

# Swinburne University of Technology

*School of Science, Computing and Engineering Technologies*

## ASSIGNMENT AND PROJECT COVER SHEET

---

Subject Code: SWE30003

Assignment number and title: 3, Object

Unit Title: Software Architectures and Design

Design Implementation

Due date: 4th August 2024

Project Group: 9

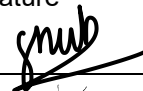

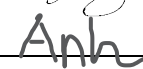
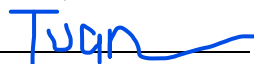
Tutorial Day and Time: Friday 11:00-12:00

Tutor: Dr. Le Minh Duc

---

### To be completed as this is a group assignment

We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
<u>104222196</u>	<u>Ta Quang Tung</u>	<u></u>
<u>104169507</u>	<u>Nguyen Quang Huy</u>	<u></u>
<u>104177513</u>	<u>Tran Hoang Hai Anh</u>	<u></u>
<u>104072029</u>	<u>Phan Sy Tuan</u>	<u></u>

---

Marker's comments:

Total Mark: \_\_\_\_\_

---

### Extension certification:

This assignment has been given an extension and is now due on \_\_\_\_\_

Signature of Convener: \_\_\_\_\_

<b>Introduction.....</b>	<b>2</b>
<b>Detailed object-oriented design.....</b>	<b>3</b>
Class diagram.....	3
Changes.....	4
Classes/interfaces.....	4
Responsibilities and Collaborators.....	5
Bootstrap and Interaction.....	13
<b>Discussion of Assignment 2 Design.....</b>	<b>15</b>
Good Aspects.....	15
Missing Aspects.....	15
Flawed Aspects.....	15
Level of interpretation.....	16
<b>Lessons learned.....</b>	<b>17</b>
Domain driven design.....	17
Easy to maintain.....	17
Loosely coupling.....	17
Software blueprint.....	17
Repository pattern.....	18
<b>Implementation.....</b>	<b>19</b>
Technical details.....	19
Features implemented.....	19
Mapping from code to design.....	20
Platform and setup instructions.....	25
Demonstration.....	26
Scenario 1: Driver signin/signup and Admin signin/signup.....	26
Scenario 2: Driver book a parking slot.....	32
Scenario 3: Admin allocates a slot for a walk-in driver.....	35
Scenario 4: Admin request for statistical report.....	36
<b>References.....</b>	<b>38</b>
<b>Appendix.....</b>	<b>39</b>

# Introduction

This document provides a comprehensive examination of the object-oriented design project focused on developing a smart parking system. Building on the groundwork of 2 previous assignments, this report serves as a complete documentation of the project's design, implementation, and execution phases.

In this assignment, we revisited and refined the initial design concepts, integrating feedback and insights gained from the previous assignments. The report provides an analysis of the design process, detailing the specific changes made to the class structures, responsibilities, and collaborations. It includes a discussion on the quality of the initial design and our reflection on the design journey we have taken.

This report also details our implementation of the design in an OOP language. The implementation phase was carefully executed to align with the refined design, ensuring that all core functionalities were accurately developed and tested. This report documents the technical aspects of the implementation, including the selection of technologies, the development environment, and the coding standards followed. At the end of the report, we provide evidence that shows our implementation works well and covers all the core requirements defined in previous assignments.



Fig 1: Final UML class diagram

## Changes

We have made a number of changes to the initial class design based on the lecturer's feedback and our reconsideration of the design. The following sections list the changes we have made categorized into class, responsibility and collaboration, and bootstrap and interaction levels.

### Classes/interfaces

Here are the changes that we have made to the class list described in Assignment 2:

Classes/interfaces	Change	Classes affected	Justification
Main	Added		The 'Main' class in our system acts like a central access point and helps initialize all of the Repository classes.
AdminRepository, DriverRepository, ParkingSessionRepository, PaymentRepository, SlotRepository	Added		These interfaces provide a template for all of the CRUD interaction with the database layer. This interface then can be implemented by different implementation classes in order not to restrict the data storage option.
MySQLAdminRepository, MySQLDriverRepository, MySQLParkingSessionRepository, MySQLPaymentRepository, MySQLSlotRepository	Added	Admin, Driver, ParkingSession, Payment, ParkingLot	These singleton classes implement their corresponding repository interfaces to perform data retrieval and query with the MySQL database.
Invoice	Discarded	ParkingSession	The only role of the Invoice class is to wrap around a ParkingSession and calculate its cost, which seems insignificant. For that reason, we decided to remove this class and let ParkingSession calculate the parking fee by itself.
ReportData, RevenueData,	Discarded		We find that these classes

FrequencyData			do not make sense because: (1) the data for reports comes from the database, and these classes are nothing but useless wrappers for that data, and (2) the data is closely linked with and should not be separated from the report.
Report to ReportGenerator.  PaperReport to PaperReportGenerator.  WebpageReport to WebpageReportGenerator.  ExcelReport to CSVReportGenerator.	Renamed		The initial logic with the reporting process splits the data and representation into two classes. In addition, two different reports of the same representation would require two Report objects. After some reconsideration, we found that the old Report classes were merely data formatters, and as such they should be singletons. These classes have now been renamed to Generators and they are capable of directly fetching data for formatting.

## Responsibilities and Collaborators

The following describes each class in the above diagram in detail, including their updated responsibilities in the system and collaborations with other classes.

<b>Class name:</b> User	
<b>Parent class(es):</b> None	
<b>Description:</b> An abstract class which captures the key abstractions of a User in the system. It can be specialized into 2 other classes which are Driver and Admin.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows information related to a user such as name, email, etc.	None
Knows credentials related to a user.	None
<b>Can sign up (Added)</b>	<b>None</b>

Can sign in (Added)	None
---------------------	------

<b>Class name:</b> Admin	
<b>Parent class(es):</b> User	
<b>Description:</b> This class represents an admin account. Admins can manage the parking slots as well as monitor parking statistics.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows ID.	None
Can request a report based on given parameters. (Updated)	Report (Updated)
<del>Can create a report based on the report data. (Removed)</del>	<del>ReportData, Report (Removed)</del>
Can sign up (Added)	None
Can sign in (Added)	None

<b>Class name:</b> Driver	
<b>Parent class(es):</b> User	
<b>Description:</b> This class represents a driver account. Drivers can pre-book slots.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows ID.	None
Knows its parking sessions.	ParkingSession
Knows its payments.	Payment
Can pre-book parking slots.	Slot, ParkingSession, OnlinePayment
Can sign up (Added)	None
Can sign in (Added)	None

<b>Class name:</b> ParkingLot	
<b>Parent class(es):</b> None	
<b>Description:</b> Represents the whole parking lot. This class acts similar to the Main class, whose responsibilities is to keep track of many aspects of the parking lot and initialize the	

system on startup.	
Responsibilities	Collaborators
Knows its slots.	Slot
<del>Knows its drivers. (Removed)</del>	<del>Driver (Removed)</del>
<del>Knows its admins. (Removed)</del>	<del>Admin (Removed)</del>
<del>Can modify slots' information (Removed)</del>	<del>Slot (Removed)</del>
Can create walk-in parking sessions.	ParkingSession, Payment
<del>Can initialize the system. (Removed)</del>	<del>Slot, Driver, Admin (Removed)</del>
Can find slot by id (Add)	None

<b>Class name:</b> Slot	
<b>Parent class(es):</b> None	
<b>Description:</b> This class represents a single slot in the parking lot. A slot is aware of when it is occupied.	
Responsibilities	Collaborators
<del>Knows the time intervals in which it is occupied. (Removed)</del>	<del>ParkingInterval (Removed)</del>
Knows its type.	None
<del>Knows its location. (Removed)</del>	<del>None (Removed)</del>
Can check its availability for a given time period. (Changed)	<del>ParkingInterval (Removed)</del> ParkingSession (Added)
Knows its number (Added)	None
Can get direction (Added)	None
Can create a booking. (Added)	Driver, ParkingSession (Added)

<b>Class name:</b> ParkingSession	
<b>Parent class(es):</b> None	
<b>Description:</b> This class represents the booking or parking session for a slot.	
Responsibilities	Collaborators



Knows its slot.	Slot
<del>Knows its parking interval. (Removed)</del>	<del>ParkingInterval (Removed)</del>
Can be confirmed, which marks the chosen slot as occupied.	ParkingSessionRepository (Added)
Can be freed, which marks the chosen slot as available.	Slot
Can calculate the parking cost.	<del>Slot (Removed)</del> None (Updated)
Knows its driver (Added)	Driver
Knows its arrival (Added)	None
Knows its departure (Added)	None

<b>Class name:</b> Payment	
<b>Parent class(es):</b> None	
<b>Description:</b> Abstract class for the different payment methods.	
<b>Responsibilities</b>	<b>Collaborators</b>
<del>Knows the invoice to be paid for. (Removed)</del>	<del>Invoice (Removed)</del>
Knows the session (Added)	ParkingSession
Knows time of payment(Added)	None
Can record payment(Added)	None

<b>Class name:</b> CashPayment	
<b>Parent class(es):</b> Payment	
<b>Description:</b> Represents a physical cash payment made when a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Can calculate the change to return to the driver.	<del>Invoice (Removed)</del>
Knows the amount paid. (Added)	None

<b>Knows the change. (Added)</b>	<b>None</b>
----------------------------------	-------------

<b>Class name:</b> OnlinePayment	
<b>Parent class(es):</b> Payment	
<b>Description:</b> An abstract class for 2 different online payment methods.	
<b>Responsibilities</b>	<b>Collaborators</b>
Can withdraw a given amount of money from an external party. This method must be implemented by the subclasses.	None

<b>Class name:</b> CreditPayment	
<b>Parent class(es):</b> OnlinePayment	
<b>Description:</b> Represents a credit card payment made when a registered driver books a slot or a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the details of the credit card.	None
Can withdraw from the credit card balance. Implements the withdrawal function of OnlinePayment.	None

<b>Class name:</b> OnlineBankingPayment	
<b>Parent class(es):</b> OnlinePayment	
<b>Description:</b> Represents an online banking payment made when a registered driver books a slot or a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the details of the bank account.	None
Can withdraw money from the bank account balance. Implements the withdrawal function of OnlinePayment.	None

<b>Class name:</b> Report(Change) -> ReportGenerator
--

<b>Parent class(es):</b> None	
<b>Description:</b> Abstract class for the different report formats.	
<b>Responsibilities</b>	<b>Collaborators</b>
<del>Knows the report data. (Removed)</del>	<del>ReportData (Removed)</del>
Can fetch data needed for the report. (Added)	ParkingSessionRepository, PaymentRepository (Added)
Can format the report data into a viewable form. This method must be implemented by the subclasses.	<del>ReportData (Change)</del> -> ParkingSession, Payment, Slot
Knows the file extension of the output file. (Added)	None

<b>Class name:</b> PaperReport (Change) -> PaperReportGenerator	
<b>Parent class(es):</b> Report (Change) -> ReportGenerator	
<b>Description:</b> Represents the paper report format.	
<b>Responsibilities</b>	<b>Collaborators</b>
<del>Knows the report data. (Removed)</del>	<del>ReportData (Removed)</del>
Can format the report data into a paper report. Implements the formatting function of Report.	<del>ReportData (Change)</del> -> ParkingSession, Payment, Slot
Knows the file extension of the output file. (Added)	None

<b>Class name:</b> ExcelReport (Change) -> CSVReportGenerator	
<b>Parent class(es):</b> Report (Change) -> ReportGenerator	
<b>Description:</b> Represents the CSV report format.	
<b>Responsibilities</b>	<b>Collaborators</b>
<del>Knows the report data. (Removed)</del>	<del>ReportData (Removed)</del>
Can format the report data into a csv report. Implements the formatting function of Report.	<del>ReportData (Change)</del> -> ParkingSession, Payment, Slot
Knows the file extension of the output file. (Added)	None

<b>Class name:</b> PaperReport (Change) -> WebpageReportGenerator	
<b>Parent class(es):</b> Report (Change) -> ReportGenerator	
<b>Description:</b> Represents the webpage report format.	
<b>Responsibilities</b>	<b>Collaborators</b>
<del>Knows the report data. (Removed)</del>	<del>ReportData (Removed)</del>
Can format the report data into a webpage report. Implements the formatting function of Report.	<del>ReportData (Change) -&gt; ParkingSession, Payment, Slot</del>
Knows the file extension of the output file. (Added)	None

**The classes below are newly added:**

<b>Class name:</b> MySQLAdminRepository	
<b>Parent class(es):</b> AdminRepository	
<b>Description:</b> Implement the AdminRepository interface	
<b>Responsibilities</b>	<b>Collaborators</b>
Can create Admin account	Admin
Can get the Admin information by email	None
Can get the Admin credential by email and password	None
Can get the Admin information by id	None

<b>Class name:</b> MySQLDriverRepository	
<b>Parent class(es):</b> DriverRepository	
<b>Description:</b> Implement the DriverRepository interface	
<b>Responsibilities</b>	<b>Collaborators</b>
Can create Driver account	Driver
Can get the Driver information by email	None
Can get the Driver credential by email and password	None
Can get the Driver information by id	None

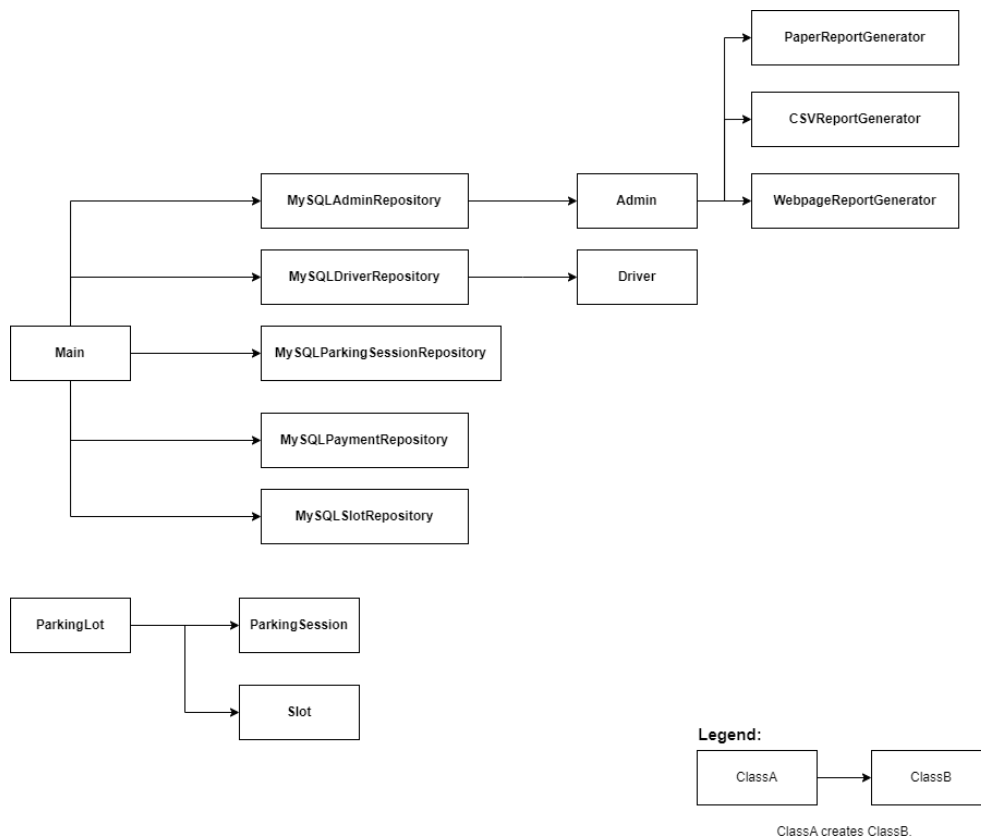
<b>Class name:</b> MySQLParkingSessionRepository	
<b>Parent class(es):</b> ParkingSessionRepository	
<b>Description:</b> Implement the ParkingSessionRepository interface	
<b>Responsibilities</b>	<b>Collaborators</b>
Can create a ParkingSession	ParkingSession
Can get the walk in session by Id	None
Can get a list of ParkingSession	Slot
Can get a ParkingSession by id	None
Can get bookings from the Driver Id	None
Can get those sessions from the walk in session which haven't been paid yet	None

<b>Class name:</b> MySQLPaymentRepository	
<b>Parent class(es):</b> PaymentRepository	
<b>Description:</b> Implement the PaymentRepository interface	
<b>Responsibilities</b>	<b>Collaborators</b>
Can create a Payment	Payment
Can get the Payment by the drivers by Id	None
Can get a list of Payment between two given dates and/or for a given list of slots.	Slot

<b>Class name:</b> MySQLSlotRepository	
<b>Parent class(es):</b> SlotRepository	
<b>Description:</b> Implement the SlotRepository interface	
<b>Responsibilities</b>	<b>Collaborators</b>
Can get information of all the slots	None
Can insert a new Slot	Slot
Can update the information of a Slot	Slot
Can delete an old Slot	Slot

Can get a list of ParkingSession which shows all of the occupied parking Slot	Slot
---	------

## Bootstrap and Interaction



*Fig 2: The updated bootstrap process of the system*

Here is a brief overview of our new bootstrap process:

- We haven't included all the classes related to Payment, namely CashPayment, CreditPayment and OnlineBankingPayment. This change happened because during the coding process, we realized that CashPayment, CreditPayment and OnlineBankingPayment can be initialized by an Express middleware. And for that reason, we don't want to mention them in this bootstrap process diagram.
- In our new bootstrap diagram, the Main class will have the role of the central access point of the system. It holds an instance of all the singleton Repository classes. MySQLAdminRepository and MySQLDriverRepository have their own roles to create the Admin and the Driver, respectively. The admin class statically creates PaperReportGenerator, CSVReportGenerator and WebpageReportGenerator if they do not already exist.
- We also pushed ParkingLot to the left-most part of the bootstrap process since it is a singleton class that is first initialized by an Express middleware.

