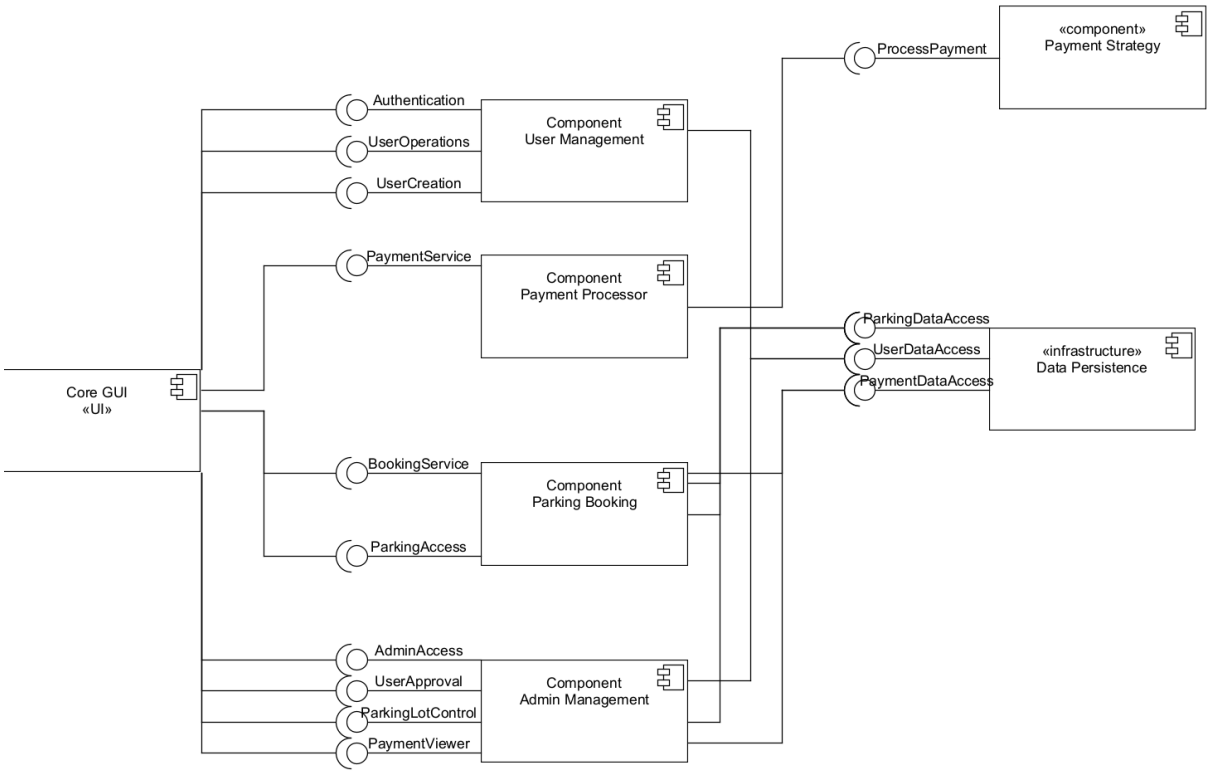
Use the Strategy design pattern to implement scalable multiple payment methods. First design the interface PaymentStrategy.java, then expand different payment methods, such as Credit Card and Mobile Payment. If you need to add payment methods later, just add a new class in the utils folder.

