XI'AN JIAOTONG-LIVERPOOL UNIVERSITY
西 交 利 物 浦 大 学
COURSEWORK SUBMISSION COVER Page

Group Number	Group 39					
Students' ID Number	2360908	2362352	2253158	2364452	2360872	2359977
Module Code	CPT203					
Assignment Title	Coursework 2					
Submission Deadline	12 December 2025, 22:00					

By uploading this coursework submission with this cover page, we certify the following:

- ❖ We have read and understood the definitions of collusion, copying, plagiarism, and dishonest use of data as outlined in the Academic Integrity Policy of Xi'an Jiaotong- Liverpool University.
- ❖ This work is **entirely** our own, original work produced specifically for this assignment. It does not misrepresent the work of another group or institution as our own.
- ❖ This work is not the product of unauthorized collaboration between ourselves and others.
- ❖ This work has not been shared wholly or in part outside of the group. Each member has performed the duty to protect the submission until the feedback is released.
- ❖ It is a submission that has not been previously published, or submitted to any modules.
- ❖ Students who would like to submit the same or similar work from previous years to the current module or other modules must receive written permission from all instructors involved in advance of the assignment due date.
- ❖ **All** group members are **equally** and **collectively** responsible for the **entire** submission. Violations of academic integrity including failure to monitor group member contribution originality constitutes negligence.
- ❖ Unreported suspicious academic misconduct by one group member will be attributed to ALL members in terms of penalties. **ALL members share the responsibilities**.
- ❖ Use of generative **AI is strictly prohibited** for all assessments involved in this module.

We understand collusion, plagiarism, dishonest use of data, and submission of procured work are serious academic misconducts. **All** group members are held **jointly accountable** for the integrity of the **entire** submission. By uploading or submitting this cover page, we acknowledge that we are jointly subject to penalties and disciplinary actions if we are found to have committed such acts.

~~~~~~

We acknowledge that the university late submission policy will be applied if applicable.
~~~~~~

~~~~~~

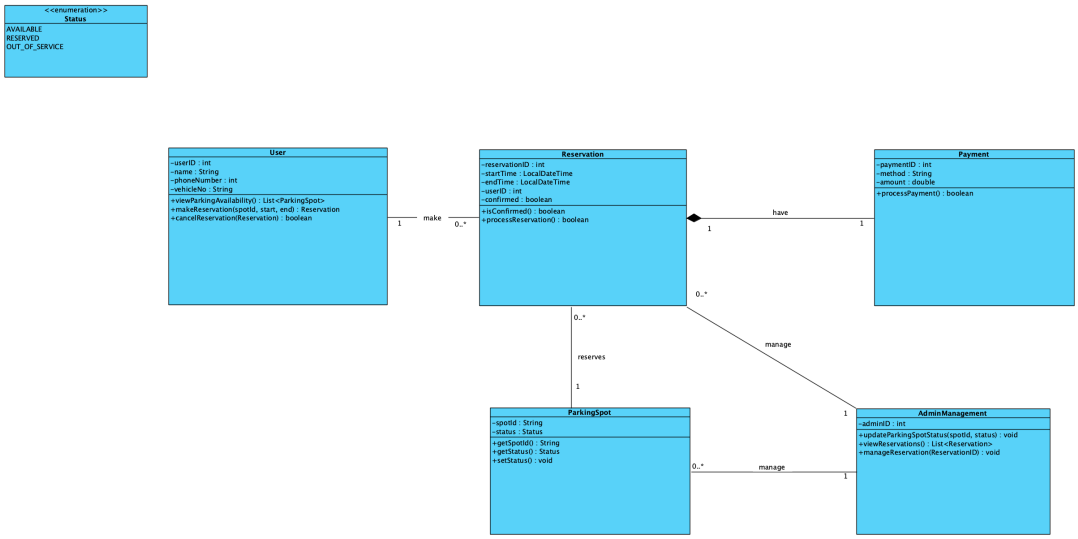
Please list the ID number of any group member NOT contributing to the submission:

Please indicate whether based on your individual contributions you meet the learning outcomes (LOs) covered by this coursework and confirm your submission meets with all above requirements by signing your name, handwritten in ink, in English (Foreign students) or PINYIN (local students):

Student ID	**✓(tick) box to confirm you contributed to all cw tasks**	**✓(tick) box to confirm you meet all LOs covered by this cw**	**Signed**
2360908	✓	✓	Qi Cao
2362352	✓	✓	Zixi Chen
2253158	✓	✓	Jingchen Ding
2359977	✓	✓	Haoyan Xu
2364452	✓	✓	Jingxuan Weng.
2360872	✓	✓	Shengjie Xu

Date: ……………2025.12.12………………………
~~~~~~

Q1. Class Diagram (15 marks).