- ParkingSession and Slot will be created from ParkingLot and we keep this unchanged from the design in Assignment 2.

# Discussion of Assignment 2 Design

## Good Aspects

### Comprehensive Domain Modeling:

The design successfully captured the essential entities and their relationships within the smart parking system. Classes such as User, ParkingLot, Slot, and ParkingSession accurately represent the real-world counterparts and provide a solid foundation for the system.

### Use of Design Patterns:

The use of design patterns like Strategy was well-considered. The Strategy pattern allowed for different payment methods (CashPayment, CreditPayment, OnlineBankingPayment) to be implemented flexibly, enabling the system to adapt to different payment processes without altering the core logic.

### Clear Separation of Concerns:

The system design clearly delineated responsibilities among classes, which helped in maintaining a clean architecture. For example, separating User responsibilities into Admin and Driver subclasses allowed for specific functionalities to be implemented distinctly, ensuring that administrative operations were isolated from user-specific actions.

## Missing Aspects

### The Repository pattern:

The pattern facilitated the separation of business logic and data access. This pattern was employed to manage database operations for entities like User, ParkingSession, and Payment, promoting a clean architecture.

### Detailed Error Handling:

While the core functionalities were outlined, there was a lack of a strategy for error handling and input validation. Including detailed scenarios for potential errors and their corresponding handling mechanisms would enhance the system's robustness.

## Flawed Aspects

### Report Data and Representation Separation:

The initial decision to separate ReportData from Report representation in the design was identified as problematic. This separation was not justified and added unnecessary complexity. A more cohesive design would integrate data and representation, ensuring that reports are generated and presented more logically and efficiently.



Relationships in the Class Diagram:

The initial design exhibited some confusing or incorrect relationships among classes. There were instances of wrong or excessive relationships, which made the diagram harder to interpret and understand. Ensuring accurate and necessary relationships between classes is crucial for maintaining a clear and coherent system architecture.

## Level of interpretation

Clarity and Confusion:

Overall, the initial class diagram provided a good high-level overview of the system, but some areas required additional clarification. Notably, the separation of ReportData from Report representation was confusing and seemed unnecessary, leading to potential misunderstandings about how report data is processed and displayed.

Ambiguities in Relationships:

Some relationships in the class diagram were either unclear or incorrectly represented, which complicated the interpretation of class interactions. These ambiguities necessitated further refinement and clarification to ensure the diagram accurately reflected the intended design.

# Lessons learned

After finishing this course, we have been provided valuable knowledge in order to design a fully functional smart parking system from a high-level design to a technical implementation level of creating it. Looking back on this course, there are several key design principles that're worth note taking when designing a system. Below this are some of the most important criterias:

## Domain driven design

Domain-Driven Design (DDD) is a software development methodology that emphasizes the significance of prioritizing the core business domain while developing complex systems. DDD encourages tight collaboration between technical and business domains to correctly model the project. DDD favors models which are more similar in real life, hence creating a tight relationship between users and enterprises. It is also a common language used by all team members and entrepreneurs to facilitate clear communication and knowledge of the system's target and capabilities.

## Easy to maintain

The project emphasized the need for software design that is adaptable to change. Using design principles that allow for simple updates and expansions guaranteed that the system could adapt to new business demands or technological advancements without requiring large changes. This adaptable strategy helps the system remain relevant and successful over time.

## Loosely coupling

Abstract classes and design patterns are critical for limiting coupling in software design. Abstract classes let developers build shared interfaces and functionality in a base class that can be inherited and customized by subclasses. This centralisation of essential functionality means that changes in one subclass have little influence on the others, resulting in more maintainable code. By using observer, strategy or factory design patterns can further loosen the relationship between different classes.

## Software blueprint

A software blueprint, which includes thorough documentation, flowcharts, data models, a class diagram, and an ER diagram, is critical for understanding system behavior and data flow. Spending time on this design allows the team to eliminate uncertainty and assumptions later on, resulting in a more accurate and successful implementation.

## Repository pattern

The Repository pattern is a design pattern that isolates data access functionality while providing a clean API to the business logic layer. It isolates the business logic from the data access code, making it easier to maintain and test by centralizing data retrieval and manipulation techniques. This pattern facilitates the construction of dummy repositories for testing and ensures that data access functionality evolves independently of the rest of the application. By doing so, we can provide a set of rules that all of the repository classes that implement that interface must follow. By doing so, we can make our system more flexible to change the data storage.

# Implementation

## Technical details

Our group's implementation of this design is a full-stack web application. We have chosen TypeScript as our programming language for the backend as it is very conducive to web development while also bearing the characteristics of an object-oriented language. This allows us to write code for the domain classes that closely matches the class design. In addition, the backend has been implemented as a RESTful API with Express.js. Each key function of the application is made possible with a combination of one or more Express routes. These Express routes and their corresponding middlewares are responsible for instantiating some of the classes.

For the frontend, we have decided that JavaScript with the Vue.js framework will be enough because we want to focus more on the object-oriented aspect of the backend. For data persistence, we have chosen a MySQL database running on localhost.

During development, we tried our best to follow the TypeScript coding standard set by Google (*Google TypeScript Style Guide*, n.d.). That said, due to the time constraints, there may be some deviations in certain parts of the code.

## Features implemented

Due to time limitations, our team has not been able to develop all of the 10 functional requirements specified in the Requirements Specification (see Appendix). We have only been able to implement 6/10 requirements, which are:

- Task 1.1 - Create driver account

- Task 1.3 - Pre-book a parking slot

- Task 1.5 - Park vehicle

- Task 1.6 - Pay for booking

- Task 1.7 - Pay for walk-in

- Task 2.1 - View booking statistics of the parking lot

However, we have decided that these tasks are enough as they form the core functionality of the application. The other four tasks, which are:

- Task 1.2 - Update driver account,

- Task 1.4 - Change a booked slot,

- Task 2.2 - Change existing parking slot,

- Task 2.3 - Add a new parking slot,

are relatively minor and can be implemented with a few simple inserts and updates to the database.

For simplification, we have also abstracted away a number of technical features such as performing bank transactions or providing detailed directions to a given parking slot. To account for these missing features, we have the program return a short message to the user.

## Mapping from code to design

Class name	Class implementation	Responsibility
User	signUp: An abstract function which will be implemented by Admin and Driver. It will be used to sign up an account.	Can sign up
	signIn: An abstract function which will be implemented by Admin and Driver. It will be used to sign in an account.	Can sign in
Admin	signUp: It is an implementation from the User class. It will call the repository of Admin in order to create a new admin account. It will also verify if the email has been used before or not.	Can sign up
	signIn: It is also an implementation which logs the user in the system by checking if the email and password is correct or not.	Can sign in
	requestReport: It will generate the report through start, end time, slot and report type. It returns a Report.	Can request a report based on given parameters.
Driver	signUp: It is an implementation from the User class. It will call the repository of Driver in order to create a new Driver account. It will also verify if the email has been used before or not.	Can sign up
	signIn: It is also an implementation which logs the user in the system by checking if the email and password is correct or not.	Can sign in
ParkingLot	getParkingLot: Call the	Can return a parking lot object

	SlotRepository to return a ParkingLot instance	
	allocateWalkInSession: Create a ParkingSession based on the slot type, arrival and departure time then record it.	Can create walk-in parking sessions.
	getSlot: Return a slot based on the slot number	Can find slot by id
	getSlots: Return a list of slots	Can find slot by id
ParkingSession	recordSession: Call the parking session repository then insert a parking session to it. Return a boolean to check whenever the operation is successful or not.	Can be confirmed, which marks the chosen slot as occupied.
	calculateCost: Return the parking fee based on the departure and arrival time in this class.	Can calculate the parking cost.
	cancel: do cancelation for this operation.	Can be freed, which marks the chosen slot as available.
Slot	createBooking: It checks the availability of a slot then creates a new ParkingSession to record it if the slot is vacant.	Can create a booking.
	isAvailable: This calls the SlotRepository to check the session if it's available or not. This function returns a boolean.	Can check its availability for a given time period.
	getDirections: This function returns a string for the direction to the slot.	Can get direction
Payment	recordPayment: It calls the PaymentRepository to insert a Payment.	Can record payment
CashPayment	Note: This class only returns the amount paid and changes.	Can calculate the change to return to the driver.
OnlinePayment	withdraw: An abstract function which will be implemented in the 2 online payment functions.	Can withdraw a given amount of money from an external party. This method must be implemented by the subclasses.
CreditPayment	withdraw: Implement the	Can withdraw from the credit card

	withDraw abstract function in the OnlinePayment class, It returns an OnlinePaymentResult.	balance. Implements the withdrawal function of OnlinePayment.
OnlineBankingPayment	withDraw: Implement the withDraw abstract function in the OnlinePayment class, It returns an OnlinePaymentResult.	Can withdraw money from the bank account balance. Implements the withdrawal function of OnlinePayment.
ReportGenerator	generateReport: This function calls the ParkingSessionRepository to get those ParkingSession to create a report.	Can fetch data needed for the report.
	formatReport: This abstract function will return a string for the format of the report.	Can format the report data into a viewable form. This method must be implemented by the subclasses.
	getReportFileExtension: This abstract function will return a string for the report file extension.	Knows the file extension of the output file.
CSVReportGenerator	formatReport: Implement the function in the parent abstract class, It runs 2 loops to get the output which consists of slotNumber, Type, bookingCount and revenue.	Can format the report data into a csv report. Implements the formatting function of Report.
	getReportFileExtension: This function returns a string for the file extension name. In this case, It is a CSV so It returns ".csv" file format.	Knows the file extension of the output file.
PaperReportGenerator	formatReport: Implement the function in the parent abstract class, It runs 2 loops to get the output which consists of slotNumber, Type, bookingCount and revenue.	Can format the report data into a paper report. Implements the formatting function of Report.
	getReportFileExtension: This function returns a string for the file extension name. In this case, It is a text file so It returns ".txt" file format.	Knows the file extension of the output file.
WebpageReportGenerator	formatReport: Implement the function in the parent abstract class, It maps the outSlotNumber to multiple table rows. It runs a	Can format the report data into a webpage report. Implements the formatting function of Report.

	loop to calculate the booking count, total revenue and slot type. It returns a website with a table formatted in HTML format.	
	getReportFileExtension: This function returns a string for the file extension name. In this case, It is a web page so It returns ".html" file format.	Knows the file extension of the output file.
MySQLAdminRepository	createAdmin: It runs the SQL in order to insert the Admin class into the database with the information of Name, Email, Password.	Can create Admin account
	getAdminByEmail: It runs the SQL in order to find an Admin by a String input.	Can get the Admin information by email
	getAdminByCredentials: It runs the SQL in order to get the information of the Admin for the SignIn.	Can get the Admin credential by email and password
	getAdminById: It runs the SQL in order to find the Admin by a Number input.	Can get the Admin information by id
MySQLDriverRepository	createDriver: It runs the SQL in order to insert the Driver class into the database with the information of Name, Email, Password, DOB, Phone, Address	Can create Driver account
	getDriverByEmail: It runs the SQL in order to find a Driver by a String input.	Can get the Driver information by email
	getDriverByCredentials: It runs the SQL in order to get the information of the Driver for the SignIn.	Can get the Driver credential by email and password
	getDriverById: It runs the SQL in order to find the Driver by a Number input.	Can get the Driver information by id
MySQLParkingSessionRepository	insertParkingSession: It creates a ParkingSession into the database through a ParkingSession object.	Can create a ParkingSession



	getWalkInSessionById: It returns a newParkingSession if the ParkingSession hasn't been created yet.	Can get the walk in session by Id
	getParkingSession: It returns a ParkingSession list of which are found.	Can get a list of ParkingSession
	getParkingSessionById: It returns a ParkingSession object through a search by Id.	Can get a ParkingSession by id
MySQLPaymentRepository	insertPayment: It inserts the Payment object into the database then it returns the Payment's session Id.	Can create a Payment
	getPayments: It searches for the Payment by StartDate, EndDate and a list of Slot. It runs the SQL to return a list of Payment objects.	Can get a list of Payment between two given dates and/or for a given list of slots.
MySQLSlotRepository	getSlots: It returns all the Slots existing by a list of Slot objects.	Can get information of all the slots
	insertSlot: It inserts a Slot object into the SQL database. Returning a boolean to check whenever the operation is successful or not.	Can insert a new Slot
	updateSlot: It updates a Slot object from the SQL database. Returning a boolean to check whenever the operation is successful or not.	Can update the information of a Slot
	deleteSlot: It deletes a Slot object from the SQL database. Returning a boolean to check whenever the operation is successful or not.	Can delete an old Slot
	getParkingSessionsForSlotBetween: It returns a list of ParkingSession objects through the inputs of Slot, ArrivalDate and DepartureDate.	Can get a list of ParkingSession which shows all of the occupied parking Slot

## Platform and setup instructions

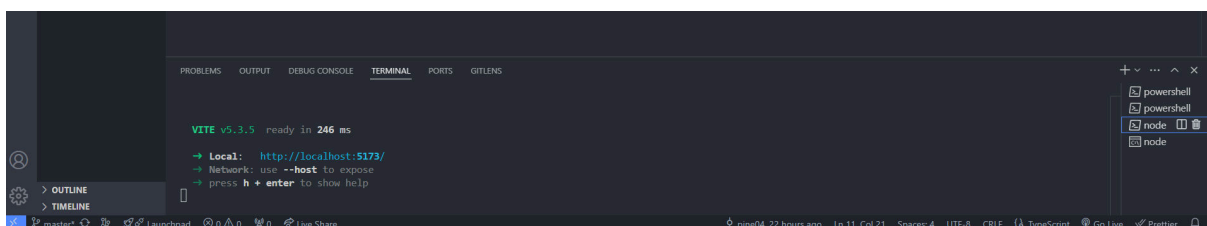
This application has been developed mainly on Windows 11 using Visual Studio Code. It uses npm as the package manager and runs on Node v20.11.1 (although later versions will still work.) It requires a local MySQL server to function.

