

OOP Report

by Jeffery Aideyan-Ohonba

Submission date: 22-Apr-2025 10:35AM (UTC+0100)

Submission ID: 256220991

File name: 58851_Jeffery_Aideyan-Ohonba_OOP_Report_662765_223229607.pdf (2.52M)

Word count: 8890

Character count: 65793



CAR PARK APPLICATION REPORT



Jeffery Aideyan-Ohonba
STUDENT ID: 2420646

Table of Contents

Introduction.....	4
Best Practices in OOP Design	4
Inheritance.....	4
Best Practices	4
Implementation in the Car Park Solution following best practice	5
Interfaces.....	7
Best Practices	7
Implementation in the Car Park Solution following best practice	8
Abstract Classes	9
Best Practices	9
Implementation in the Car Park Solution following best practice	10
Polymorphism	12
Best practices	12
Implementation in the Car Park Solution following best practice	12
Generics	14
Implementation in the Car Park Solution following best practice	14
Classes Structure	16
Vehicle, Car, Truck, Motorcycle	16
IDReader, BarcodeReader, and NumberPlateReader	16
Barrier, Sensor, and FullSign.....	17
Ticket.....	18
Payable	19
InputValidator	20
CarPark.....	21
CarParkSystem.....	22
Static vs Dynamic Memory Allocation in Java.....	23
Selecting the Best Data Structure for the Car Park Application	23
Unit Test Documentation	24
CarParkTest.....	24
Test Methods and Explanations	24
Test Results Summary.....	30
BarcodeReaderTest.....	30
Test Methods and Explanations	30
Test Results Summary.....	32
CarParkSystemTest.....	32

Test Methods and Explanations	32
Test Results Summary	33
Working Program Demonstration	33
Conclusion	37
Video Demo Link	37
Bibliography.....	38
Appendix A: Source Code	39
A.1 CarParkSystem.java.....	39
A.2 CarPark.java	40
A.3 Ticket.java	44
A.4 Vehicle.java	45
A.5 Car.java	45
A.6 Motorcycle.java	46
A.7 Truck.java	46
A.8 InputValidator.java	46
A.9 IDReader.java	47
A.10 BarcodeReader.java	48
A.11 NumberPlateReader.java	48
A.12 Payable.java.....	49
A.13 Main.java	49
A.14 Barrier.java	49
A.15 FullSign.java	50
A.16 Sensor.java	51
A.17 CarParkTest.java	51
A.18 BarcodeReaderTest.java	56
A.19 CarParkSystemTest.java	57
 Figure 1: Vehicle abstract class structure	5
Figure 2: Car class structure	6
Figure 3: Truck class structure	6
Figure 4: Motorcycle class structure.....	7
Figure 5: Payable interface class structure	8
Figure 6: Car class implementing Payable interface	8
Figure 7: Truck class implementing Payable interface	9
Figure 8: IDReader abstract class	10