Q2. Software Design Concepts (15 marks).

2.1 Abstraction

Our parking system demonstrates abstraction at multiple levels. Take the Payment class as an example – At a high level, the class name itself indicates the basic concept; At a medium level, the attributes in the class reveal that in our case, these are payments with individual IDs, statuses and amounts (data abstraction); At a low level, a detailed solution processPayment() method is implemented, providing profound knowledge of the purpose and functionality of this class. Meanwhile, the method processPayment() also demonstrates procedural abstraction by embodying a sequence of detailed steps of how each payment is actually processed. With this approach, users just need to know whether their payment has been made or not, instead of worrying about how each payment is processed. This abstraction enhances maintainability by allowing internal changes without affecting external code, and improves scalability by enabling us to easily add new payment types or processing methods within the class.

2.2 Modularity

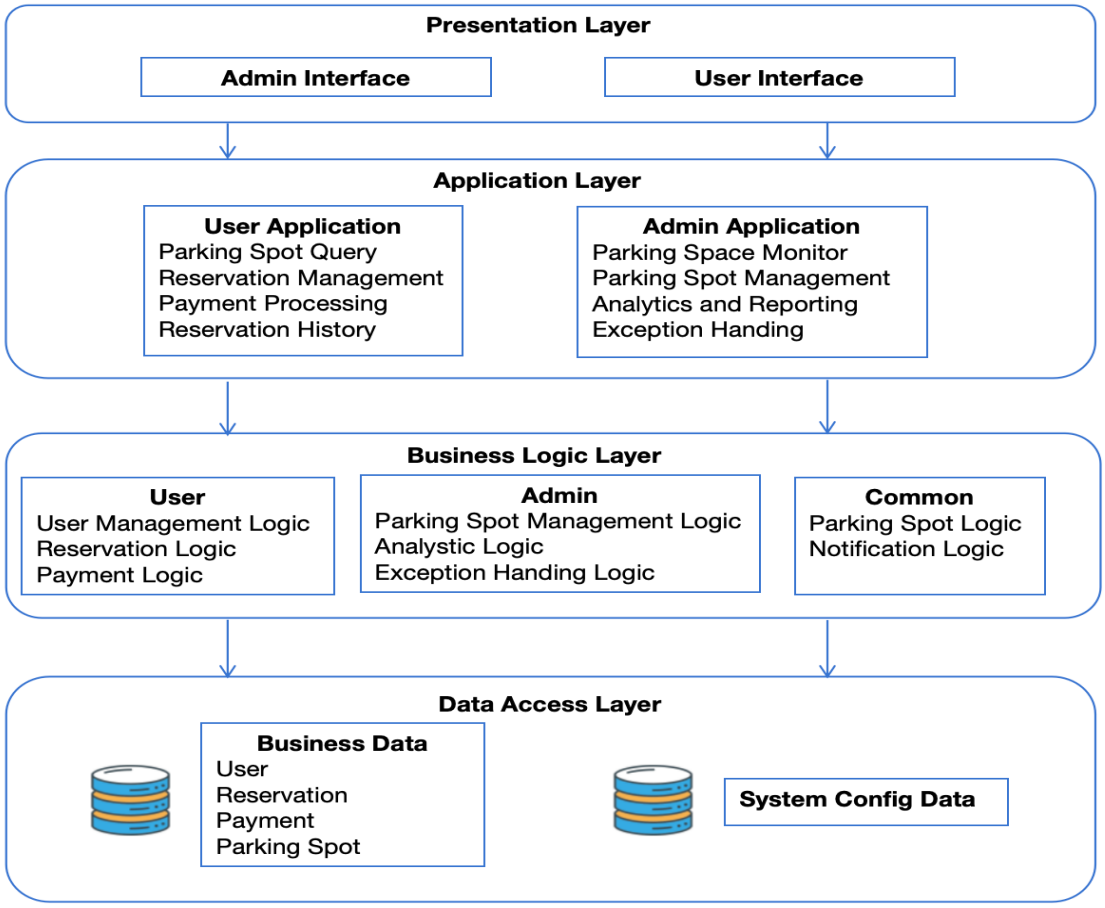
The system is divided into five main modules that can work independently: the User management module, the reservation handling module, the parking spot tracking module, the payment processing module, and the admin control module. By setting up boundaries and dividing responsibilities to the separated modules, we can save time in the development process and ensure easy maintainability when only parts of the system need to be modified. For instance, if we need to change how payments work or fix payment crashes in actual deployment, we only need to update and modify the Payment class without touching other classes' logic. Additionally, this modular structure supports scalability by allowing us to expand individual modules independently – for example, we can scale up the reservation module to handle more concurrent bookings without affecting other parts of the system.