For those interested in running the application, the source code is available on this GitHub repository: <https://github.com/pine04/smartparking>. Follow these steps to get the application up and running on your machine:

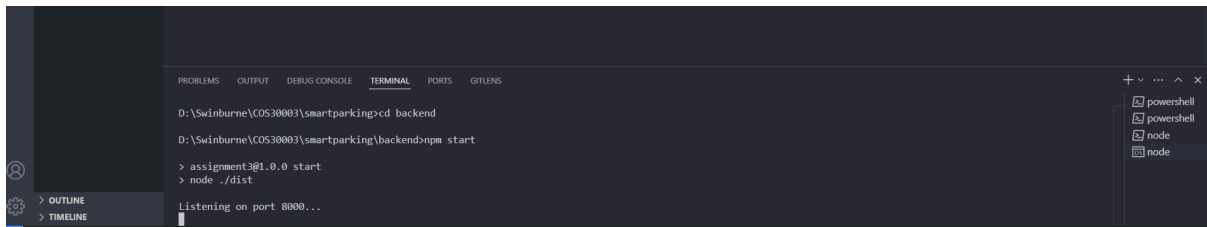
1. The application requires Node.js and MySQL to run. Please follow the instructions provided on their official websites to install them on your machine.
2. Clone the code from the GitHub repository using:

```
git clone https://github.com/pine04/smartparking.git
```

3. After cloning, navigate to the frontend and backend directories and run `npm i` in each to install all the dependencies to your local machine.
4. Navigate to the backend directory and import the `database.sql` code to your local SQL database. After that, navigate to backend/services and modify `mysql_pools.ts` to match your database credentials.
5. Run `npm install -g typescript` to install the Typescript compiler globally. After that, navigate to the backend folder and run `npm run compile` in the command prompt. A folder name `dist` will appear. Run `npm start` to start the backend server.
6. Navigate back to the frontend folder. Use `npm run dev` to start the frontend server and the website is ready to be used at port 5173 with the URL <http://localhost:5173/>



*Fig 3: Successful execution of frontend*



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT LENS
D:\Swinburne\COS30003\smartparking>cd backend
D:\Swinburne\COS30003\smartparking\backend>npm start
> assignment3@1.0.0 start
> node ./dist
Listening on port 8000...
```

*Fig 4: Successful execution of backend*

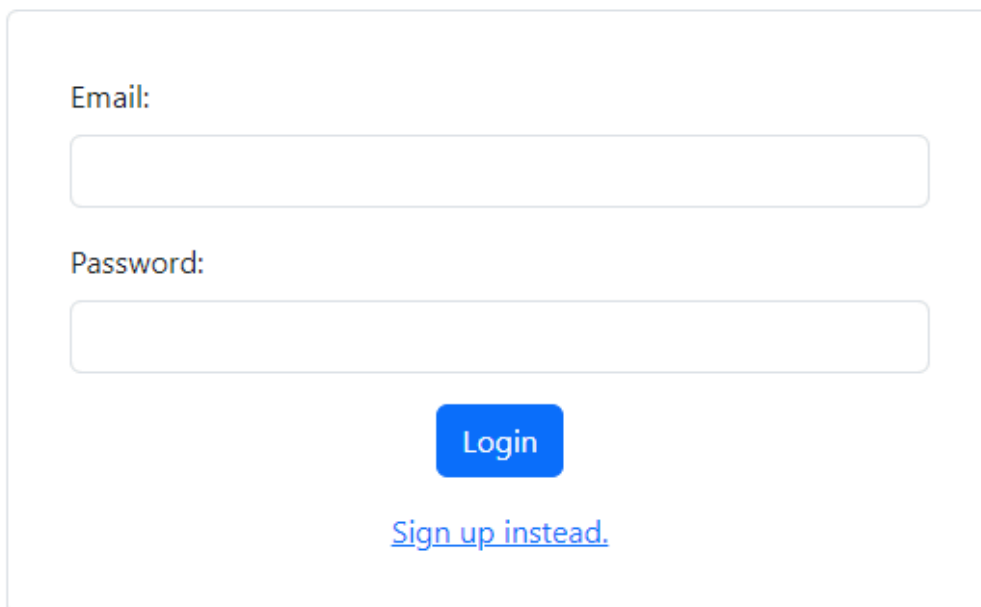
## Demonstration

The following section shows several scenarios of a user interacting with the application to carry out key tasks. A video demonstration of these scenarios is available on [this Google Drive link](#).

### Scenario 1: Driver signin/signup and Admin signin/signup

The user types in their information to signup/login as we can see on Fig 5,6,7,8. If there are any errors with their input, the error message will pop up as shown in Fig 9. After successfully logging into the system. A home screen will display to show the dashboard of the driver or admin as we can see on Fig 10 and Fig 11.

## Driver Login



Email:

Password:

Login

[Sign up instead.](#)

*Fig 5: Driver login form*


# Driver Signup

Name:

Email:

Password:

Date of Birth:



Phone:

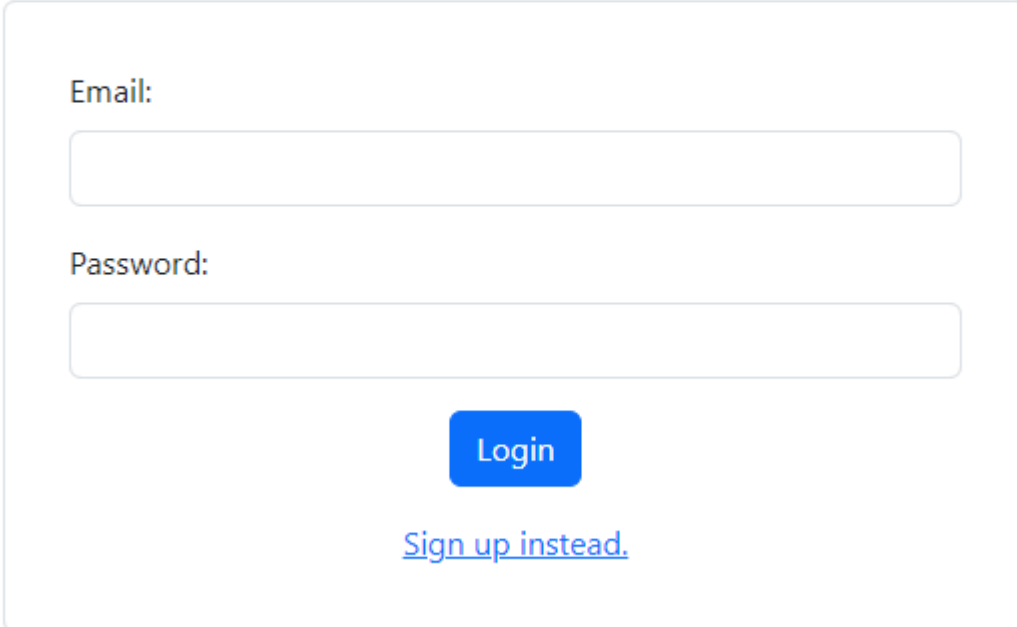
Address:

Signup

[Sign in instead.](#)

Fig 6: Driver signup form

# Admin Login



The form is a light gray rounded rectangle containing two text labels, two input fields, a blue login button, and a link.

Email:

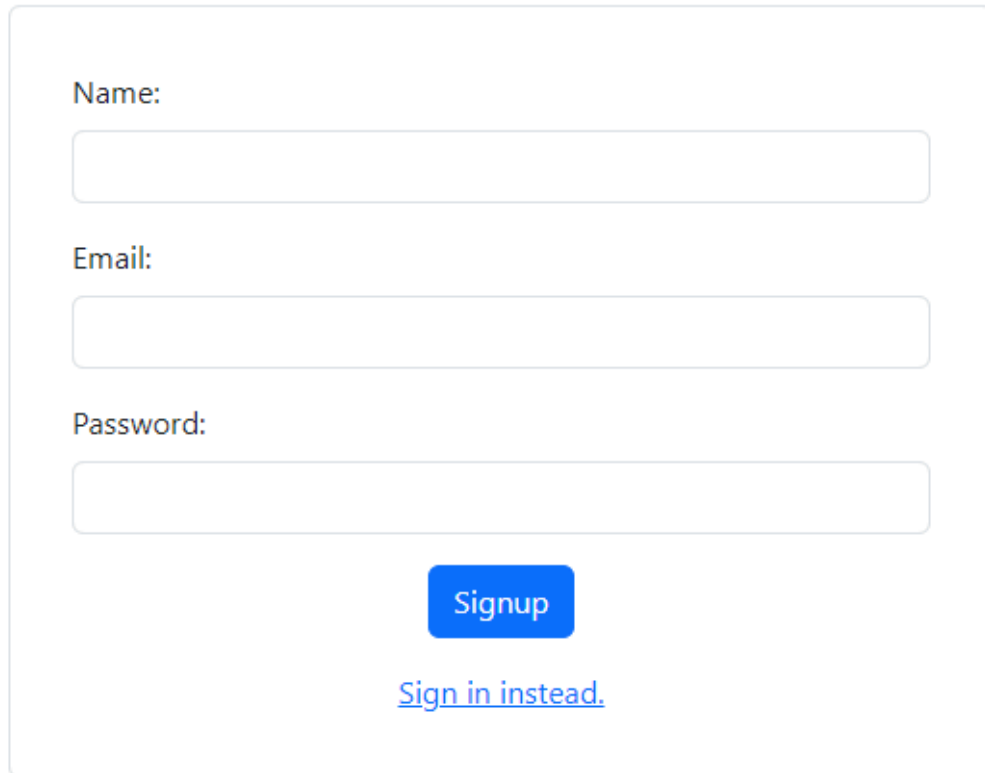
Password:

Login

[Sign up instead.](#)

*Fig 7: Admin login form*

# Admin Signup



A form for admin signup with three input fields and a button. The form is enclosed in a light gray rounded rectangle. The first field is labeled 'Name:', the second 'Email:', and the third 'Password:'. Each label is positioned to the left of its corresponding input field. Below the input fields is a blue button with the text 'Signup'. Below the button is a blue hyperlink that reads 'Sign in instead.'.

Name:

Email:

Password:

[Signup](#)

[Sign in instead.](#)

*Fig 8: Admin signup form*

# Admin Login

Email:

Password:

Login

[Sign up instead.](#)

Email or password is not correct.

Fig 9: Form validation for admin login

## Driver Information

Name: huy

Email: 123@gmail.com

DOB: 2024-08-04T00:00:00.000Z

Phone: 13123123

Address: 313123131

Log out

## Bookings

Book a slot

You have not made any bookings.

## Payments

You have not made any payments.

*Fig 10: Driver dashboard*

## Admin Dashboard

The Admin Dashboard interface consists of two main sections. The top section, titled 'Allocate session for walk-in drivers', contains a 'Log out' button in the top left corner. Below the title, there is a 'Slot type:' dropdown menu with 'Motorcycle' selected. Underneath is a 'Departure time:' field with a date-time picker showing 'mm/dd/yyyy --:-- --'. An 'Allocate' button is positioned at the bottom right of this section. The bottom section, titled 'Record payment', features a 'Select a parking session:' dropdown menu with 'Please select one' as the placeholder. Below this is a 'Select a payment method:' dropdown menu with 'Cash' selected. At the bottom of this section is an 'Amount paid:' text input field. A 'Record' button is located at the bottom left of the 'Record payment' section.

*Fig 11: Admin dashboard*

## Scenario 2: Driver book a parking slot

To perform a booking, the driver must already have an account and is already logged into the system. On the dashboard screen, the user can navigate to the booking form by clicking on the 'book a slot' button as shown in Fig 12. After that, the driver will have access to the booking page as shown in Fig 13. Once the driver has completed filling out the form, they can proceed to the payment process. Fig 14 shows the payment form that the driver needs to fill in their payment information before a message indicates that the user has successfully performed the payment. If the driver clicks the link to go back to the dashboard, the dashboard will display the updated bookings and payments as we can see in Fig 16.



## Driver Information

Name: driver

Email: driver1@gmail.com

DOB: 2024-07-29T00:00:00.000Z

Phone: 123412341

Address: duytan

Log out

## Bookings

Book a slot

You have not made any bookings.

## Payments

You have not made any payments.

*Fig 12: Driver click the 'book a slot' button*

## Booking Page

Select a parking slot:

15 - motorcycle

Arrival time:

08/05/2024 11:36 PM

Departure time:

08/06/2024 11:36 PM

Book

*Fig 13: The driver must fill out the form to perform a booking*

# Payment Page

### Invoice

Booking ID: 3

Arrival: Mon Aug 05 2024 23:36:00

Departure: Tue Aug 06 2024 23:36:00

Slot: 15 - motorcycle

Total amount to be paid: 120000 VND

Select a payment method:

Credit Card

Card Name:

Card Number:

Pay

### Receipt

Total fee: 120000 VND

Payment method: credit

Card name:

Card number:

Fig 14: The payment form that the driver need to fill

# Payment Page

### Invoice

Booking ID: 3

Arrival: Mon Aug 05 2024 23:36:00

Departure: Tue Aug 06 2024 23:36:00

Slot: 15 - motorcycle

Total amount to be paid: 120000 VND

Payment success. Withdrawn 120000 from credit card 13213123 with number 12312312312. [Back to profile.](#)

Select a payment method:

Credit Card

Card Name:

13213123

Card Number:

12312312312

Pay

### Receipt

Total fee: 120000 VND

Payment method: credit

Card name: 13213123

Fig 15: A success message indicate that the user have completed the transaction

## Driver Information

Name: driver

Email: driver1@gmail.com

DOB: 2024-07-29T00:00:00.000Z

Phone: 123412341

Address: duytan

[Log out](#)

## Bookings

[Book a slot](#)

ID: 3

Slot: 15 - motorcycle

Arrival: Mon Aug 05 2024 23:36:00

Departure: Tue Aug 06 2024 23:36:00

## Payments

Session information: From Mon Aug 05 2024 23:36:00 to Tue Aug 06 2024 23:36:00 at slot 15 for .

Payment information: Credit - 13213123 - 12312312312

*Fig 16: Updated driver dashboard*

## Scenario 3: Admin allocates a slot for a walk-in driver

In order to perform session allocation for walk-in drivers, the admin must be inside the system first. As shown in Fig 17, the user has to fill in the form to allocate a session for walk-in first. After filling out all of the basic information and clicking the 'allocate' button, A success message will be displayed and the parking session can be selected in the record payment section. After the walk-in driver has completed the data for the transaction and the 'Record' button is clicked, a success message will be displayed as shown in Fig 18 and the slot allocation is done.

**Allocate session for walk-in drivers**

Slot type:  
Motorcycle

Departure time:  
08/06/2024 03:02 AM

Allocated slot successfully. Directions: Enter and go straight, then turn right at the first column. Look for a spot near the wall.

Allocate

**Record payment**

Select a parking session:

Please select one

1 - motorcycle - Tue Aug 06 2024 00:03:31 - Tue Aug 06 2024 00:02:00

1 - motorcycle - Tue Aug 06 2024 00:04:07 - Tue Aug 06 2024 03:02:00

Card Name:  
eqewq

Card Number:  
eqwewqe

Record

*Fig 17: Form for session allocation*

**Record payment**

Select a parking session:

Select a payment method:  
Credit Card

Card Name:  
huy

Card Number:  
12341234

Payment success. Withdrawn 14823.611111111111 from credit card huy with number 12341234.

Record

*Fig 18: A success message showing that the payment has been made*

## Scenario 4: Admin request for statistical report

To request a statistical report, the admin will have to log in to the system. When the admin navigates to the bottom of the dashboard page, there will be a form for the admin to request the statistical report (Fig 19). Admin may need to fill out the form to get the statistics that they want to look into. There will be 3 kinds of reports that the admin may be interested in, including web page reports, CSV reports, or paper reports. Based on the choice of the admin, the system will output the corresponding report including all of the information.

### Request statistics report

Select parking slots:

Select one or more slots. Ignore to select all.

- 1 - motorcycle
- 2 - motorcycle
- 3 - motorcycle
- 4 - motorcycle
- 5 - motorcycle
- 6 - motorcycle
- 7 - motorcycle
- 8 - motorcycle
- 9 - motorcycle
- 10 - motorcycle

Arrival time:

08/04/2024 12:23 AM

Departure time:

08/07/2024 12:23 AM

Select report type:

Webpage

Download report

Fig 19: Form for admin to request statistic report

## SmartParking Report between Sun Aug 04 2024 and Wed Aug 07 2024

Slot number	Slot type	# of parking sessions	Revenue
55	motorcycle	1	5000 VND
71	car	1	14754.166666666666 VND
15	motorcycle	1	120000 VND
1	motorcycle	2	14697.222222222222 VND

Fig 20: Webpage statistic report