Figure 9: BarcodeReader class extending IDReader	11
Figure 10: NumberplateReader class extending IDReader	11
Figure 11: Creating new vehicle object	12
Figure 12: CreateVehicle() method	13
Figure 13: Dynamically calling displayInfo() method	13
Figure 14: Generic Ticket class structure	15
Figure 15: Generic issueTicket() method	15
Figure 16: Sensor and FullSign at the car park entrance.....	17
Figure 17: Barrier at the car park entrance	18
Figure 18: Payable interface calculateFee() method	19
Figure 19: parkVehicle_NewVehicle_VehicleShouldBeAdded().....	25
Figure 20: unparkVehicle_VehicleExists_VehicleShouldBeRemoved().....	25
Figure 21: unparkVehicle_VehicleDoesNotExist_ShouldShowError()	26
Figure 22: unparkVehicle_PayableVehicle_ShouldCalculateFee()	26
Figure 23: createVehicle_WhenUserSelectsCar_ShouldReturnCar()	27
Figure 24: createVehicle_WhenUserSelectsMotorcycle_ShouldReturnMotorcycle()	27
Figure 25: createVehicle_WhenUserSelectsTruck_ShouldReturnTruck()	27
Figure 26: createVehicle_WhenUserSelectsInvalidOption_ShouldReturnCar()	28
Figure 27: issueTicket_WhenCalledWithValidVehicleId_ShouldReturnTicket().....	28
Figure 28: issueTicket_WhenCalled_ShouldPrintCorrectTicketDetails()	29
Figure 29: removeTicket_TicketExists_ShouldRemoveTicketAndPrintMessage()	29
Figure 30: removeTicket_TicketDoesNotExist_ShouldPrintErrorMessage()	30
Figure 31: Unit test result for CarPark class	30
Figure 32: generateBarcode_ShouldReturnNumberBetween1000And9999()	30
Figure 33: readId_WhenGivenValidInput_ShouldReturnParsedBarcode()	31
Figure 34: displayEntranceOrExitMessage_WhenLocationIsEntrance_ShouldPrintEntranceMessage() ..	31
Figure 35: displayEntranceOrExitMessage_WhenLocationIsExit_ShouldPrintExitMessage().....	31
Figure 36: Unit test result for BarcodeReader class unit test	32
Figure 37: chooseCarPark_ValidInputType_ShouldReturnSelectedCarPark().....	32
Figure 38: chooseCarPark_InvalidInput_ShouldPromptAgain()	33
Figure 39: Unit test result for CarParkSystem class	33
Figure 40: Working program demo - program startup	34
Figure 41: Working program demo - choosing a car park.....	34
Figure 42: Working program demo - parking a vehicle	35
Figure 43: Working program demo - issuing a ticket.....	36
Figure 44: Working program demo - unparking a vehicle	36

Introduction

This report covers the development of a console-based Car Park Simulation System, designed to demonstrate the practical application of core Object-Oriented Programming (OOP) principles. Built in Java, the system simulates real-world parking operations such as selecting a car park, parking and unparking vehicles, generating tickets, and calculating fees. Throughout the project,  sed on applying key OOP concepts like encapsulation, inheritance, polymorphism, and abstraction to create a modular and maintainable design, while following best practices. This report outlines the system's structure, explains how these concepts were applied, and includes tests and screenshots to illustrate key functionalities in action.

Best Practices in OOP Design

This section covers the analyses and recommendations of the best practices for implementing the key OOP concepts used in building the Car Park solution.

Inheritance

Best Practices

Inheritance is an essential part of object-oriented programming (OOP), so using inheritance the right way is important because it can either improve or complicate a program. Instead of always using inheritance, developers should think carefully about when and how to use it to avoid problems. As stated in [1], the following best practices help ensure a good inheritance structure:

- **Depth of inheritance matters** – Research suggests that using around three levels of inheritance makes programs easier to maintain compared to not using inheritance at all. However, if inheritance goes beyond five levels, it can make the code harder to manage than if inheritance wasn't used at all.
- **Check if inheritance is necessary** – if a superclass has only one subclass, it's worth questioning whether inheritance is really needed. In some cases, other relationships, like composition (where one class contains another instead of inheriting from it), might be a better choice.

- **Favour extension over modification** – it's better to extend a class by adding new methods and attributes rather than modifying existing ones. This approach keeps the original (parent) class stable and reusable while reducing the chances of breaking the program.
- **Reduce unnecessary method overriding** – Subclasses should make full use of the attributes and methods they inherit. Instead of completely replacing inherited methods, they should try to reuse them and only override them when absolutely necessary.

Implementation in the Car Park Solution following best practice

After exploring the recommended best practices for inheritance, this section will demonstrate how inheritance has been applied in the Car Park solution, adhering to these principles.