2.3 Coupling

We've designed for loose coupling in our system. The classes interact through well-defined interfaces rather than depending heavily on each other. For instance, the Reservation class connects to Payment through a simple "have" relationship. Reservation doesn't need to know how Payment processes transactions - it just needs to know a payment exists. This separation of concerns supports flexibility and maintainability when a different payment system is required – we can simply replace it with the current one with minimize changes without affecting the Reservation class logic. Moreover, it enhances scalability by allowing us to add new components or integrate other services with minimal impact on existing code.

Q3. Architecture Selection (15 marks)

3.1 Structural Diagram of Layered-Architecture



3.2 Reasons for Architecture Selection

Based on the business requirements and technical characteristics of the Smart Parking System (SPS), the Layered Architecture is the most suitable development framework.

3.2.1 Layered-Architecture Alignment & Low Development and Maintenance Complexity

The core business process of SPS follows the data sequence of user interaction-business logic-data processing, which is perfectly aligned with the layered architecture's principle that each layer only relies on the facilities and services offered by the layer immediately beneath it. The presentation layer can cater two core user interaction requirements: regular users and administrators; The business

logic layer encapsulates core business rules, including payment calculation algorithms and verification logic for reservations. The data access layer implements persistent storage of five kinds of data entity (user, admin, parking spot, reservation and payment), ensuring system's data consistency. Layered architecture's decoupling of layers enables teams to be divided by layers, facilitating independent and parallel development and maintenance.

3.2.2 High Stability and Security

Considering that students and staff may enter, exit, or make reservations all the time, the SPS system requires 24/7 operation. The hierarchical design of layered architecture can effectively contain failures within individual layers. For instance, if the Payment module in the business logic layer fails, users can still perform queries and make reservations. Additionally, security is equally paramount in the SPS system. Examples include that regular users are not permitted to update spot status, and some sensitive data (such as license plate number) must be encrypted. The adoption of layered architecture allows for the development of targeted security measures for each layer, which significantly enhances the overall system safety.

3.2.3 Adapting to Multi-Terminal Access

In SPS practical deployment scenarios, staff and students may access the system via various terminals such as web and mobile terminals (apps/mini programs). The complete decoupling between presentation and business logic layers provides robust support for the requirement of multi-terminal access: it only requires developing the corresponding presentation layer and calling unified interfaces in business logic layer to obtain information, without modifying other layers.

3.4 Support and Limits for Future Expansion

The decoupling of different layers can reduce expansion costs for SPS system. For example, when adding new parking areas and payment methods, the changes are cleanly isolated: extending corresponding data fields in the data layer, adding new components in presentation layer and sub-module in business logic layer. This clear hierarchical modification can avoid large-scale refactoring, greatly simplify the expansion process and reduce the resource investment.

However, the top-down dependencies and rigid layer division can also bring constraints for SPS's high real-time performance and high flexibility requirements. For instance, adding real-time parking spot navigation, one-way dataflow and the multi-layer invocations may cause latency, while shortening the call chain would break the layer structure. This is undoubtedly a main drawback of layered architecture.