```
Slot Number,Slot Type,# of parking sessions,Total revenue
55,motorcycle,1,5000 VND
71,car,1,14754.166666666666 VND
15,motorcycle,1,120000 VND
1,motorcycle,2,14697.222222222222 VND
```

Fig 21: CSV statistic report

This is the statistics for SmartParking between Sun Aug 04 2024 and Wed Aug 07 2024:

Slot Number - Slot Type - # of parking sessions - Total revenue

55 - motorcycle - 1 - 5000 VND

71 - car - 1 - 14754.166666666666 VND

15 - motorcycle - 1 - 120000 VND

1 - motorcycle - 2 - 14697.222222222222 VND

Fig 22: Paper statistic report

# References

*Google TypeScript Style Guide*. (n.d.). Google. Retrieved August 6, 2024, from  
<https://google.github.io/styleguide/tsguide.html>

# Appendix

Attached on the remainder of the pages is the Assignment 2 report that this document was based on.

<b>Executive Summary.....</b>	<b>2</b>
<b>Introduction.....</b>	<b>3</b>
Outlook of solution.....	3
Interface guidelines.....	3
<b>Problem Analysis.....</b>	<b>4</b>
Goals.....	4
Assumptions.....	4
Simplifications.....	5
<b>Object Design.....</b>	<b>6</b>
Candidate Classes.....	6
Discarded Classes.....	6
Design Justification.....	6
Class Diagram.....	8
CRC Cards.....	8
Design Quality.....	15
Design Heuristics.....	15
Design Patterns.....	16
Bootstrap Process.....	17
Verification.....	19
Pre-book a slot and pay for booking.....	19
Park vehicle.....	20
Pay for walk-in.....	21
View parking lot statistics.....	22
<b>Appendix.....</b>	<b>23</b>



# Executive Summary

SmartParking is a business that provides paid parking facilities for cars and motorbikes on Duy Tan Street. The current setup requires drivers to find and pay for their parking spots in person when they arrive, which can lead to extended waiting periods and the risk of not being able to find a parking slot. To address this issue, SmartParking's owners are planning to introduce an Online Smart Parking Space (OSPS). This system will allow customers to reserve parking spaces online and manage bookings, payments, and parking directions. For the owners, it offers improved administration of the parking areas.

In this report, we will delve deeper into the given case study by showcasing an initial object-oriented design for the system. After identifying and justifying all the candidate classes for our solution, we will develop the CRC card design. This will provide a brief description of each class, its responsibilities, and the collaborators required for each responsibility. Following this, the report will illustrate any Design Patterns or Design Heuristics that have been used in the initial design, providing reasons for their inclusion. Lastly, to verify the design, four typical, non-trivial interaction patterns or scenarios are presented, which will help demonstrate how the system determines the validity of user input and how it applies to specific business activities or processes.

# Introduction

This document describes the analysis used to determine the classes, interfaces, and guidelines required for the system. It serves as a resource for developers, managers, and customers to share information about system design and future testing.

## Outlook of solution

The proposed solution involves carefully planning the roles and interactions of different parts of the Online Smart Parking Space (OSPS). By using Responsibility-Driven Design, we aim to clearly define what each part of the system should do and how they should work together. This design is built to be flexible, scalable, and modular, meaning it can easily be expanded or modified in the future. We have utilized tools like class diagrams and CRC cards to visually show and describe the system's components, making it easier for stakeholders to understand the design. This approach is intended to create a robust, efficient and applicable OSPS.

## Interface guidelines

The following table describes the naming convention we use for various programming constructs in this document.

Type	Convention	Example
Classes	PascalCase	MyClass
Objects	camelCase	myObject
Functions or methods	camelCase	myFunction()
Properties, attributes, or parameters	camelCase	myProperty

# Problem Analysis

## Goals

The goal of this project is to devise a preliminary object-design for the online smart parking system (OSPS) as per the Requirement Specification document (see Appendix). The design is expected to support the functions and quality attributes defined therein, which are:

- F1** Create driver account
- F2** Update driver account
- F3** Pre-book a parking slot
- F4** Change a booked slot
- F5** Park vehicle
- F6** Pay for booking
- F7** Pay for walk-in
- F8** View booking statistics of the parking lot
- F9** Change existing parking slot
- F10** Add a new parking slot

And:

- Q1** Security
- Q2** Performance
- Q3** Scalability
- Q4** Usability
- Q5** Portability

## Assumptions

On top of the assumptions listed in the Requirement Specification (see Appendix), the design described in this document makes a number of additional assumptions about the system, which are:

- A1** SmartParking is not interested in saving detailed information about the vehicles that arrive at the parking lot, except for their license plate number.
- A2** The information of walk-in drivers does not have to be saved.
- A3** Only one vehicle can park in a slot at any given time.
- A4** Walk-in drivers must confirm their departure time on arrival.
- A5** Currently, the owners of SmartParking are only interested in monitoring

revenue and frequency data and they would like to see it displayed in paper, Excel, or web format. However, the system will be designed to easily support more data types and formats in the future.

**A6** The system operates on a database, which is not mentioned in the design at this level. It is implicitly understood that classes will interact with the database to fetch and save data as needed.

## Simplifications

Based on the assumptions described above, the design has made a number of simplifications on the class hierarchy, which are:

**S1** The only relevant piece of information related to vehicles is their license plate numbers. As such, no separate Vehicle class is needed, and this information can be an attribute of ParkingSession.

**S2** The Driver class represents a driver who has registered an account on the system who can pre-book slots. Walk-in drivers are not represented by this class or any other class.

# Object Design

## Candidate Classes

- User
  - Admin
  - Driver
- ParkingLot
- Slot
- ParkingSession
- ParkingInterval
- Invoice
- Payment
  - CashPayment
  - OnlinePayment
    - CreditPayment
    - OnlineBankingPayment
- Report
  - PaperReport
  - ExcelReport
  - WebpageReport
- ReportData
  - RevenueData
  - FrequencyData

## Discarded Classes

- Booking
- SlotType
- Receipt

## Design Justification

After carefully considering which classes to include in the design, we have come up with the complete candidate class list. Here is a brief explanation of why those classes should be chosen:

- ParkingLot, Slot, ParkingSession, and ParkingInterval play a role in the parking and booking process. In general, they keep track of the availability of parking slots and allocate drivers to their slots respectively.
- Invoice and Payment help to handle all the payment processes in the system. We have specialized Payment into smaller classes to take advantage of polymorphism. CashPayment, CreditPayment, and OnlineBankingPayment can all be treated as a Payment in our system.
- Report and ReportData help the owners to monitor the parking lot data, namely RevenueData and FrequencyData, and display it. Based on the needs of the admin, we can choose one of three Report types: PaperReport, ExcelReport, or WebpageReport.

Besides the chosen candidate class, we also discard some classes:

- The Booking class was originally designed to track user bookings, such as time and slot. However, we have decided to merge it with the ParkingSession class, which can keep track of both the booking and the parking session of the driver.
- SlotType only stores the vehicle type of each slot, so it is not necessary to make it a separate class in the system. It can instead be an attribute of Slot.
- Receipt was originally included in the domain model and we initially considered this class. However, as Payment and its subclasses have all the payment-related information and are more or less a representation of Receipt, we have decided to exclude this class.

# Class Diagram

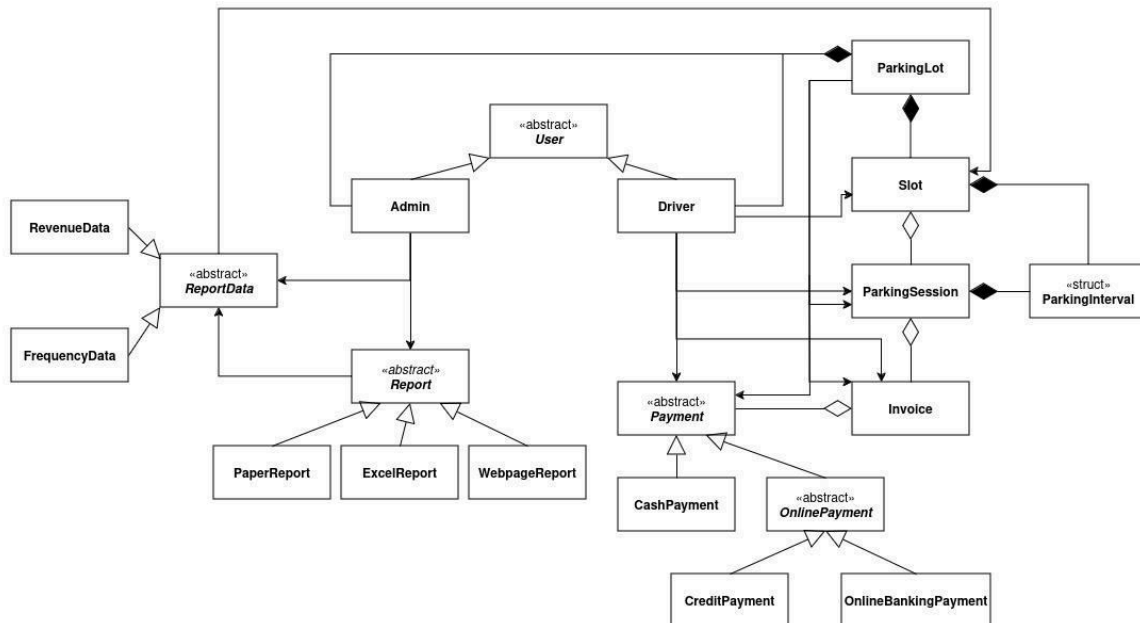


Fig. 1: The UML class diagram of the design.

## CRC Cards

The following describes each class in the above diagram in detail, including their responsibilities in the system and collaborations with other classes.

<b>Class name:</b> User	
<b>Parent class(es):</b> None	
<b>Description:</b> An abstract class which captures the key abstractions of a User in the system. It can be specialized into 2 other classes which are Driver and Admin.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows information related to a user such as name, age, etc.	None
Knows credentials related to a user.	None

<b>Class name:</b> Admin	
<b>Parent class(es):</b> User	
<b>Description:</b> This class represents an admin account. Admins can manage the parking	

slots as well as monitor parking statistics.	
Responsibilities	Collaborators
Knows ID.	None
Can request report data.	ReportData
Can create a report based on the report data.	ReportData, Report

<b>Class name:</b> Driver	
<b>Parent class(es):</b> User	
<b>Description:</b> This class represents a driver account. Drivers can pre-book slots.	
Responsibilities	Collaborators
Knows ID.	None
Knows its parking sessions.	ParkingSession
Knows its payments.	Payment
Can pre-book parking slots.	Slot, ParkingSession, Invoice, OnlinePayment

<b>Class name:</b> ParkingLot	
<b>Parent class(es):</b> None	
<b>Description:</b> Represents the whole parking lot. This class acts similar to the Main class, whose responsibilities is to keep track of many aspects of the parking lot and initialize the system on startup.	
Responsibilities	Collaborators
Knows its slots.	Slot
Knows its drivers.	Driver
Knows its admins.	Admin
Can modify slots' information.	Slot
Can create walk-in parking sessions.	ParkingSession, Invoice, CashPayment
Can initialize the system.	Slot, Driver, Admin



<b>Class name:</b> Slot	
<b>Parent class(es):</b> None	
<b>Description:</b> This class represents a single slot in the parking lot. A slot is aware of when it is occupied.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the time intervals in which it is occupied.	ParkingInterval
Knows its type.	None
Knows its location.	None
Can check its availability for a given time period.	ParkingInterval

<b>Class name:</b> ParkingSession	
<b>Parent class(es):</b> None	
<b>Description:</b> This class represents the booking or parking session for a slot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows its slot.	Slot
Knows its parking interval.	ParkingInterval
Can be confirmed, which marks the chosen slot as occupied.	Slot
Can be freed, which marks the chosen slot as available.	Slot
Can calculate the parking cost.	Slot

<b>Class name:</b> ParkingInterval	
<b>Parent class(es):</b> None	
<b>Description:</b> A data class with only two time values that specify the arrival and departure time of a parking session.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows its arrival and departure time.	None

<b>Class name:</b> Invoice	
<b>Parent class(es):</b> None	
<b>Description:</b> Represents an invoice for a given ParkingSession, which informs the user of the amount of money that they have to pay.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows its parking session.	ParkingSession