In this system, I've created an abstract *Vehicle* class that acts as the base for different types of vehicles like *Car*, *Truck*, and *Motorcycle*. These subclasses inherit shared properties (*make*, *model*, and *year*) from *Vehicle*, and each of them implements their own version of the *displayInfo()* method. This setup follows the “is-a” relationship – a *Car* is-a *Vehicle*, and so on.

```
package org.carpark.model;

public abstract class Vehicle { 16 usages  3 inheritors  ± Jayvibeee
    protected String make; 4 usages
    protected String model;
    protected int year; 4 usages

    public Vehicle(String make, String model, int year) { 3 usages  ± Jayvibeee
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public abstract String displayInfo(); 6 usages  3 implementations  ± Jayvibeee
}
```

Figure 1: Vehicle abstract class structure

Each subclass extends this base class and adds specific logic when needed, without modifying the parent class. For example:

```
package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Car extends Vehicle implements Payable { 7 usages ± Jayvibeee

    public Car(String make, String model, int year) { 5 usages ± Jayvibeee
        super(make, model, year);
    }

    @Override 6 usages ± Jayvibeee
    public String displayInfo(){
        return "Car: " + make + " " + model + " " + year;
    }

    @Override 1 usage ± Jayvibeee
    public double calculateFee(long hours){ // Method from payable interface
        return hours * 2.50; // £2.50 per hour
    }

}
```

Figure 2: Car class structure

```
package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Truck extends Vehicle implements Payable { 2 usages ± Jayvibeee

    public Truck(String make, String model, int year){super(make, model, year);}

    @Override 6 usages ± Jayvibeee
    public String displayInfo(){return "Truck: " + make + " " + model + " " + year;}

    @Override 1 usage ± Jayvibeee
    public double calculateFee(long hours){return hours * 5.00; // Trucks pay £5 per hour}

}
```

Figure 3: Truck class structure

```

package org.carpark.model;

public class Motorcycle extends Vehicle { 2 usages ± Jayvibeee
    public Motorcycle(String make, String model, int year) { 1 usage ± Jayvibeee
        super(make, model, year);
    }

    @Override 6 usages ± Jayvibeee
    public String displayInfo(){
        return "Motorcycle: " + make + " " + model + " " + year;
    }
}

```

Figure 4: Motorcycle class structure

This approach sticks to the principle of “favouring extension over modification” – instead of constantly changing the base *Vehicle* class, each subclass extends it and handles its own specific behaviour. This keeps the parent class clean, focused, and reusable.

I’ve also avoided unnecessary overriding. The only method that gets overridden here is *displayInfo()*, and only because each vehicle type needs to present itself differently. This follows the idea of “reducing unnecessary method overriding” to maintain stability in the base class.

Interfaces

Best Practices

One important best practice when working with interfaces is to follow the Interface Segregation Principle (ISP). As stated in [2], this principle is fulfilled by deliberately keeping interfaces small and focused, each one containing only methods that are absolutely necessary for its role. This kind of “atomic” design ensures that classes aren’t forced to implement methods they don’t actually need.

Following ISP leads to a few key benefits:

- Interfaces become easier to understand and implement.
- The codebase stays cleaner and easier to maintain.
- Developers can mix and match smaller interfaces to build flexible, modular systems.

In short, smaller and more specific interfaces = cleaner, more adaptable code.

Implementation in the Car Park Solution following best practice

In the Car Park system, I've followed this best practice by introducing the *Payable* interface. This interface defined a single responsibility: providing a method to calculate the parking fee based on time.

```
package org.carpark.interfaces;

public interface Payable { 7 usages 2 implementations ± Jayvibeee
    double calculateFee(long hours); 1 usage 2 implementations ± Jayvibeee
}
```

Figure 5: Payable interface class structure

Rather than stuffing unrelated methods into this interface, it's kept laser-focused on just one task - payment. This aligns directly with the Interface Segregation Principle, keeping things clean and specific.