Q4. Test Case Design. (10 marks)

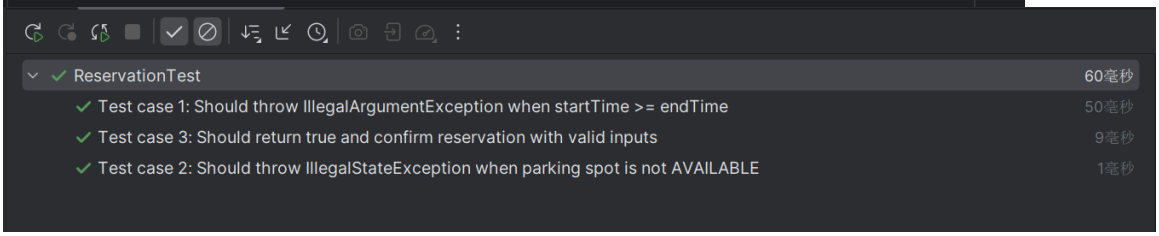
Category	Test Objective	Input Type	Input Value	Output	Note
Valid Test Case 1	To verify that the system successfully creates a reservation when a user selects an available parking spot with valid date and time	ParkingSpot ID: String, User ID: String, Start Time: DateTime, End Time: DateTime	ParkingSpot ID: "A-101", User ID: "U12345", Start Time: "2025-12-15 09:00", End Time: "2025-12-15 17:00"	Success message: "Reservation confirmed for spot A-101". System creates a new Reservation record. ParkingSpot status changes from "Available" to "Reserved". Confirmation email sent to user.	Assumes the parking spot is available and the time slot is within operating hours (06:00-22:00)
Valid	To verify that the	ParkingSpot ID:	ParkingSpot ID:	Success message:	Tests a shorter

Category	Test Objective	Input Type	Input Value	Output	Note
Test Case 2	system correctly handles a reservation made for the immediate next available time slot	String, User ID: String, Start Time: DateTime, End Time: DateTime	“B-205”, User ID: “U67890”, Start Time: “2025-12-15 14:30”, End Time: “2025-12-15 16:00”	“Reservation confirmed for spot B-205”. System creates a new Reservation record. ParkingSpot status updated to “Reserved”. Reservation details stored in database.	duration reservation (1.5 hours) to ensure the system handles various time periods
Invalid Test Case 1	To verify that the system rejects a reservation attempt when the user ID does not conform to the required format	ParkingSpot ID: String, User ID: String, Start Time: DateTime, End Time: DateTime	ParkingSpot ID: “A-101”, User ID: “12345”, Start Time: “2025-12-15 10:00”, End Time: “2025-12-15 12:00”	Error message: “Invalid User ID format. User ID must start with ‘U’ followed by 5 digits”. No Reservation record created. No system data changed. User prompted to correct the input.	Assumes the system requires User ID format to be “U” followed by exactly 5 digits (e.g., U12345)
Invalid Test Case 2	To verify that the system rejects a reservation when the end time is before the start time	ParkingSpot ID: String, User ID: String, Start Time: DateTime, End Time: DateTime	ParkingSpot ID: “C-303”, User ID: “U22222”, Start Time: “2025-12-15 15:00”, End Time: “2025-12-15 13:00”	Error message: “Invalid time range. End time must be after start time”. No Reservation record created. No system data changed. Input form remains active for correction.	Tests logical validation of time inputs
Boundary Test Case	To verify that the system correctly handles a reservation at the minimum allowed duration (e.g., 30 minutes)	ParkingSpot ID: String, User ID: String, Start Time: DateTime, End Time: DateTime	ParkingSpot ID: “D-404”, User ID: “U33333”, Start Time: “2025-12-15 11:00”, End Time: “2025-12-15 11:30”	Success message: “Reservation confirmed for spot D-404”. System creates a new Reservation record. ParkingSpot status changes to “Reserved”. Minimum parking fee calculated.	Assumes the system has a minimum reservation duration of 30 minutes. This tests the lower boundary of acceptable reservation duration