Knows the amount of money to be paid.	None

<b>Class name:</b> Payment	
<b>Parent class(es):</b> None	
<b>Description:</b> Abstract class for the different payment methods.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the invoice to be paid for.	Invoice

<b>Class name:</b> CashPayment	
<b>Parent class(es):</b> Payment	
<b>Description:</b> Represents a physical cash payment made when a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Can calculate the change to return to the driver.	Invoice

<b>Class name:</b> OnlinePayment	
<b>Parent class(es):</b> Payment	
<b>Description:</b> An abstract class for 2 different online payment methods.	
<b>Responsibilities</b>	<b>Collaborators</b>
Can withdraw a given amount of money from an external party. This method must be implemented by the subclasses.	None

<b>Class name:</b> CreditPayment	
<b>Parent class(es):</b> OnlinePayment	
<b>Description:</b> Represents a credit card payment made when a registered driver books a slot or a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the details of the credit card.	None
Can withdraw from the credit card balance. Implements the withdrawal function of OnlinePayment.	None

<b>Class name:</b> OnlineBankingPayment	
<b>Parent class(es):</b> OnlinePayment	
<b>Description:</b> Represents an online banking payment made when a registered driver books a slot or a walk-in driver leaves the parking lot.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the details of the bank account.	None
Can withdraw money from the bank account balance. Implements the withdrawal function of OnlinePayment.	None

<b>Class name:</b> Report	
<b>Parent class(es):</b> None	
<b>Description:</b> Abstract class for the different report formats.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the report data.	ReportData
Can format the report data into a viewable form. This method must be implemented by the subclasses.	ReportData

<b>Class name:</b> PaperReport	
<b>Parent class(es):</b> Report	
<b>Description:</b> Represents the paper report format.	

<b>Responsibilities</b>	<b>Collaborators</b>
Knows the report data.	ReportData
Can format the report data into a paper report. Implements the formatting function of Report.	ReportData

<b>Class name:</b> ExcelReport	
<b>Parent class(es):</b> Report	
<b>Description:</b> Represents the Excel report format.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the report data.	ReportData
Can format the report data into an Excel report. Implements the formatting function of Report.	ReportData

<b>Class name:</b> WebpageReport	
<b>Parent class(es):</b> Report	
<b>Description:</b> Represents the webpage report format.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the report data	ReportData
Can format the report data into a web page report. Implements the formatting function of Report.	ReportData

<b>Class name:</b> ReportData	
<b>Parent class(es):</b> None	
<b>Description:</b> Abstract class for the different types of report data that SmartParking is interested in.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the start and end dates of the data of interest.	None
Knows the slots of interest.	Slot
Can query the report data from an external source. This method must be implemented	Slot

by the subclasses.	
Can process the report data based on different rules. This method must be implemented by the subclasses.	None

<b>Class name:</b> RevenueData	
<b>Parent class(es):</b> ReportData	
<b>Description:</b> Represents the revenue data of the parking lot. Can be given start and end dates and slot information to limit the search.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the start and end dates of the data of interest.	None
Knows the slots of interest.	Slot
Can query the report data from an external source. Implements the fetching function of ReportData.	Slot
Can process the report data based on different rules. Implements the processing function of ReportData.	None

<b>Class name:</b> FrequencyData	
<b>Parent class(es):</b> ReportData	
<b>Description:</b> Represents the frequency data of the parking lot. Can be given start and end dates and slot information to limit the search.	
<b>Responsibilities</b>	<b>Collaborators</b>
Knows the start and end dates of the data of interest.	None
Knows the slots of interest.	Slot
Can query the report data from an external source. Implements the fetching function of ReportData.	Slot
Can process the report data based on different rules. Implements the processing function of ReportData.	None

# Design Quality

## Design Heuristics

The following design heuristics have guided the design process. They have been quoted directly from the book Object-Oriented Design Heuristics by Arthur J. Riel.

**Heuristic 1:** A class should capture one and only one key abstraction.

**Heuristic 2:** Keep related data and behavior in one place.

**Heuristic 3:** Be sure the abstractions that you model are classes and not simply the roles objects play.

**Heuristic 4:** Distribute system intelligence horizontally as uniformly as possible.

**Heuristic 5:** Do not create god classes/objects in your system.

**Heuristic 6:** Eliminate irrelevant classes from your design.

**Heuristic 7:** Eliminate classes that are outside the system.

**Heuristic 8:** Do not turn an operation into a class.

**Heuristic 9:** Minimize the number of classes with which another class collaborates.

**Heuristic 10:** If a class contains objects of another class, the containing class should send messages to the contained objects.

**Heuristic 11:** Inheritance should be used only to model a specialization hierarchy.

**Heuristic 12:** Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

**Heuristic 13:** Inheritance hierarchies should be no deeper than an average person can keep in their short-term memory.

**Heuristic 14:** All abstract classes must be base classes.

**Heuristic 15:** All base classes should be abstract classes.

**Heuristic 16:** Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.

**Heuristic 17:** If two or more classes have common data and behavior, they should each inherit from a common base class.

**Heuristic 18:** If two or more classes share only a common interface, they should inherit from a common base class only if they will be used polymorphically.

**Heuristic 19:** It should be illegal for a derived class to override a base class method with a NOP method.

**Heuristic 20:** When given a choice between a containment relationship and an association relationship, choose the containment relationship.

## Design Patterns

### ***Singleton Pattern:***

**Usage:** ParkingLot

**Justification:** Ensures that only one instance of the ParkingLot exists, providing a single point of control for system operations. This pattern is crucial for managing shared resources consistently.

### ***Strategy Pattern:***

**Usage:** OnlinePayment, Report.

**Justification:** Using Strategy Pattern on OnlinePayment allows the system to switch between different payment methods dynamically. The Payment class uses different strategies, namely CreditPayment and OnlineBankingPayment to process payments, making the system flexible and extensible. This pattern can also be used for Report class as it can shift between ExcelReport, PaperReport and WebPageReport based on the requirement of the admin.

### ***Template Method Pattern:***

**Usage:** ReportData

**Justification:** Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm. The ReportData class uses this pattern to allow different report data types (RevenueData, FrequencyData) to implement specific steps of the data fetching process while maintaining a common structure.

## Bootstrap Process

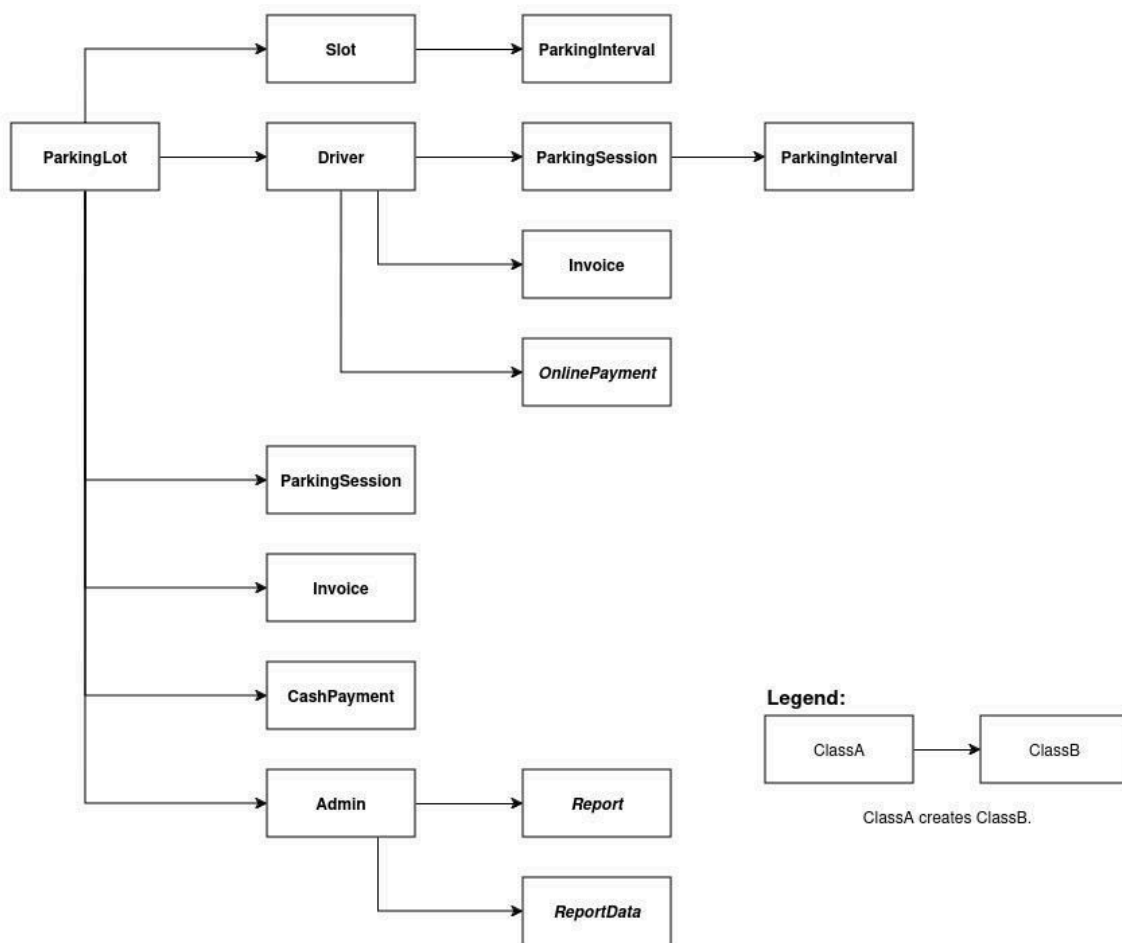


Fig. 2: The bootstrap process.

The bootstrap process of the design is reflected in the above image. It is designed to fully support the functional requirements defined in the Appendix and goes as follows:

1. On startup, the **ParkingLot** class is created first by the executing process. This process could be something similar to the `main()` method.
2. The **ParkingLot** class then initializes the system by creating instances of the **Slot**, **Driver**, and **Admin** classes. It is assumed that the data needed to populate these instances comes from a database, although it is not included in the diagram.
3. During the execution of the system, the **ParkingLot**, **Driver**, and **Admin** instances can create instances of other classes.



4. To keep track of the time periods in which it is occupied or booked, a Slot instance can create instances of the ParkingInterval data class.
5. To allow driver accounts to be added and updated (tasks 1.1 and 1.2), ParkingLot can add or update instances of Driver as needed.
6. To support the task of pre-booking and paying for booked slots (tasks 1.3 and 1.6), a Driver can create a ParkingSession, an Invoice, and an instance of one of OnlinePayment's subclasses to cover the Invoice.
7. To allow bookings to be updated (task 1.4), an instance of Driver can update or remove their ParkingSession instances as needed.
8. To accommodate walk-in drivers and record their payment in the system (tasks 1.5 and 1.7), ParkingLot can create a ParkingSession, an Invoice, and a CashPayment for this session.
9. To enable the owners of SmartParking to view the lot's statistics (task 2.1), an Admin first creates an instance of one of ReportData's subclasses. To format the data, it then creates an instance of one of Report's subclasses, passing it the ReportData.
10. To allow slots to be added and updated (tasks 2.2 and 2.3), ParkingLot can add, update, or remove instances of Slot as needed.

## Verification

The following scenarios illustrate how the design supports the key functionalities defined in the Requirement Specifications. Note that the attribute and method names in these diagrams are tentative and meant to illustrate the scenarios only. Future works will refine them as necessary.

### Pre-book a slot and pay for booking

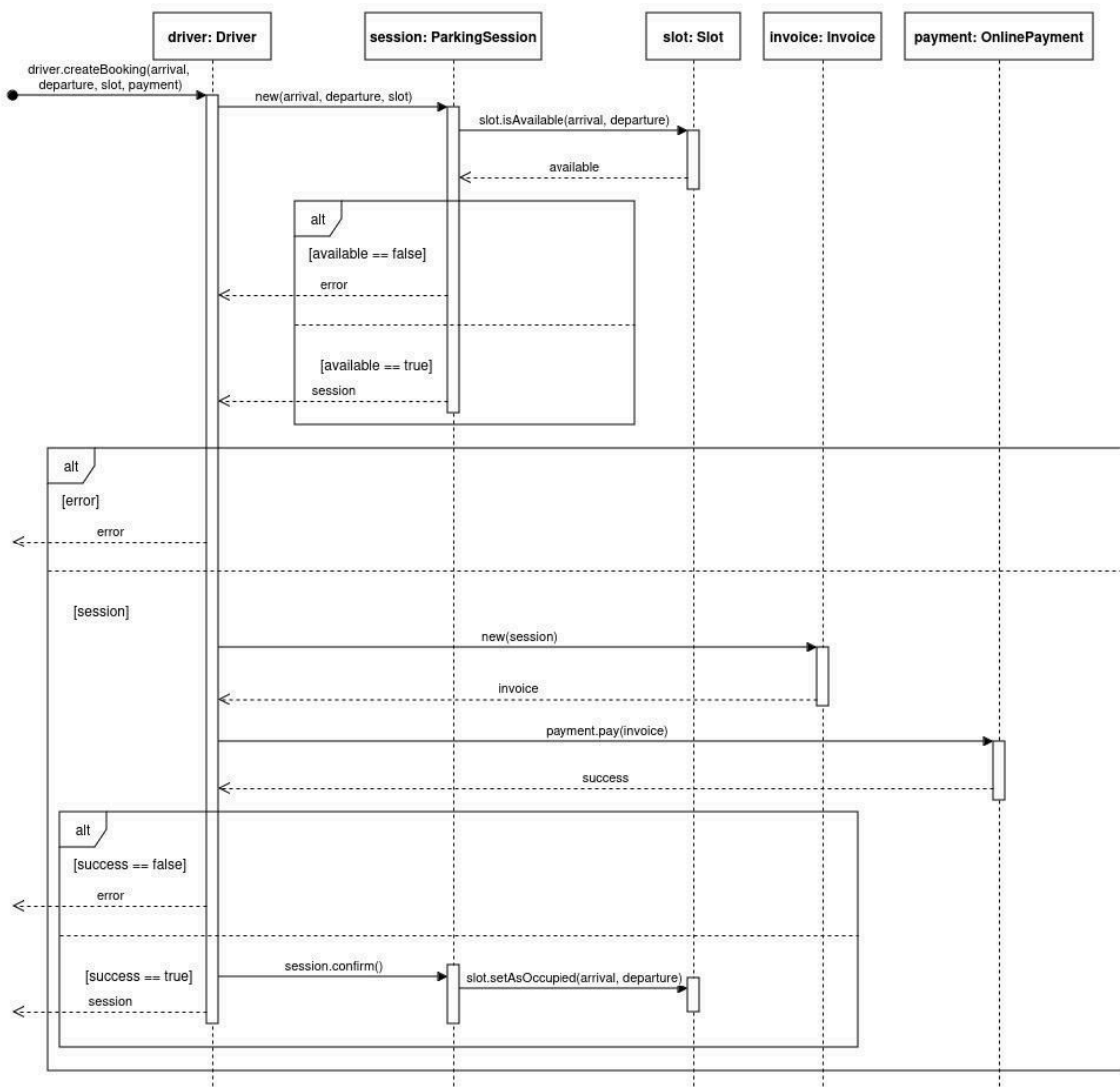


Fig. 3: Pre-book a slot and pay for booking

To begin the booking process, a message is sent to a Driver instance containing the desired parking time, slot, and online payment method. Driver then attempts to create a ParkingSession, which succeeds only if the Slot is available in the specified time period. If it

is unavailable, an error is returned. Otherwise, Driver creates a new Invoice instance, passing in the ParkingSession, and then tells the OnlinePayment instance to pay it. Note that OnlinePayment is an abstract class, and in reality an instance of either CreditPayment or OnlineBankingPayment is used here. The payment call returns a boolean flag indicating success or failure. If the payment fails, an error is returned. Otherwise, the parking session is confirmed and the Slot is marked as occupied for the time period.

## Park vehicle

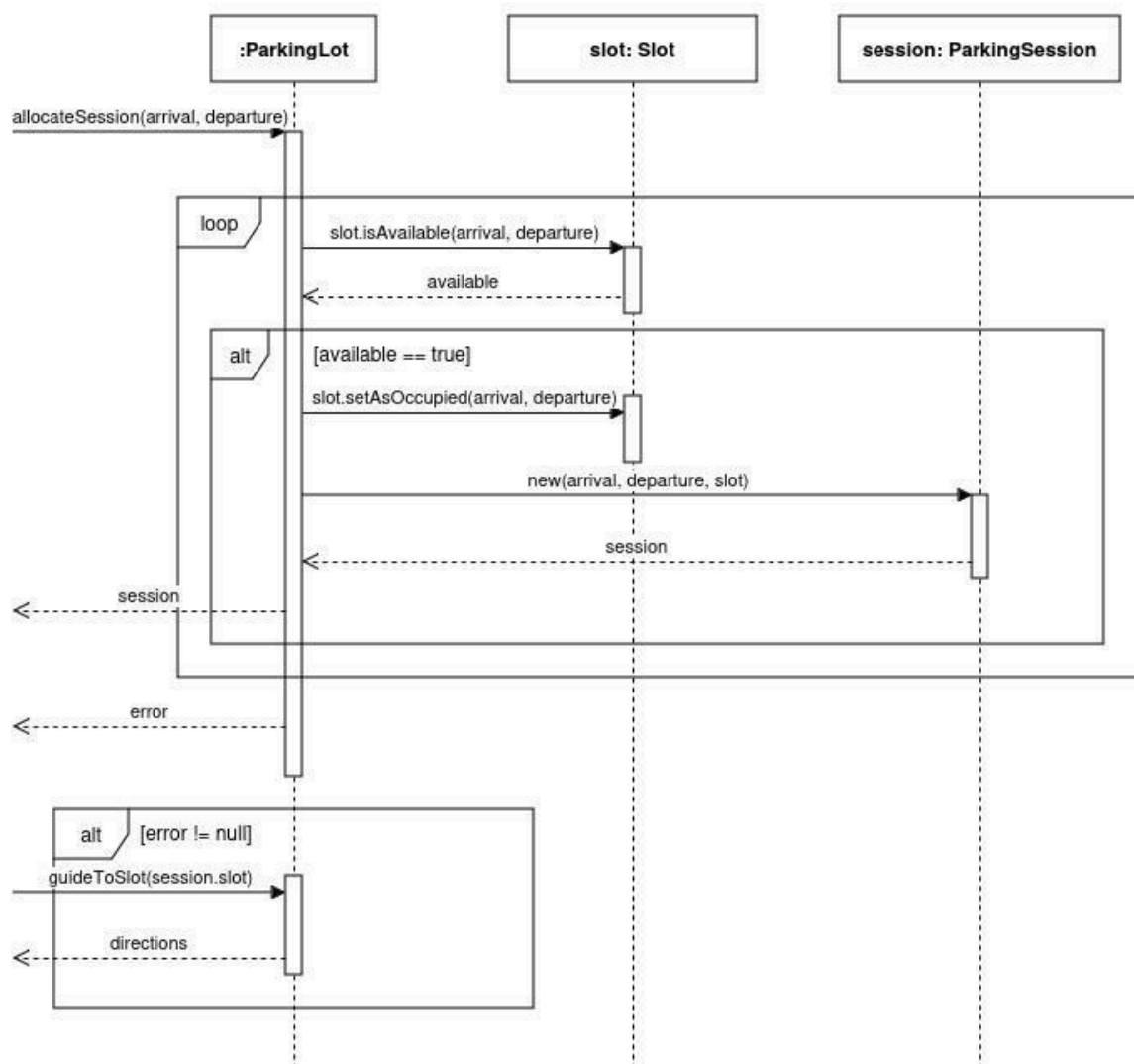


Fig. 4: Park vehicle

When a driver arrives at the parking lot, one of two things can happen. The driver may already have a booking and can drive directly to their slot. However, some drivers may not have booked beforehand and the system must allocate them a slot. The diagram above illustrates this case. The ParkingLot first iterates through all slots and checks if one is

available. If none is available, an error is thrown. Otherwise, the available slot is marked as occupied and a new ParkingSession is created and returned to the caller. If the caller receives a ParkingSession, it will send a message to ParkingLot to guide the driver to the slot.

## Pay for walk-in

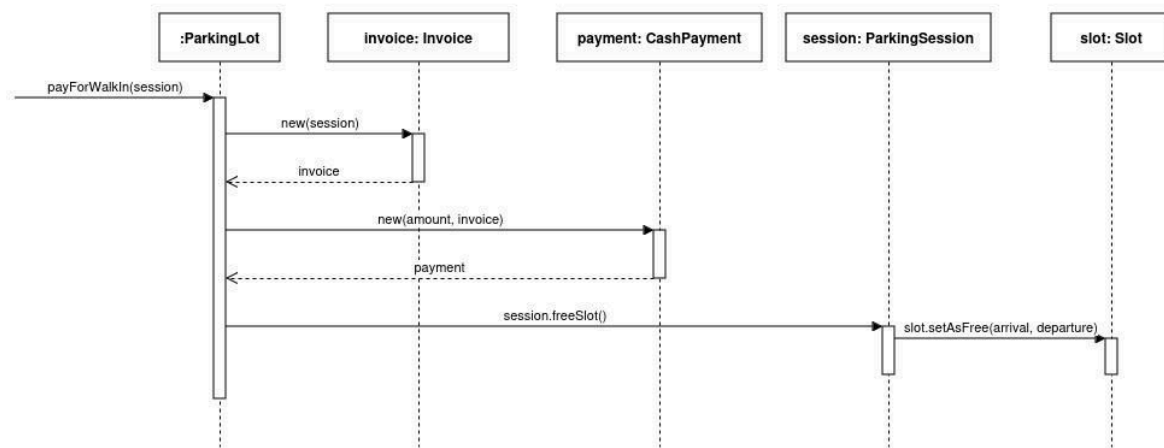


Fig. 5: Pay for walk-in