Only the classes that actually need to calculate parking fees implement this interface, such as *Car* and *Truck*. This makes the interface reusable and ensures that no class is forced to deal with irrelevant methods.

```
package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Car extends Vehicle implements Payable { 7 usages ± Jayvibeee

    public Car(String make, String model, int year) { super(make, model, year); }

    @Override 6 usages ± Jayvibeee
    public String displayInfo() { return "Car: " + make + " " + model + " " + year; }

    @Override 1 usage ± Jayvibeee
    public double calculateFee(long hours) { // Method from payable interface
        return hours * 2.50; // £2.50 per hour
    }
}
```

Figure 6: Car class implementing Payable interface

```

package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Truck extends Vehicle implements Payable { 2 usages  ⚡ Jayvibeee

    public Truck(String make, String model, int year) { super(make, model, year); }

    @Override 6 usages  ⚡ Jayvibeee
    public String displayInfo() { return "Truck: " + make + " " + model + " " + year; }

    @Override 1 usage  ⚡ Jayvibeee
    public double calculateFee(long hours) { return hours * 5.00; // Trucks pay £5 per hour }
}

```

Figure 7: Truck class implementing Payable interface

By following this approach, the *Payable* interface remains tightly scoped and easy to maintain. If I ever need to define another payment-related class (like an *ElectricCharger*), I can just have it implement *Payable* without needing to change the interface itself.

This makes the system easier to extend and maintain, which is exactly the kind of modularity and clarity that ISP encourages.

Abstract Classes

Best Practices

When it comes to abstract classes, the main idea is to group together all the features that multiple classes share. Basically, to pull out everything they have in common and put it in one place. According to [3], abstract classes should be created by identifying these shared features across the concrete (i.e., fully implemented) classes in the application. It might take a bit of effort to find the right balance when doing this, but that effort usually pays off in the long run through easier maintenance and better code reuse.

A good rule of thumb is to move shared behaviour as high up the inheritance structure as possible, so that more subclasses can take advantage of it without duplicating code. This keeps your code DRY (don't repeat yourself) and keeps things clean and manageable [3].

Implementation in the Car Park Solution following best practice

The *IDReader* abstract class in my Car Park system is a clear example of how abstract classes can be used to define a shared structure while leaving room for specific implementations. It contains common logic (like a *Scanner* for input) and outlines two abstract methods: *readId()* and *displayEntranceOrExitMessage(String location)*. This sets a consistent contract for all ID-reading behaviours in the system:

```
package org.carpark.reader;

import java.util.Scanner;

public abstract class IDReader<T> { 9 usages 2 inheritors 1 Jayvibee
    protected Scanner scanner; 3 usages

    public IDReader() { 2 usages 1 Jayvibee
        this.scanner = new Scanner(System.in);
    }

    public abstract T readId(); 4 usages 2 implementations 1 Jayvibee

    public abstract void displayEntranceOrExitMessage(String location); 4 usages 2 implementations 1 Jayvibee
}
```

Figure 8: *IDReader* abstract class

Two different reader classes extend this base class: *BarcodeReader* and *NumberPlateReader*. Each one provides its own logic for reading IDs and displaying messages depending on whether the user is entering or exiting the car park.

This design follows the best practice of pushing shared functionality as high up the inheritance chain as possible. Rather than duplicating scanner logic or message-handling structure across multiple classes, everything is centralised in *IDReader*. This keeps things DRY and makes the codebase easier to maintain or extend, say if another type of reader needed to be added later.

And since *IDReader* can't be instantiated on its own, it's used strictly as a foundation, just as abstract classes are meant to be.

Polymorphism

Best practices

Polymorphism is one of those OOP concepts that just makes life easier. Best way to handle polymorphism? As suggested in [4], keep method names clear, avoid overriding unless it's really necessary, and don't go overboard with overloading. When it's used properly, it makes your code way easier to work with, especially when things start getting more complex.