Q5. JUnit Testing (15 marks)

```
    @Test
    @DisplayName("Test case 2: Should throw IllegalStateException when parking spot is not AVAILABLE")
    void testUnavailableParkingSpot() {
        // Create a parking spot with RESERVED status
        ParkingSpot reservedSpot = new ParkingSpot( spotId: 2, ParkingSpot.Status.RESERVED);
        Reservation invalidRes = new Reservation( reservationID: 1004,  userID: 2004, reservedSpot, now, now.plusHours(1));

        assertThrows(IllegalStateException.class, invalidRes::processReservation);
    }

    @Test
    @DisplayName("Test case 3: Should return true and confirm reservation with valid inputs")
    @Timeout(100)
    void testSuccessfulReservation() {
        // Process the reservation with valid inputs
        boolean result = res.processReservation();

        // Verify the method returns true
        assertTrue(result);

        // Verify the reservation is confirmed
        assertTrue(res.isConfirmed());

        // Verify the parking spot status is changed to RESERVED
        assertEquals(ParkingSpot.Status.RESERVED, spot.getStatus());
    }
}
```

```java
import org.junit.jupiter.api.*;
import java.time.LocalDateTime;
import static org.junit.jupiter.api.Assertions.*;

class ReservationTest {
    private Reservation res;  4 个用法
    private ParkingSpot spot;  6 个用法
    private LocalDateTime now;  10 个用法

    @BeforeEach
    void setUp() {
        // Initialize a fixed LocalDateTime value (e.g., a baseline "now")
        now = LocalDateTime.of( year: 2025,  month: 12,  dayOfMonth: 15,  hour: 10,  minute: 0);
        // Initialize a parking spot with AVAILABLE status
        spot = new ParkingSpot( spotId: 1, ParkingSpot.Status.AVAILABLE);
        // Initialize a reservation object
        res = new Reservation( reservationID: 1001,  userID: 2001, spot, now, now.plusHours(2));
    }

    @AfterEach
    void tearDown() {
        // Reset objects used in the test
        res = null;
        spot = null;
        now = null;
    }

    @Test
    @DisplayName("Test case 1: Should throw IllegalArgumentException when startTime >= endTime")
    void testInvalidTimeRange() {
        // Create a reservation where startTime equals endTime
        Reservation invalidRes1 = new Reservation( reservationID: 1002,  userID: 2002, spot, now, now);
        assertThrows(IllegalArgumentException.class, invalidRes1::processReservation);

        // Create a reservation where startTime is after endTime
        Reservation invalidRes2 = new Reservation( reservationID: 1003,  userID: 2003, spot, now.plusHours(2), now);
        assertThrows(IllegalArgumentException.class, invalidRes2::processReservation);
    }
```



Q6. Risk Management. (15 marks)

(a) 4 realistic SPS-related risks

Risk	Risk Type	Source	Likelihood	Severity	Explanation
R1. Payment Gateway Integration Failure	Product Risk	Technology / External Dependency	High	High	Failure or instability in third-party payment APIs may prevent fee collection and break the reservation–payment workflow.
R2. Inaccurate Parking Spot Status Due to Sensor Malfunction	Product Risk	Hardware Technology	Medium	High	Faulty sensors or communication errors may produce incorrect availability data, causing user frustration and system misuse.
R3. System Failure During Peak Usage Periods	Product Risk	Performance / Technical Requirements	High	High	High traffic periods (e.g., semester start) may overwhelm system capacity, resulting in severe slowdowns or crashes that disrupt core SPS services
R4. User Resistance to Adoption	Business Risk	People / Usability	High	Medium	Staff or regular visitors may continue informal parking habits, slowing adoption and reducing the system's perceived value.

(b) and (C) Top Two Highest-Priority Risks and Mitigation Strategies

Risk	Response Strategy	Plan Details	Justification
R1. Payment Gateway Integration Failure	Minimization	1. Integrate redundant payment providers with automatic failover mechanisms. 2. Implement real-time API monitoring and alerting to detect outages immediately. 3. Add retry, timeout, and fallback logic for unstable network connections.	This risk cannot be completely avoided because it depends on external services. Mitigating impact through redundancy and monitoring significantly reduces downtime.
R3. System Failure During Peak Usage Periods	Minimization	1. Conduct rigorous load and stress testing using realistic peak-traffic simulations. 2. Deploy a scalable cloud-native architecture with auto-scaling capability. 3. Optimise database queries, caching strategy, and indexing to eliminate performance bottlenecks.	Performance bottlenecks can be eliminated through design and testing. Prevention is more effective than reacting to outages during critical usage periods.