This scenario illustrates the payment process for walk-in drivers. First, a message is sent to ParkingLot containing the ParkingSession to pay for. Then, an Invoice is generated for this session. Next, ParkingLot creates an appropriate Payment object based on the driver's payment option. In this example, CashPayment has been chosen by the driver. After the payment is complete, ParkingLot tells ParkingSession to free the Slot for future parkers.

## View parking lot statistics

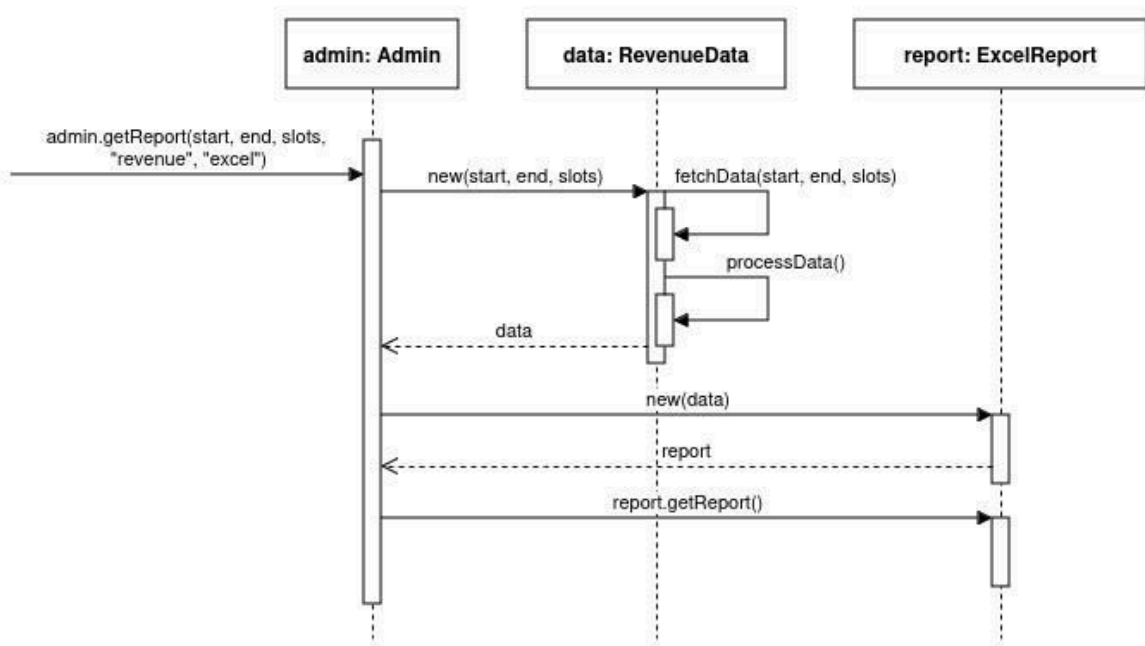


Fig. 6: View parking lot statistics

To view the parking lot statistics, a message specifying the data type and format is first sent to an Admin instance. Admin then creates the appropriate ReportData and Report instances based on the data type and format, respectively. In this diagram, RevenueData and ExcelReport have been chosen by the caller. ExcelReport receives the RevenueData and generates the output when the `getReport()` method is called. At this level, we assume that this method prints the formatted output to a file, in this case an Excel file.

# Appendix

Attached on the remainder of the pages is the Requirement Specifications that this document was based on.

<b>1 - Introduction.....</b>	<b>3</b>
<b>2 - Project Background.....</b>	<b>4</b>
Overview.....	4
Pain points.....	4
Goals.....	4
Assumptions.....	5
Scope.....	5
<b>3 - Problem Domain.....</b>	<b>6</b>
Domain vocabulary.....	6
Actors.....	6
Domain model and entity descriptions.....	6
Tasks.....	7
<b>4 - Functional Requirements.....</b>	<b>8</b>
Work area 1 - Parking service.....	8
Task 1.1 - Create driver account.....	9
Task 1.2 - Update driver account.....	10
Task 1.3 - Pre-book a parking slot.....	11
Task 1.4 - Change a booked slot.....	12
Task 1.5 - Park vehicle.....	13
Task 1.6 - Pay for booking.....	14
Task 1.7 - Pay for walk-in.....	15
Work area 2 - Parking lot administration.....	16
Task 2.1 - View booking statistics of the parking lot.....	17
Task 2.2 - Change existing parking slot.....	18
Task 2.3 - Add a new parking slot.....	19
<b>5 - Workflows.....</b>	<b>20</b>
Task 1.1 - Create driver account.....	20
Task 1.2 - Update driver account.....	21
Task 1.3 - Pre-book a parking slot.....	22
Task 1.4 - Change a booked slot.....	23
Task 1.5 - Park vehicle.....	24
Task 1.6 - Pay for booking.....	25
Task 1.7 - Pay for walk-in.....	26
Task 2.1 - View booking statistics of the parking lot.....	27
Task 2.2 - Change existing parking slot.....	28
Task 2.3 - Add a new parking slot.....	29
<b>6 - Quality Requirements.....</b>	<b>30</b>
Security.....	30
Performance.....	30
Scalability.....	30
Usability.....	31
Portability.....	31
<b>7 - Other Requirements.....</b>	<b>32</b>

Product-level requirements.....	32
Design-level requirements.....	32
<b>8 - Validation.....</b>	<b>33</b>
CRUD Check.....	33
<b>9 - Possible solutions.....</b>	<b>34</b>
Solution 1 - Mobile app for OSPS.....	34
Solution 2 - IoT-Enabled OSPS.....	34



# 1 - Introduction

This specification document details the requirements for a Smart Parking information system to be developed at Duy Tan Street. This system will help drivers to book parking slots online conveniently and the owners to better monitor and manage the parking facilities.

This document is structured as follows: **Section 2** discusses the business, the issues with its current operations, and the goals, assumptions, and scope of the new online system. **Section 3** describes the problem domain and domain model. **Section 4** details the functional requirements of the new system with the Task and Support method. **Section 5** illustrates the basic workflows of the system. **Section 6** lists five core quality requirements of the system. **Section 7** discusses additional requirements such as design or product requirements. **Section 8** shows our validation of the requirements using the CRUD check. Finally, **section 9** suggests two possible solutions based on the specified requirements.

## 2 - Project Background

### Overview

<b>Customer</b>	SmartParking company
<b>Background</b>	SmartParking offers paid parking spaces at Duy Tan Street for cars and motorbikes. Under their current system, drivers must physically locate and pay for their parking spots on arrival, resulting in long wait times and the possibility of being turned away. The company also needs an easy way to upgrade and monitor the operation of the parking lot.
<b>Purpose</b>	This project aims to develop an Online Smart Parking Space (OSPS) system for SmartParking. The system will let drivers pre-book slots, manage bookings, handle payments, and receive parking guidance. It will also allow the company owners to better monitor and manage the parking lot.
<b>Project Type</b>	Tender

### Pain points

The following are the “pain points” of SmartParking’s current system, grouped into two areas:

- Parking services:
  - Drivers have to locate and pay for their parking spots on arrival.
  - Drivers face long waiting times at peak hours and cannot park on time.
  - Some drivers are turned away after all spots are occupied.
  - The parking services are only available offline.
- Administration:
  - The owners currently cannot monitor the statistics of the parking slots.

### Goals

Based on the pain points above, the project sets the following goals for the new Online Smart Parking Space (OSPS) system:

- Allow drivers to pre-book parking slots for a faster and more convenient parking experience.
- Provide a seamless parking experience when drivers arrive at the parking lot.
- Provide drivers with a fast and secure payment flow that supports invoice and receipt generation.
- Help SmartParking establish an online presence and broaden its customer base.
- Allow SmartParking to oversee its operations and upgrade its parking lot as new facilities are available.

## Assumptions

The following assumptions are made about SmartParking the Online Smart Parking Space (OSPS):

- Currently, SmartParking only has one location in Duy Tan Street and the parking lot has up to 500 slots.
- The number of drivers entering/leaving the parking lot can peak at 300 twice a day, once in the morning from 7:30 AM to 8:30 AM and once in the afternoon from 5:00 PM to 6:00 PM. Outside these peak periods, the average number of drivers per hour is 3/hr.
- Drivers don't need to use the online system to be able to park at SmartParking. They can still arrive at the parking lot and park like before.
- Slot bookings must be pre-paid online. The only allowed payment method for bookings is credit.
- If the driver does not book in advance, they can pay for their parking slot on departure. In this case, they can pay by either cash or credit.
- SmartParking charges for the parking slot based on the type of vehicle and the duration of the parking session.

## Scope

The SmartParking company wants the new system to shorten the queuing time for the driver as well as inform all the users about the availability of the parking lots. After creating a new account in the system, the driver can manage their parking slots, pay for their services, and get the invoice and receipt back. The system will also be able to update its information using real-time data and display it over various periods. Besides that, the system also needs to have spare room for future system expansion as they intend to update the current facility in the future.

## 3 - Problem Domain

### Domain vocabulary

This section defines some important terms that will be used throughout this document:

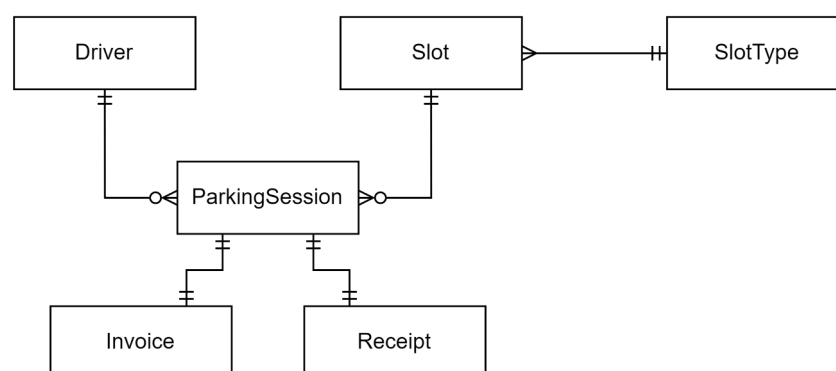
- OSPS: Online Smart Parking Space. This is the system that the SmartParking company wants to develop.
- Customer: Refers to the driver who uses the parking services offered by SmartParking. It is used interchangeably with “driver”; however, “driver” will be preferred whenever possible for clarity.
- Vehicle: The physical vehicle that the customer drives to and parks at SmartParking. Currently, two types of vehicles are supported: cars and motorbikes. In the future, the company may add more types.
- Parking lot: The area owned by SmartParking where drivers can park their vehicles.
- Parking space/parking slot: A portion of physical space in the parking lot where one vehicle can be parked (a car or motorcycle).

### Actors

The following are the “actors” who will be interacting with the system:

- Drivers: These are the customers of SmartParking who will be using the parking services.
- Owners: The owners of SmartParking would like to use the system to monitor their parking facilities.
- Guards: Guards are responsible for checking who enters and leaves the parking lot and handling payment.

### Domain model and entity descriptions



- Driver: This entity represents both the driver and their account on the system.
- Slot: This entity represents a parking slot. A slot is tied to one slot type.
- SlotType: This entity represents the type of a parking slot. Currently, two types are supported corresponding to the two supported vehicles: cars and motorbikes.
- ParkingSession: This entity represents a parking session made by the driver. This parking session can be pre-booked or started on arrival.

- Invoice: This entity represents an invoice tied to a particular booked parking session. Invoices are not generated for walk-in parking sessions as it is unnecessary.
- Receipt: This entity represents a receipt tied to a particular parking session. It is proof that the driver has paid for their parking time at SmartParking.

## Tasks

These are the tasks that the OSPS must support, divided into two work areas:

1. Parking service
  - 1.1. Create driver account
  - 1.2. Update driver account
  - 1.3. Pre-book a parking slot
  - 1.4. Change a booked slot
  - 1.5. Park vehicle
  - 1.6. Pay for booking
  - 1.7. Pay for walk-in
2. Parking lot administration
  - 2.1. View booking statistics of the parking lot
  - 2.2. Change existing parking slot
  - 2.3. Add a new parking slot

## 4 - Functional Requirements

### Work area 1 - Parking service

<b>General</b>	<p>This area covers the core business activities of SmartParking. SmartParking drivers can either pre-book a slot or get allocated one on arrival. Choosing the former requires the driver to have an account on the system but lets them reserve a spot and conveniently pay online. Choosing the latter may result in the driver being turned away when the lot is out of space. Activities in this work area can take place at any time of day, although the bulk happens during the morning and afternoon rush hours and in the evening.</p>
<b>Actors involved</b>	<ol style="list-style-type: none"><li>1. <b>Drivers</b> that want to use the parking services. They may have diverse IT backgrounds, but all are assumed to be average mobile/computer users.</li><li>2. <b>Guards</b> are involved in some of the tasks. They have very low IT skills.</li></ol>

## Task 1.1 - Create driver account

<b>Purpose</b>	Grant the driver access to the system, a prerequisite for activities such as booking slots and updating bookings.	
<b>Trigger</b>	A new driver would like to use the system and access it for the first time.	
<b>Frequency</b>	<ul style="list-style-type: none"> <li>• 30 registrations/day during the first week of deployment.</li> <li>• Average 5 registrations/day afterward.</li> </ul>	
<b>Critical</b>	A large influx of new users (approximately 200) at the system launch.	
<b>Sub-tasks</b>		<b>Example solution</b>
1 - Provide the driver's signup credentials.		The system provides a user interface through which the driver can type in their credentials such as email, phone number, and password.
2 - Verify signup credentials.		After the driver submits their credentials, the system checks their validity. The system could compare them against an existing database to ensure the email or phone number has not been used, for example.