Implementation in the Car Park Solution following best practice

Polymorphism shows up in my system when different types of vehicles: *Car*, *Truck*, and *Motorcycle*, are handled using a single reference type: *Vehicle*. This allows the system to treat all vehicles in a generalised way while still maintaining their specific behaviour where it matters.

When a vehicle arrives at the entrance, the *createVehicle()* method is called, which dynamically constructs the correct vehicle object based on user input. Even though the returned object might be a *Car*, *Truck*, or *Motorcycle*, it's stored as a *Vehicle*, like this:

```
// Create a new vehicle object
Vehicle vehicle = createVehicle();
```

Figure 11: Creating new vehicle object

```

public Vehicle createVehicle(){ 5 usages ± Jayvibee
    System.out.println("Select Vehicle Type: ");
    System.out.println("1. Car");
    System.out.println("2. Motorcycle");
    System.out.println("3. Truck");

    int choice = InputValidator.getNumberInput(prompt: "Enter your choice (1-3):");
    String make = InputValidator.getStringInput(prompt: "Enter vehicle make:");
    String model = InputValidator.getStringInput(prompt: "Enter vehicle model:");
    int year = InputValidator.getNumberInput(prompt: "Enter vehicle year:");

    switch (choice) {
        case 1:
            return new Car(make, model, year);
        case 2:
            return new Motorcycle(make, model, year);
        case 3:
            return new Truck(make, model, year);
        default:
            System.out.println("Invalid choice. Defaulting to Car.");
            return new Car(make, model, year);
    }
}

```

Figure 12: CreateVehicle() method

Later, when the vehicle is parked using *parkVehicle()*, the system calls:

```

System.out.println(vehicle.displayInfo() + " with ID " + vehicleId + " successfully parked! Remaining spaces: " + spacesLeft());

```

Figure 13: Dynamically calling displayInfo() method

Even though the vehicle reference is of type *Vehicle*, Java dynamically figures out which *displayInfo()* method to call, whether that's from *Car*, *Truck*, or *Motorcycle*. This is runtime polymorphism at work, allowing each subclass to define its own version of the method, while the rest of the system interacts with them through the base class.

This approach also lines up nicely with the recommended best practices. Instead of cluttering the code with conditionals or duplicate logic for each vehicle type, polymorphism helps keep things readable and maintainable. Plus, method overriding is used purposefully, only *displayInfo()* is overridden by each subclass, which avoids unnecessary complexity and keeps behaviour predictable and clean.

Generics

Generics are all about helping you avoid duplicate code while keeping things safe and reusable. Instead of writing separate methods or classes for different types, generics let you handle multiple scenarios with one clean, flexible piece of code. According to [5], the goal when working with generics is to “avoid duplications” and “write type-safe code which is abstract enough to fit as many situations as possible without any modifications.”

That said, while generics give you flexibility, there’s a trade-off. As mentioned in [5], “it has its own cost”, meaning you don’t want to go overboard and make your code more complicated than it needs to be. The best practice is to avoid overusing them or using them in situations that don’t really require that level of abstraction.

Implementation in the Car Park Solution following best practice

One place where generics really shine in this system is in the *Ticket<T>* class. Instead of having separate ticket classes for every possible ID type (like Integer for barcodes or String for number plates), this generic setup keeps things clean and reusable. When a new ticket is issued, the generic type *T* is automatically set based on whatever ID is passed in.

```

package org.carpark.model;

import java.time.LocalDateTime;

public class Ticket <T>{ 4 usages  ± Jayvibeee
    private T id; 3 usages
    private LocalDateTime entryTime; 3 usages
    private String carParkName; 2 usages

    public Ticket(T id, String carParkName){ 1 usage  ± Jayvibeee
        this.id = id;
        this.entryTime = LocalDateTime.now();
        this.carParkName = carParkName;
    }

    public T getId(){ return id; }

    public LocalDateTime getEntryTime(){ return entryTime; }

    public void printTicket(){ ... }
}