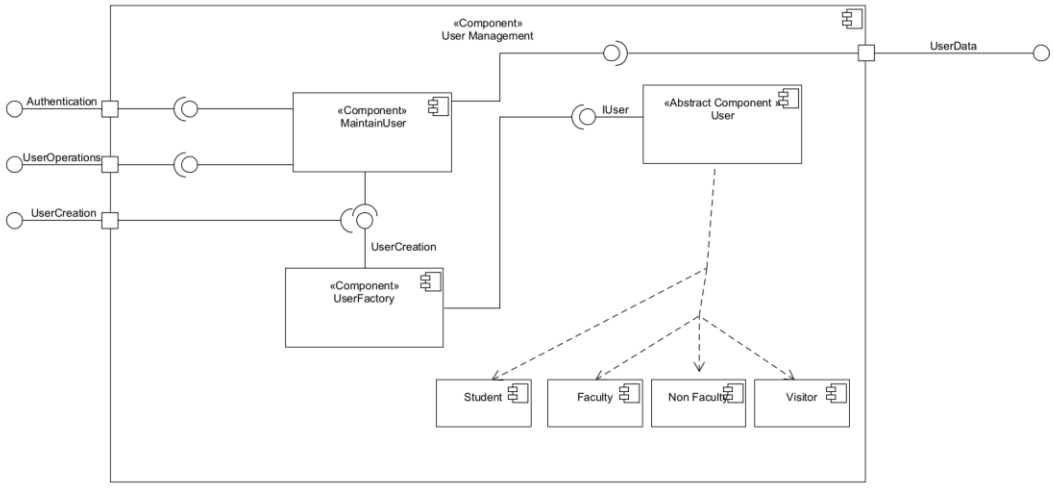
```java
package utils;


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

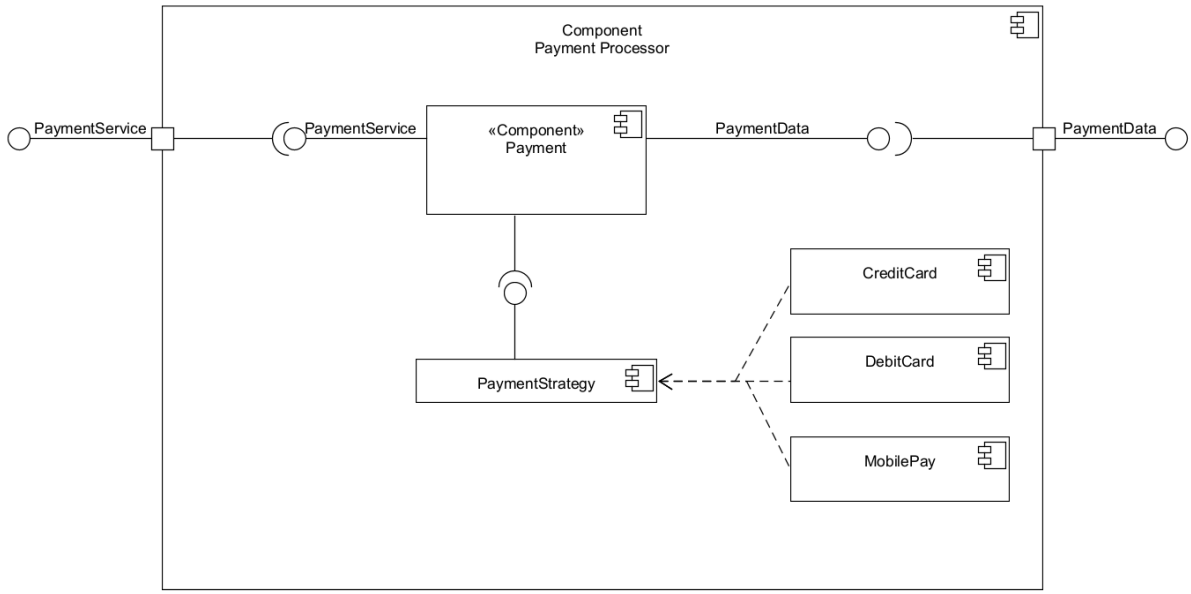
```java
public class CreditCardPayment implements PaymentStrategy {
@Override
public void pay(double amount) {
   System.out.println("Pay by credit card $" + amount);
}
}
```

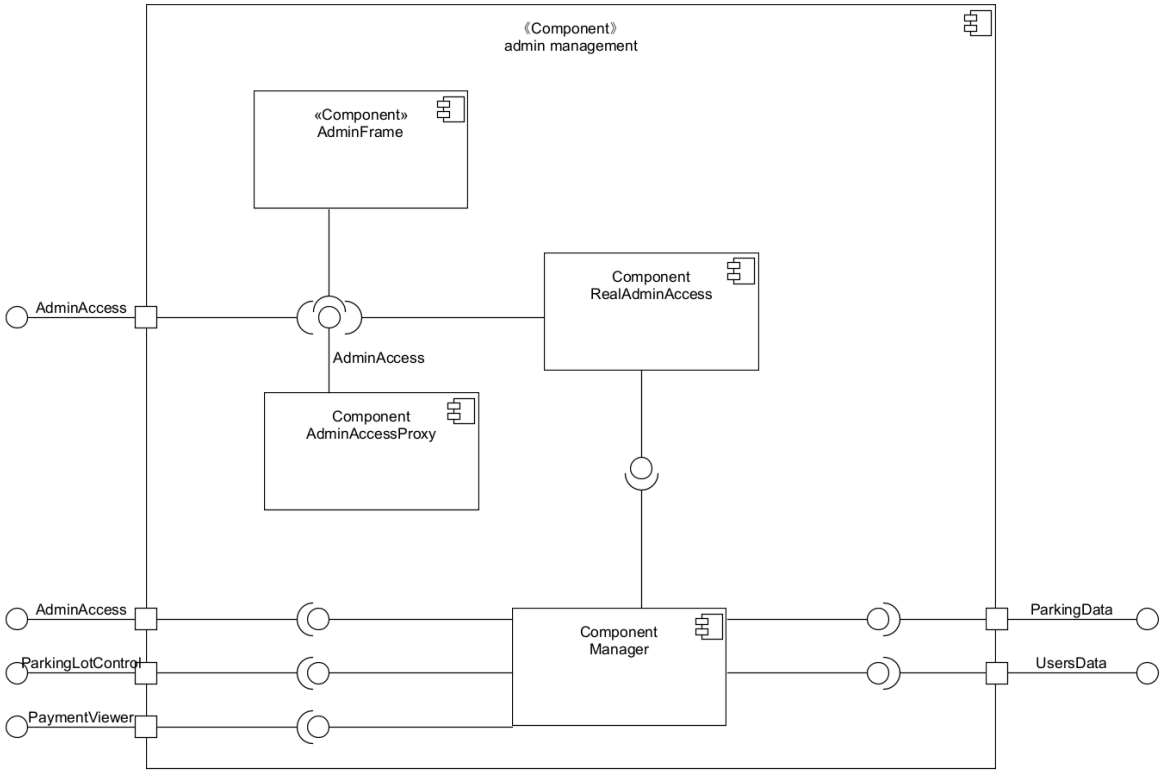
```java
        // Select payment method
        add(new JLabel("Payment Method:"));
        String[] paymentMethods = {"Credit Card", "Mobile Payment", "Debit Card"};
        JComboBox<String> paymentDropdown = new JComboBox<>(paymentMethods);
        add(paymentDropdown);