3 - Provide the driver's personal information.		Similar to the example solution of subtask 1.
4 - Verify personal information.		Similar to the example solution of subtask 2.
5 - Record the new account information.		The system saves the newly created account into the database.
<b>Variants</b>		<b>Example solution</b>
2a - Signup credentials are invalid.		The system displays an error message and prompts the driver to enter new valid credentials.
4a - The driver's personal information is invalid.		Similar to the example solution of variant 2a.

## Task 1.2 - Update driver account

<b>Purpose</b>	Change the credentials and/or personal information of the driver's account.	
<b>Trigger/precondition</b>	A driver wants to update his account information. A driver has already logged into the system.	
<b>Frequency</b>	10 updates/month.	
<b>Critical</b>	None	
<b>Sub-tasks</b>		<b>Example solution</b>
1 - Provide new credentials (optional).		The system provides a user interface through which the driver can type in their new credentials such as password.
2 - Verify credentials (optional, only if 1 was performed).		After the driver submits their new credentials, the system checks their validity. The system could compare them against an existing database to ensure the email or phone number has not been used, for example.
3 - Provide new personal information (optional).		Similar to the example solution of subtask 1.
4 - Verify personal information (optional, only if 3 was performed).		Similar to the example solution of subtask 2.
5 - Record the change.		The system saves the new changes into the database.
<b>Variants</b>		<b>Example solution</b>
2a - Signup credentials are invalid.		The system displays an error message and prompts the driver to enter new valid credentials.
4a - The driver's personal information is invalid.		Similar to the example solution of variant 2a.



### Task 1.3 - Pre-book a parking slot

<b>Purpose</b>	Let the driver book a parking slot in advance to ensure it is available on arrival.
<b>Trigger/precondition</b>	A driver wants to book a parking slot. The driver has already logged into the system.
<b>Frequency</b>	200 bookings/day.
<b>Critical</b>	Drivers are most likely to book slots the evening before arriving at the parking lot, resulting in more booking traffic at this time (about 150 requests).
<b>Sub-tasks</b>	<b>Example solution</b>
1 - Select a parking slot.	The system displays a map of the parking lot showing selectable slots. The driver then clicks on a slot to select it.
2 - Provide the arrival and departure times.	The system provides an interface through which the driver can specify their parking time.
3 - Pay for the booking (see subtask 1.6).	After confirming the slot location and parking time, the user proceeds to the payment screen. See subtask 1.6 for an example solution.
4 - Record the booking.	The system adds the booking information to the database and marks the slot as booked for the specified duration.
<b>Variants</b>	<b>Example solution</b>
2a - The slot is already occupied at the specified time.	The system displays an error message and prompts the driver to select a different time.

## Task 1.4 - Change a booked slot

<b>Purpose</b>	Update the time and location of the booked slot or cancel it.	
<b>Trigger</b>	The driver has already successfully booked a slot and wants to update it.	
<b>Frequency</b>	2 updates/day.	
<b>Critical</b>	None.	
<b>Sub-tasks</b>	<b>Example solution</b>	
1 - Select the booked slot.	The system displays a list of bookings made by the driver. The driver then selects the booking that they want to change.	
2 - Update slot time/location (optional).	The system provides an interface through which the driver can specify their new parking time and/or location.	
3 - Cancel the booking (optional).	The driver clicks on the "Cancel" button on the interface.	
4 - Record the update.	The system updates or deletes the booking entry in the database.	
<b>Variants</b>	<b>Example solution</b>	
2a - The desired time or location is not available.	The system displays an error message and prompts the driver to select a different slot or time.	

## Task 1.5 - Park vehicle

<b>Purpose</b>	Let the driver park their vehicle.
<b>Trigger</b>	A driver arrives at the parking lot.
<b>Frequency</b>	400 times/day.
<b>Critical</b>	The number of drivers arriving at the parking lot can peak at 300 between 7:30 AM - 8:30 AM.
<b>Sub-tasks</b>	<b>Example solution</b>
1 - Allocate an unoccupied slot to the driver.	The guard checks the system, which displays a map of parking slots and their statuses (empty, occupied, booked), and informs the driver of an empty slot.
2 - Guide the driver to the slot.	The driver clicks on the slot on their mobile phone, and the system will give them directions.
3 - Record the slot as occupied.	A sensor at the slot registers when the driver arrives and informs the system to mark the slot as occupied.
<b>Variants</b>	<b>Example solution</b>
1a - No unoccupied slots are available.	The guard will have to turn the driver away in this case.
1b - The driver has pre-booked a slot.	Slot allocation on arrival is no longer necessary, and the system will guide the driver to their booked slot on their mobile phone.
2a - The driver does not park at their allocated slot.	When a slot is allocated or booked, the system will record the driver's license plate. If the system finds it at a different slot, it will notify the guard, who will then come and guide the driver to the correct slot.

## Task 1.6 - Pay for booking

<b>Purpose</b>	Charge the driver for their booking.	
<b>Trigger</b>	The driver has selected their location and time slot and needs to pay for the booking.	
<b>Frequency</b>	200 times/day.	
<b>Critical</b>	Drivers are most likely to book slots the evening before arriving at the parking lot, resulting in more booking traffic at this time (about 150 requests).	
<b>Sub-tasks</b>		<b>Example solution</b>
1 - Generate invoice.		The system calculates the parking fee based on the vehicle type and duration and then sends the invoice to the driver's phone.
2 - Handle payment.		The system provides an interface through which the driver can pay for the booking with their bank account.
3 - Generate receipt.		The system prints or sends a receipt to the driver's phone as a confirmation of payment.
4 - Record payment.		The system records the receipt in the database as proof of payment.
<b>Variants</b>		<b>Example solution</b>
2a - The payment method is invalid.		The system shows an error message and prompts them to enter a different bank account.
2b - The driver has insufficient funds.		Similar to variant 2a.

## Task 1.7 - Pay for walk-in

<b>Purpose</b>	Charge the driver for their parking slot.	
<b>Trigger/precondition</b>	<ul style="list-style-type: none"> <li>• A driver leaves the parking slot.</li> <li>• This driver <b>had not</b> booked their slot.</li> </ul>	
<b>Frequency</b>	400 times/day.	
<b>Critical</b>	The number of drivers leaving the parking lot can peak at 300 between 5:00 PM - 6:00 PM.	
<b>Sub-tasks</b>		<b>Example solution</b>
1 - Determine the slot to be paid for and calculate the fee.		The system shows the guard the parking slot and duration corresponding to the driver's license plate. It then calculates the fee that has to be paid based on the vehicle type and parking duration.
2 - Handle payment.		The guard informs the driver of the amount and asks them to pay either in cash or by credit.
3 - Generate receipt.		The system prints a receipt as a confirmation of payment.
<b>Variants</b>		<b>Example solution</b>
2a - The payment method is invalid.		The guard will ask the driver to choose another payment method and prevent the driver from leaving until they have paid successfully.
2b - The driver has insufficient funds.		Similar to variant 2a.

## Work area 2 - Parking lot administration

<b>General</b>	This work area covers administrative tasks such as monitoring parking statistics and making changes to the parking lot. It is often done periodically, once or twice a month depending on the needs of the business.
<b>Actors involved</b>	The business owners, who have moderate to high IT skills.

## Task 2.1 - View booking statistics of the parking lot

<b>Purpose</b>	Monitor how frequently different parts of the parking lot are being booked to optimize resources and make potential upgrades.		
<b>Trigger</b>	It is the end of the month and the owners want to check the booking statistics to review their performance.		
<b>Frequency</b>	Once a month.		
<b>Critical</b>	None.		
<b>Sub-tasks</b>		<b>Example solution</b>	
1 - Select the desired time period (i.e. day, week, or month).		The system provides an interface where the owner can specify the time range they want to monitor.	
2 - Retrieve the statistics within the specified time.		The system queries data for the specified period from the database.	
3 - View the statistics.		The system formats the queried data for easier reading. For instance, it can present the data as a heat map of the parking lot to show the most frequently booked areas. When the owner clicks on a particular slot, detailed statistics for that slot will be shown.	
<b>Variants</b>		<b>Example solution</b>	
None.			

## Task 2.2 - Change existing parking slot

<b>Purpose</b>	Let the owners modify an existing parking slot to accommodate new facility changes.	
<b>Trigger/precondition</b>	<ul style="list-style-type: none"> <li>• The owner wants to change an existing parking slot.</li> <li>• The chosen slot must not be occupied or booked.</li> </ul>	
<b>Frequency</b>	5 times/year	
<b>Critical</b>	None.	
<b>Sub-tasks</b>	<b>Example solution</b>	
1 - Select the slot to change.	The system displays a map of the parking lot and lets the owner click on a slot to select it.	
2 - Update the slot's information (optional).	The owner types in the slot's new information and click submit.	
3 - Remove the parking slot (optional).	The owner clicks the "Remove" button to delete the slot from the parking lot.	
4 - Record the changes.	The system updates or deletes the slot entry in the database.	
<b>Variants</b>	<b>Example solution</b>	
1a - The slot to change is currently occupied or booked.	The system displays an error message notifying the owner that this slot is currently in use.	

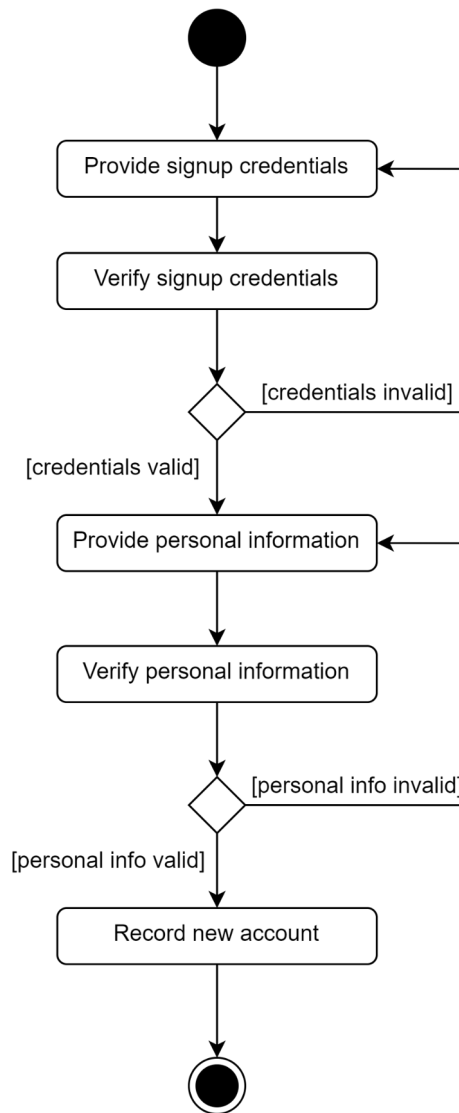


### Task 2.3 - Add a new parking slot

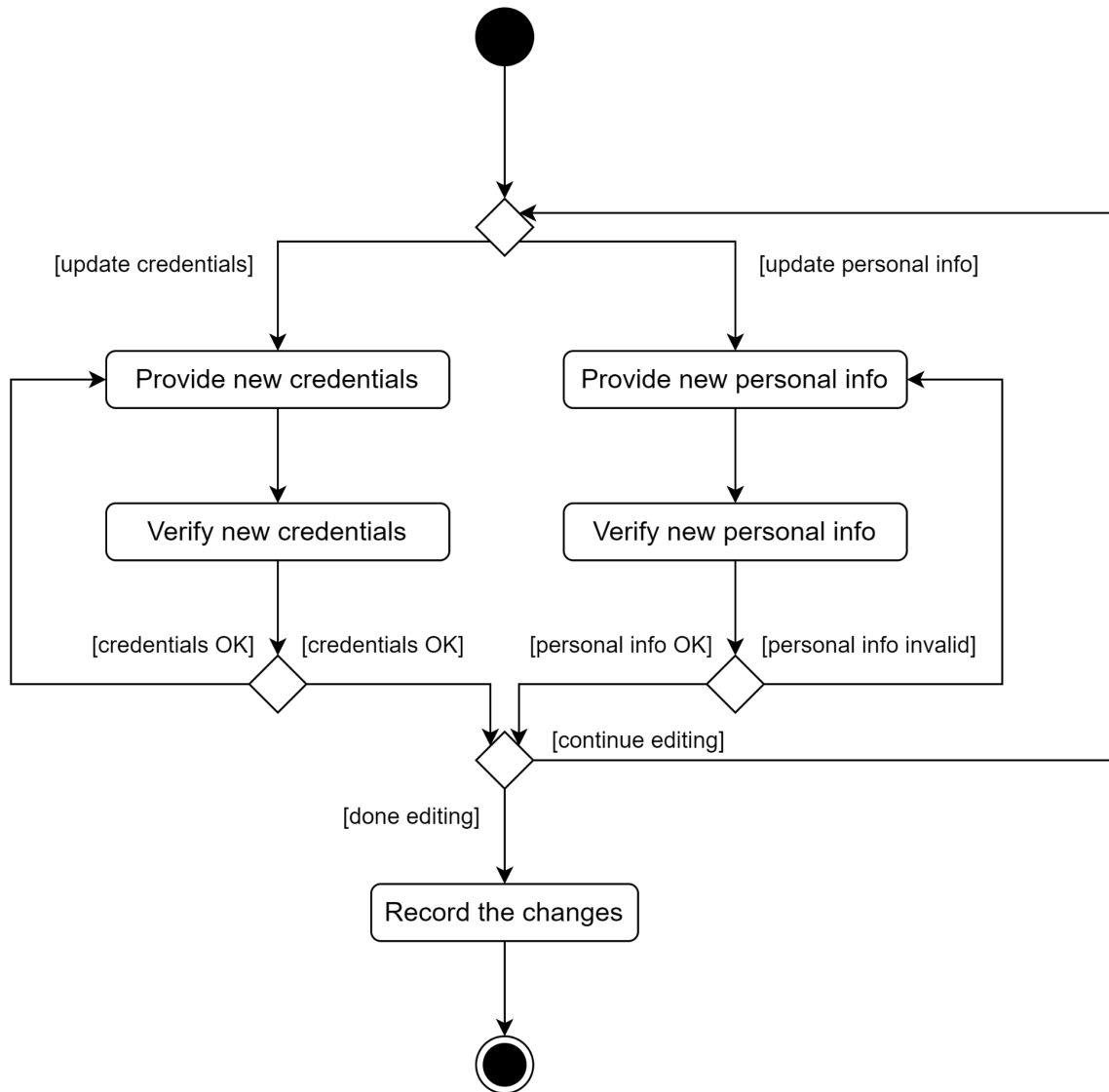
<b>Purpose</b>	Expand the parking lot by adding a new parking slot.	
<b>Trigger</b>	SmartParking's facilities have expanded and there is room for a new parking slot.	
<b>Frequency</b>	15 times/year	
<b>Critical</b>	None	
<b>Sub-tasks</b>		<b>Example solution</b>
1 - Allocate physical space for the new slot.		The owners manually find a space for the new slot.
2 - Specify the slot information.		The system provides an interface where the owner can specify the new slot's information such as vehicle type supported and location on the map.
3 - Record the new slot.		The system records the new slot in the database.
<b>Variants</b>		<b>Example solution</b>
None.		

## 5 - Workflows

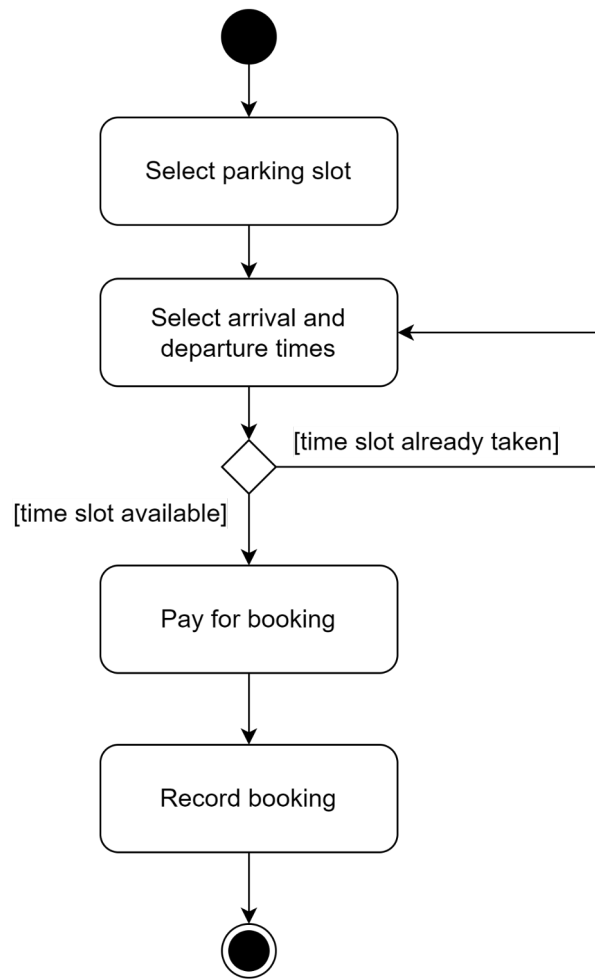
### Task 1.1 - Create driver account



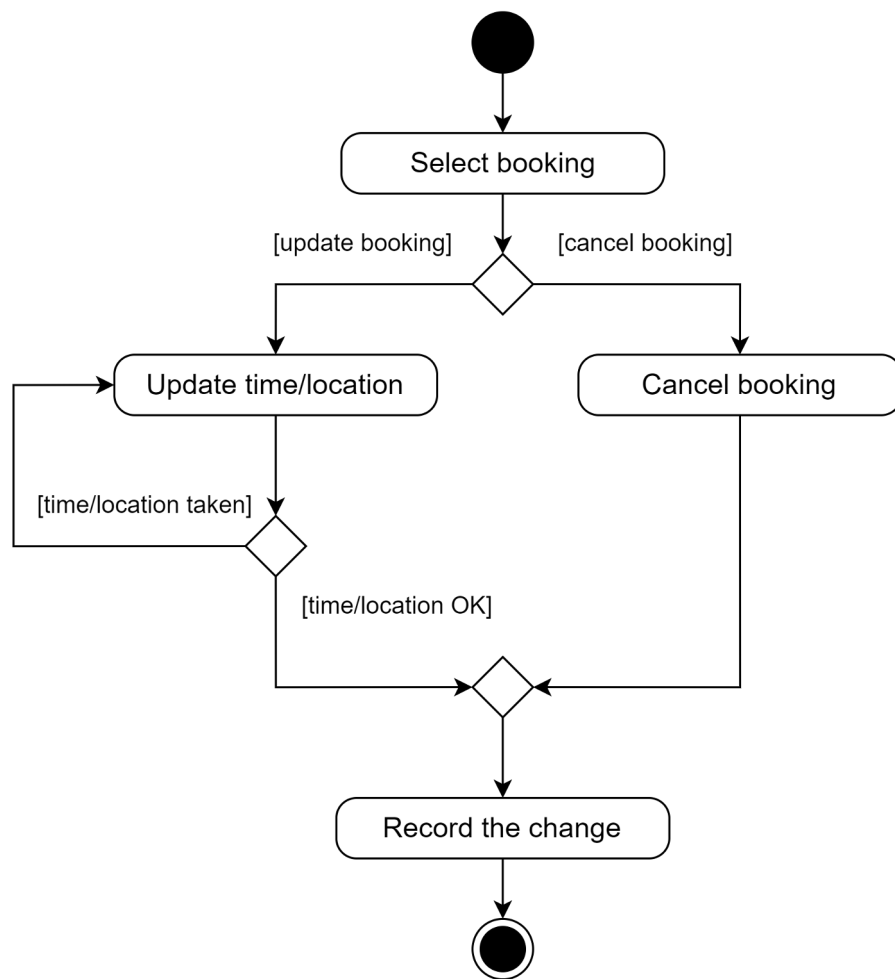
## Task 1.2 - Update driver account



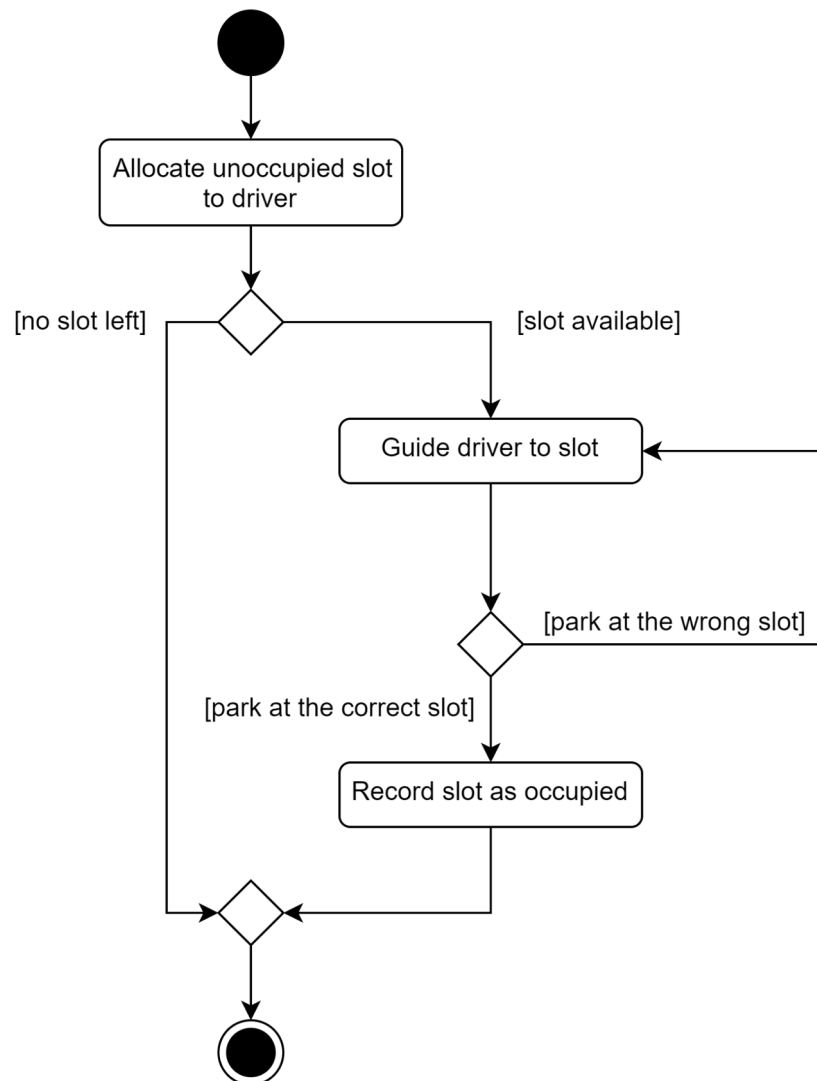
### Task 1.3 - Pre-book a parking slot



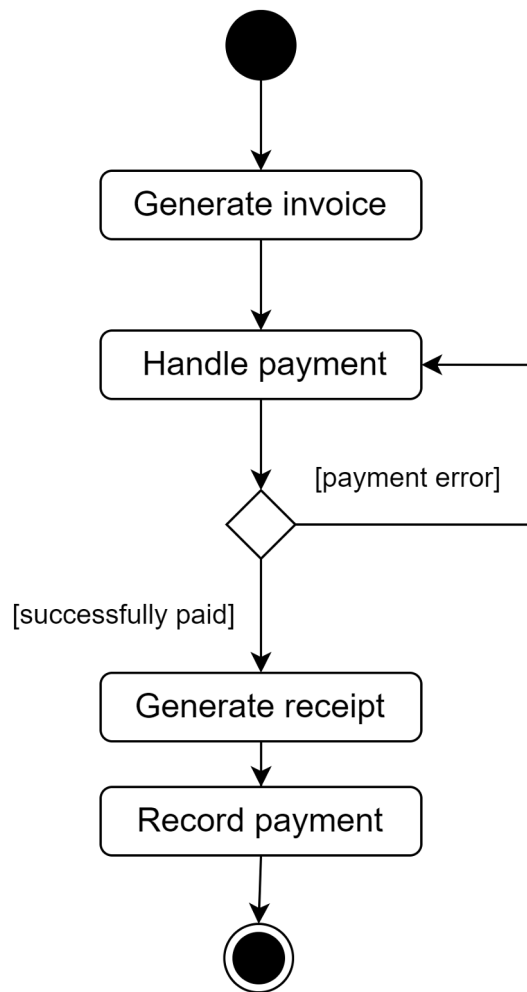
## Task 1.4 - Change a booked slot



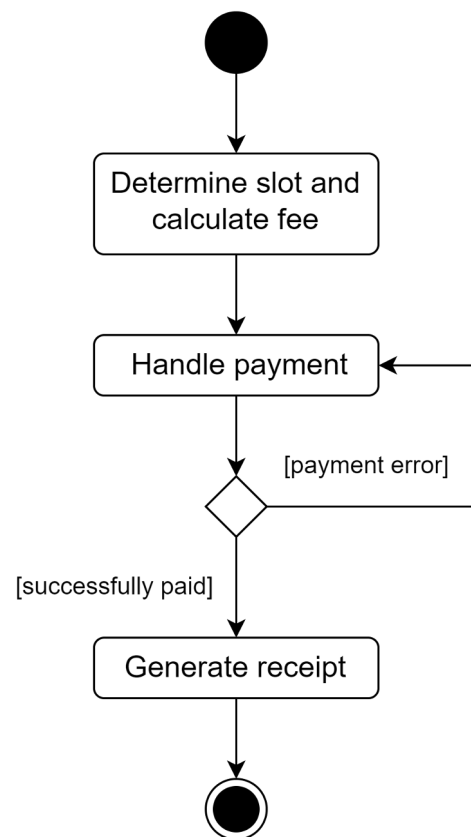
## Task 1.5 - Park vehicle



## Task 1.6 - Pay for booking

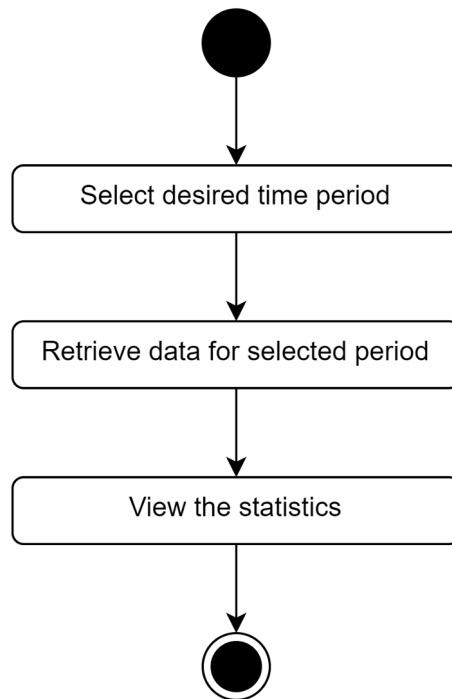


## Task 1.7 - Pay for walk-in

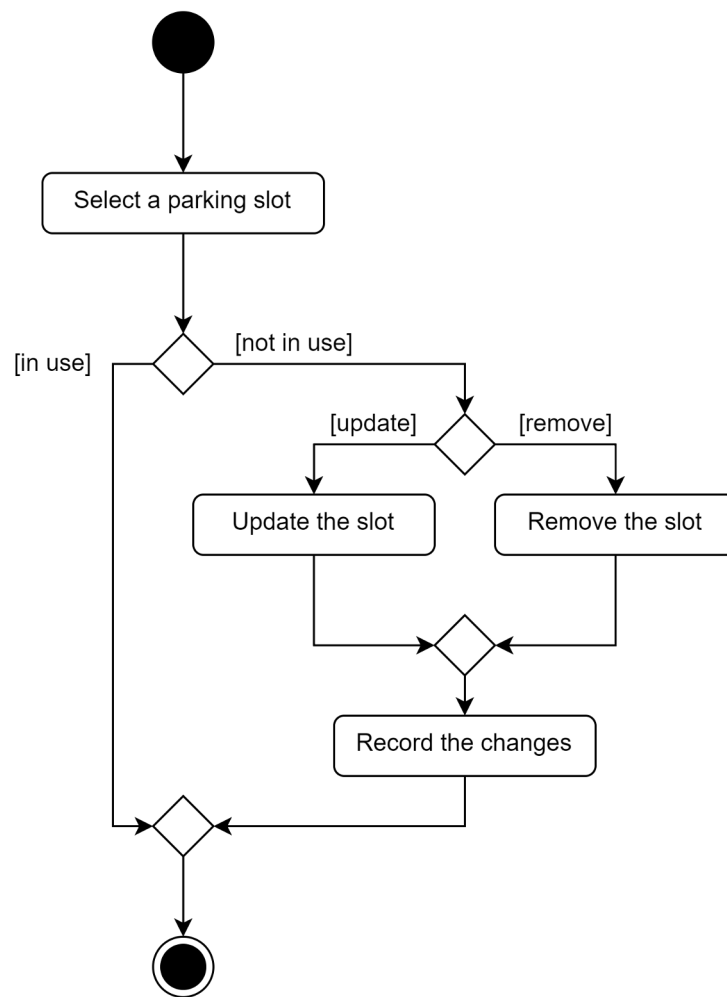




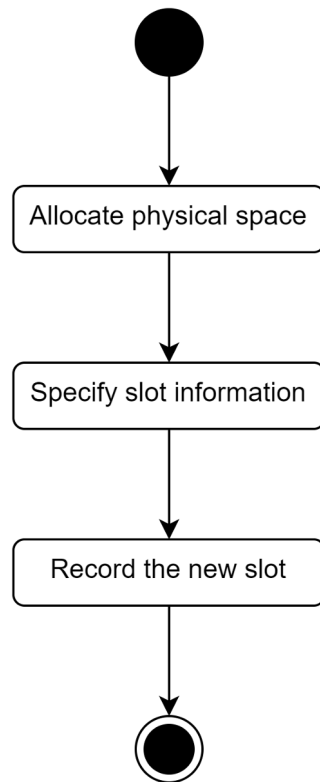