```

Figure 14: Generic Ticket class structure

And when issuing a ticket:

```

public <T> void issueTicket(T vehicleId){ 4 usages  ± Jayvibeee
    Ticket<T> ticket = new Ticket<>(vehicleId, this.name); // The ticket created will be the same type as the vehicle id passed in (Integer, String)
    activeTickets.put(vehicleId, ticket);
    ticket.printTicket();
}

```

Figure 15: Generic issueTicket() method

This implementation directly supports what [5] describes as the goal of generics — to “avoid duplications” and “write type-safe code which is abstract enough to fit as many situations as possible without any modifications.” By using generics this way, the code handles multiple ID formats without needing to rewrite logic or duplicate methods, keeping it simple and flexible.

At the same time, it doesn’t go overboard with abstraction, it’s used exactly where it adds value without making things harder to understand. This is important since, as [5]

also notes, generics “*have their own cost*”, so applying them deliberately like this helps strike that sweet spot between flexibility and readability.

Classes Structure

Vehicle, Car, Truck, Motorcycle

The *Vehicle* class is the base for all the vehicle types in my system. It handles the shared attributes like *make*, *model*, and *year*, and defines an abstract method *displayInfo()*, which each subclass overrides to provide specific vehicle details. This setup lets the system treat different vehicle types in a unified way while still preserving their differences, especially when displaying info or calculating parking fees.

I’ve got three subclasses: *Car*, *Truck*, and *Motorcycle*. Both *Car* and *Truck* implement the *Payable* interface, meaning they provide their own rules for calculating fees based on parking duration. *Motorcycle*, on the other hand, does not implement *Payable*, making it exempt from fee calculations in the current system.

All vehicles are created dynamically based on user input during entry, using a *createVehicle()* method that returns a *Vehicle* object. This allows the system to store and interact with any vehicle type using just the base class, leveraging polymorphism to call the correct *displayInfo()* or *calculateFee()* when needed.

IDReader, BarcodeReader, and NumberPlateReader

The *IDReader<T>* class is a generic abstract class that provides a flexible base for handling different types of vehicle IDs. It defines two key abstract methods: *readId()* (for collecting and validating ID input) and *displayEntranceOrExitMessage()* (to show different instructions depending on whether the vehicle is entering or exiting).

Two concrete subclasses extend this:

- ***BarcodeReader*** - works with Integer types and can automatically generate random barcodes for new entries.
- ***NumberPlateReader*** - works with String types and validates number plates using a regex format (e.g. AB12 XYZ).

This setup uses generics to let each reader handle its own data type safely without duplicating logic, while still fitting into the larger system through the shared *IDReader* interface.

At runtime, the system interacts with all ID readers as *IDReader* objects, but the actual behaviour is determined by the specific subclass. This flexibility is especially useful when choosing between different entry ID systems.

Barrier, Sensor, and FullSign

These classes represent physical components of the car park and are managed internally by the *CarPark* class to control access, monitor vehicles, and signal availability.

Barrier simulates the entrance and exit gates. Each instance keeps track of its position (up or down) and provides *raise()* and *lower()* methods to visually animate the barrier through terminal output. This mimics real-world gate behaviour when a vehicle is entering or exiting.

Sensor is used to detect the presence of a vehicle at either the entrance or exit. Its *detectCar()* method is triggered at the beginning of the entry/exit process, and *reset()* is used to turn off the sensor once the vehicle has moved.

FullSign is a simple indicator that lights up when the car park reaches full capacity. Through *switchOn()* and *displayFullSign()*, it alerts incoming vehicles that entry is no longer possible.