Q7. Continuous Improvement. (15 marks)

(a) Continuous Improvement Strategies Applied to SPS

1. Strategy 1: CI/CD Pipeline for Post-Release Updates

After the initial deployment of the Smart Parking System on campus, the SPS team can implement a CI/CD pipeline to support future improvement.

How CI/CD is applied specifically in the SPS system:

CI/CD specifically supports continuous improvement in SPS in several key functional areas:

1. Real-time parking availability logic

When developers modify algorithms for real-time spot availability—for example improving sensor data processing or correcting inconsistencies between Reservation and ParkingSpot status—automated tests run to ensure workflows such as viewing spots, reserving spaces, and updating availability remain functional.

2. Reservation workflow validation

Because reservations depend on time validation, spot-state transitions, and conflict checking, CI pipelines run unit and integration tests to prevent regressions whenever reservation logic changes.

3. Payment module updates

If new payment methods (e.g., campus card or mobile wallet) are added, CI/CD verifies their compatibility with Reservation and User modules. Automated security scans ensure sensitive payment data is protected.

How CI/CD improves system quality, team performance, and user satisfaction:

- **System Quality**
  Critical workflows are automatically tested after each change, preventing issues such as double reservations, fee miscalculations, or payment errors, even during rush hours.
- **Team Performance**
  Developers avoid repetitive manual testing and deployment, enabling faster and more reliable delivery of improvements.
- **User Satisfaction**
  Users receive timely fixes and new features, improving trust through more accurate availability and quicker payment confirmation.

2. Strategy 2: Post-Release Refactoring Based on SPS Operational Data

Once SPS is running, logs reveal inefficiencies and bugs not seen during development. Refactoring is continuously performed without changing external functionality, but improving internal structure and maintainability.

How refactoring is applied specifically to the SPS context

1. **Improving ParkingSpot–Reservation interaction**
   To reduce tight coupling, pricing logic is moved into a Payment class—for instance, replacing reservation.calculateFee() with Payment.processPayment(reservation), keeping data and logic together.
2. **Optimizing fee calculation logic**
   If incorrect charges occur (e.g., overnight parking), the pricing logic inside Payment class can be refactored into separate reusable components, making future pricing policy changes easier.
3. **Simplifying AdminManagement**
   Breaking the monolithic AdminManagement class into smaller components (e.g., AvailabilityUpdater, ReportGenerator) improves cohesion and extensibility.

How refactoring improves system quality, team performance, and user satisfaction

- **System Quality**
  A cleaner structure reduces hidden bugs and improves stability during peak usage.
- **Team Performance**
  Modular design accelerates development of new features such as EV-charging spots or visitor passes.
- **User Satisfaction**
  Better performance, fewer errors, and more accurate reservation status directly enhance user experience.

(b)Metrics for Evaluating Effectiveness of Continuous Improvement

1. Deployment Frequency

Measures how often updates and new features are released. It fits SPS well because parking areas, payment policies, and admin requirements change frequently in a campus environment

2. Defect Density and Resolution Time

Tracks how quickly defects—such as double bookings, inaccurate availability, or payment failures—are fixed. These issues directly impact daily parking efficiency and must be resolved rapidly.

3. Customer Satisfaction Score

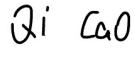
Collected through short in-app surveys after reservations or payments. It is suitable for SPS because user experience depends heavily on accurate availability data and smooth workflows.

Appendix (if any)

Peer review form template

CPT203 Coursework
Peer review
Individual Contribution for Group Report

Group Number: 39

Name	**ID Number**	**Contribution (%)** The sum of this column should be 100	**Signature**
1. Qi Cao	2360908	16.67	
2. Zixi Chen	2362352	16.67	
3. Jingchen Ding	2253158	16.67	
4. Haoyan Xu	2359977	16.67	
5. Jingxuan Weng	2364452	16.67	
6. Shengjie Xu	2360872	16.67	

END