        // Pay button
        JButton payButton = new JButton("Pay");
        add(payButton);

        double finalRate = rate;
        payButton.addActionListener(e -> {
            Payment payment = new Payment(finalRate, userEmail);

            String selected = (String) paymentDropdown.getSelectedItem();
            if (selected.equals("Credit Card")) {
                payment.setPaymentStrategy(new CreditCardPayment());
            } else if (selected.equals("Mobile Payment")) {
                payment.setPaymentStrategy(new MobilePayment());
            } else {
                payment.setPaymentStrategy(new DebitCardPayment());
            }

            payment.processPayment();
            JOptionPane.showMessageDialog(null, "Payment successful!");
        });

        setVisible(true);
    }
```

Part D: Component diagram









Part E: Justification for component decomposition and interactions

The YorkU Parking System has been broken down into its component parts, with data

persistence, display, and business logic all kept separate. This design ensures high cohesion and low coupling while enabling scalability, following the Single Responsibility Principle

Main components breakdown

Component	Responsibility	Design Rationale
Core UI	Handles user authentication UI and input validation.	Decouples GUI from authentication logic for reusability.
Booking & Parking	Manage parking reservation, extension and cancelation for booking	Isolates booking workflow with integrated time validation.
Admin Management	Provides admin controls for user/lot management	Uses Proxy Pattern to restrict unauthorized access.
User Management	Loads/saves user data and verifies credentials.	Centralizes user operations to avoid code duplication.
Payment	Processes payments via Credit/Debit/Mobile methods.	Strategy Pattern allows flexible payment method extensions.
Data Persistence CSVUtils	Handles all CSV file I/O operations.	Abstracts file operations for maintainability.
Payment Strategy	Easier to expand later.	Using strategy pattern makes subsequent expansion easier.

Interaction Justification

GUI Separation: Frames (Booking, Login, Admin…) delegate logic to backend classes (Manager, MaintainUser), ensuring UI components only handle presentation.

Centralized Control: The Manager singleton functions as a cover for administrative activities, preventing duplicating state management.

Extensible Payments: New payment methods can be added without changing the fundamental logic because of the strategy implementations in payment

Data Access Isolation: Other components can read and write data without the need for CSV-specific code because of CSVUtils, that packs file operations.

Authentication Flow: LoginFrame -> MaintainUser -> users.csv. Make sure credential authentication is safe and diverse.

The design of the system's components follows SOLID principles, and design patterns and method delegation are used to handle interactions. The design is adaptable and allows expansion of individual components.