All three are tightly integrated into the *CarPark* workflow:

```
entrySensor.detectCar(location: "entrance");

// Check if there is space in the car park first before trying to let them in
if(fullSign.isLightOn()){
    fullSign.displayFullSign();
    entrySensor.reset(location: "entrance"); //turn entry sensor off
}
```

Figure 16: Sensor and FullSign at the car park entrance

Later, the barriers are automatically raised or lowered during entry/exit:

```
entryBarrier.raise(location: "Entrance");
parkedVehicles.put(vehicleId, vehicle);
decrementSpaces();
System.out.println(vehicle.displayInfo() + " with ID " + vehicleId + " successfully parked! Remaining spaces: " + spacesLeft());
entryBarrier.lower(location: "Entrance");
entrySensor.reset(location: "entrance");
```

Figure 17: Barrier at the car park entrance

They're all pretty lightweight, but by splitting them into their own classes, I was able to keep the code in the *CarPark* focused on logic, not visual stuff. And if I wanted to expand later (like adding sound effects or timing logic), I wouldn't have to touch the main logic. It keeps things modular and makes testing way easier.

Ticket

The *Ticket<T>* class handles the entry records for vehicles that come into the car park. I made it generic so it can work with different types of IDs like an Integer for barcodes or a String for number plates. That way, it's flexible enough to work with both types of ID readers used in the system without needing separate versions. This design aligns with the overall system's generic approach to ID handling, ensuring flexibility across different *IDReader* implementations.

Attributes include:

- *id*: the unique identifier of the vehicle (generic type T).
- *entryTime*: timestamp recorded at the time of ticket creation.
- *carParkName*: the name of the car park where the ticket was issued.

Key methods:

- *getId()* and *getEntryTime()* give access to ticket details.
- *printTicket()* visually simulates the printing of a physical ticket, complete with formatting and a time delay for realism.

Within the *CarPark* class, tickets are:

- **Issued** via the *issueTicket()* method after a vehicle is successfully parked.
- **Stored** in the *activeTickets* map using the same ID used to identify the vehicle.

- **Referenced** at exit to calculate the total time spent in the car park, which is especially important for vehicles that implement the *Payable* interface (like *Car* or *Truck*).
- **Removed** upon exit using the *removeTicket()* method.

By keeping *Ticket* generic and self-contained, makes it super easy to reuse or even extend in the future, like if I wanted to support QR codes or other ID types later on. The main idea was to make it flexible and focused on doing just one thing: keeping track of entry info in a clean way.

Payable

The *Payable* interface defines a contract for any vehicle that incurs a parking fee. It contains a single method:

```
public interface Payable { 7 usages 2 implementations ± Jayvibeee
    double calculateFee(long hours); 1 usage 2 implementations ± Jayvibeee
}
```

Figure 18: *Payable* interface *calculateFee()* method

This method expects the number of hours a vehicle has stayed parked and returns the total amount owed.

In the system, *Payable* is implemented by the *Car* and *Truck* class. These classes override *calculateFee()* to apply their own fee logic (e.g., £2.50/hour for cars and £5.00/hour for trucks). This encapsulation ensures each vehicle type knows how to calculate its own fee without external logic.

The *CarPark* class interacts with *Payable* during the vehicle exit process:

- When a vehicle is about to leave, the system checks whether it implements *Payable* using the *instanceof* operator.
- If true, the parking fee is calculated by retrieving the entry time from the associated *Ticket*, comparing it with the current time, and then calling *calculateFee()* on the vehicle object.

The use of the *Payable* interface demonstrates polymorphism, and it allows for new fee-paying vehicle types to be added without modifying existing *CarPark* logic. It also keeps billing logic modular, easy to test, and specific to each vehicle type.

InputValidator

The *InputValidator* class acts as a centralised utility for handling and validating all user input throughout the system. Instead of cluttering every class with repeated validation code, I kept things clean by centralising all that logic here using static utility methods.