## Task 2.1 - View booking statistics of the parking lot



## Task 2.2 - Change existing parking slot



## Task 2.3 - Add a new parking slot



## 6 - Quality Requirements

### Security

Security is one of the most important qualities of the system as it needs to store drivers' personal information and handle payments. Therefore, it must be secured against unauthorized access as well as prevent users' data from being exposed. To achieve these requirements, the system must:

- Be configured securely by removing any unnecessary services and setting up appropriate users' permission.
- Perform regular backups and updates to keep the system free from vulnerabilities and data loss.
- Implement a secure payment gateway to perform transactions.

### Performance

Performance is an important factor in smart parking software since it directly affects user experience and operational efficiency. High-performance software detects available parking spaces quickly and accurately, minimizing the amount of time vehicles spend looking for them. This reduces congestion, lowers pollutants, and increases customer pleasure. Furthermore, dependable and quick performance enables seamless interaction with other systems, such as payment processing and navigation, resulting in a more enjoyable overall parking experience, the system must meet the following requirements:

- The app should respond to user inputs and questions in 1-2 seconds.
- The response time for loading available parking spaces should be less than three seconds.
- The parking guidance should be at least 95% accurate.
- Should be stable when receiving around 1000 queries at the same time.
- Login time should be less than 3 to 4 seconds.

### Scalability

Scalability is a crucial quality attribute for the SmartParking Online Smart Parking Space (OSPS) system, ensuring that the system can handle increased load and expanded functionality over time without compromising performance or user experience. To ensure scalability, the system must meet the following requirements:

- Handle the creation of 30 new user accounts per day initially, reducing to 5 per day after the first week. Manage a large influx of new users, approximately 200, at system launch. Accommodate an annual growth rate of 15%, starting from 300 accounts in the first year, over the next 5 years.
- Support peak traffic of 300 drivers between 7:30 AM - 8:30 AM, with additional peaks of 200 requests between 5:00 PM - 6:00 PM, and 150 requests between 8:00 PM - 10:00 PM. Maintain a smooth user experience during average traffic loads throughout the day.

- Efficiently manage growing data volumes related to user accounts, bookings, parking slots, and transactions. Must support up to approximately 2GB of data each year. This amount is not expected to grow too significantly every year.
- Process 200 payment transactions per day, with scalability to accommodate increased volumes. This amount will grow by 15% each year.

## Usability

Usability is an important quality of the system as it will be used by drivers, guards, and owners with varying levels of IT expertise. A system that is difficult to use will slow down operations and deter drivers from using the services, damaging SmartParking's revenue. To ensure usability, the system must meet the following requirements:

- Drivers shall successfully perform the tasks of booking a slot and updating the booking in at most 5 minutes.
- Guards shall be able to use the system after 12 hours of training.
- Owners shall be able to use the system after 12 hours of training.
- The system must show users error messages and provide clear guidance when an error happens.
- The system must follow the guidelines established in the Swinsoft Consulting UI/UX document.

## Portability

Portability is critical in smart parking software because it guarantees that the program runs smoothly across several devices and operating systems. This broad compatibility improves user convenience by providing access to parking information from any device. It also facilitates maintenance and upgrades, resulting in a consistent and dependable user experience, which is critical for wider adoption. the system must meet the following requirements:

- Available on both Android and IOS platforms.
- Available as a web application for computer users.

## 7 - Other Requirements

### Product-level requirements

- Create Driver Account:
  - The system should provide an interface for drivers to create a new account. This should include fields for the driver's name, contact information, and vehicle details.
- Update Driver Account
  - The system should allow drivers to update their account information. This includes changing their contact information, password, and vehicle details.
- Pre-book a Parking Slot
  - The system should allow drivers to pre-book a parking slot. The system should display available slots for the driver to choose from.
- Change a Booked Slot
  - The system should allow drivers to change their booked slot. This includes canceling a booking and making a new booking.
- Park Vehicle
  - The system should allow drivers to check in their vehicles once they arrive at the parking lot. The system should confirm the booking and guide the driver to the booked slot.
- Pay for Walk-in
  - The system should allow walk-in drivers to pay for their parking. The system should calculate the parking fee based on the duration of parking.
- View Booking Statistics of the Parking Lot
  - The system should provide an interface for administrators to view booking statistics. This includes the number of bookings, revenue generated, and occupancy rate.
- Change Existing Parking Slot
  - The system should allow administrators to change the details of an existing parking slot. This includes changing the slot size, location, and availability.
- Add a New Parking Slot
  - The system should allow administrators to add a new parking slot. The system should prompt the administrator to enter the slot details such as size, location, and availability.

### Design-level requirements

Besides the product-level requirements that have been mentioned in the previous sections, the system must also follow these design-level requirements:

- The UI/UX of the system must be accessible to users by using appropriate font style and color contrast.
- The system should be responsive to ensure the user interface works with a wide range of devices and screen sizes.
- Visualize the statistics of the booked slots nicely to the owners of the system for the purpose of monitoring the parking lot.
- The system may need a notification system to remind the users about their services.

## 8 - Validation

### CRUD Check

Entities Tasks	Driver	Slot	SlotType	ParkingSession	Invoice	Receipt
1.1 - Create driver account	C					
1.2 - Update driver account	R, U					
1.3 - Pre-book a parking slot		R	R	C		
1.4 - Change a booked slot		R	R	R, U, D		
1.5 - Park vehicle		R	R	C, U		
1.6 - Pay for booking		R	R	R	C, R, D	C, R
1.7 - Pay for walk-in		R	R	R		C, R
2.1 - View booking statistics of the parking lot		R	R	R		
2.2 - Change existing parking slot		R, U, D	C, U, D			
2.3 - Add a new parking slot		C	C, U, D			
<b>Missing</b>	<b>D</b>				<b>U</b>	<b>U, D</b>

Explanation of missing operations:

- **Delete** for **Driver**: We assume that SmartParking does not allow the deletion of Driver accounts.
- **Update** for **Invoice** and **Receipt**: It does not make sense to update an invoice or receipt.
- **Delete** for **Receipt**: We assume that SmartParking may want to calculate the historical profits generated from their parking lot, requiring the retention of receipts for long periods. Also, given their small size, it is acceptable to store all old receipts.

## 9 - Possible solutions

### Solution 1 - Mobile app for OSPS

A Smart Parking mobile app is intended to streamline the parking experience by providing extensive and user-friendly features. It has strong user account management, allowing users to register with their email, password, and personal information and manage their profiles securely.

The software displays real-time parking slot availability, allowing users to choose slots, specify arrival and departure times, and receive booking confirmations and reminders. Payment processing is safe, and it accepts a variety of payment methods such as credit cards and PayPal, with automatic invoice generation. Administrators benefit from a comprehensive administration interface that allows them to control slots and monitor real-time activity. Reporting tools offer insights into usage trends, financial activities, and system performance data.

The software is designed for scalability, processing large numbers of reservations effectively and providing quick response times. Security is the top priority, with strong data protection and access control procedures in place. The software is mobile-compatible, with an easy layout for a seamless user experience. Interaction with third-party services such as Google Maps and payment gateways improves functionality, whilst API access enables more extensive system interaction. Comprehensive support and maintenance services ensure that the app stays functional and up to date. This mobile app should be available on both Android and IOS platforms for versatility.

### Solution 2 - IoT-Enabled OSPS

An IoT-enabled smart parking system leverages the power of the Internet of Things (IoT) to provide a highly automated and efficient parking management solution. This system integrates IoT sensors, mobile apps, and web interfaces to deliver real-time data and automated control over parking operations. Users can register with their email, password, and personal information, managing their profiles securely through mobile and web interfaces, with strong authentication and role-based access controls ensuring data security and user privacy.

IoT sensors installed in parking lots provide real-time data on slot availability, allowing users to book slots, specify arrival and departure times, and receive booking confirmations and reminders. The system supports both advance and immediate reservations, with real-time updates on slot status. Secure payment processing is available with various payment options like credit cards and PayPal, integrated into both the mobile app and web interface, with automatic generation of invoices and receipts for completed transactions.

Administrators have access to a comprehensive dashboard that integrates data from IoT sensors. Reporting tools provide insights into usage trends, financial activities, and system performance data, enhanced by real-time IoT data. Designed to handle large volumes of data from IoT sensors, the system ensures scalability and reliability through cloud hosting (e.g., AWS, Azure), with load balancing and auto-scaling features managing varying traffic loads effectively, maintaining quick response times and high availability.



Security is a top priority, with robust data protection measures and access controls in place, complemented by regular security audits and compliance with industry standards. Enhanced security protocols for IoT devices and data transmission protect user data and system integrity. Integration with third-party services such as Google Maps for navigation and payment gateways for secure transactions enhances functionality, while API access allows for further system interaction and integration capabilities. Comprehensive support and maintenance services ensure the system remains functional and updated, providing a seamless user experience across all devices and allowing users to access the system from smartphones, tablets, desktops, laptops, and through automated IoT interfaces.