Key Methods:

- *getYesOrNoInput(String prompt)* Prompts the user until a valid “yes” or “no” response is entered. This is used a lot in places like *CarPark.manageOperations()* when checking for vehicle presence.
- *validateNumber(String input)* Attempts to parse a string into an integer. If parsing fails, the user is repeatedly prompted until valid numeric input is received.
- *getNumberInput(String prompt)* Combines prompting and validation in one step. Ensures only integers are returned.
- *getStringInput(String prompt)* Ensures that the input string is not empty. Used when collecting vehicle information such as make and model.

Rather than duplicating validation logic across the codebase, this utility centralises input concerns. This reduces clutter and ensures all parts of the system benefit from consistent validation behaviour.

CarPark

The CarPark class serves as the central controller for the parking system, managing vehicle flow, ticket issuance, capacity tracking, and fee calculation. It ties together multiple components like barriers, sensors, readers, and tickets into a cohesive operation loop.

Key Responsibilities:

- Manages vehicle entry and exit
- Tracks parking capacity and space availability
- Issues and removes parking tickets
- Calculates parking fees (for *Payable* vehicles)
- Interfaces with hardware-like classes (sensors, barriers, signs)

Major Components:

- *entrySensor / exitSensor*: Detect whether a car is present at the gates.
- *entryBarrier / exitBarrier*: Raised/lowered during vehicle movement.
- *fullSign*: Signals if the car park is at capacity.
- *IDReader<?>*: Used to obtain a vehicle's unique ID, which may be manually entered or barcode-generated.
- *parkedVehicles*: Tracks vehicles currently in the car park.
- *activeTickets*: Associates each parked vehicle with a timestamped *Ticket*.

Key Methods:

- *manageOperations()*: The core operation cycle that handles both entry and exit events. Entry is denied if the car park is full, and duplicate entries are prevented. During exit, ticket times are used to compute a fee for vehicles that implement *Payable*.
- *createVehicle()*: Prompts the user to input the vehicle type and details, returning a specific subclass of *Vehicle*.
- *issueTicket() / removeTicket()*: Encapsulate the process of generating and managing ticket lifecycles.

CarParkSystem

The *CarParkSystem* class is basically the brain behind the whole simulation. It kicks things off, lets you pick a car park, and handles the iterative loop that simulates real-world parking activity.

Core Responsibilities:

- Sets up different car parks, each with their own type of ID reader (like *BarcodeReader* or *NumberPlateReader*)
- Lets the user pick which one to simulate
- Starts and controls the simulation loop
- Hands over control to *CarPark.manageOperations()* for all the actual parking logic

Key Features:

- **Car Park Selection** – Users get to pick from a list of car parks that differ in location and ID reader type, which helps test different setups.
- **Simulation Loop** - It runs a loop that mimics real-world car park operations. After each round, it checks if you want to keep going.
- **Smart ID reader check** – It automatically figures out whether a car park uses barcode or number plate IDs and tells you how the system will identify your vehicle. It's a small touch but helps keep things user-friendly.

One thing I really like about this class is that it doesn't try to do too much. It just runs the loop and leaves the actual parking logic (like opening barriers, detecting cars, and issuing tickets) to the *CarPark* class. That separation makes the code easier to follow and maintain.

Static vs Dynamic Memory Allocation in Java

In Java  2 static memory allocation typically applies to fixed-size data structures like arrays, where memory is reserved on the stack at compile time. This approach works well only when a required size is known in advance. On the other hand, dynamic memory allocation is used by structures like ArrayList, HashMap, and other classes in the Java Collections Framework, which allocate memory on the heap at runtime. These structures can grow or shrink as needed, making them more memory-efficient in real-world scenarios [6].

For example, when designing a system like an Employee Information Management System (EIMS) as described in [6], allocating a fixed-size array of 10,000 employees from the start would waste memory if only 10 employees exist initially. A dynamic structure would allocate memory only as new records are added and reclaim it when they are removed, which leads to better memory utilisation [6].



Selecting the Best Data Structure for the Car Park Application

When building the Car Park application, I knew I needed a reliable way to keep track of parked vehicles and their corresponding tickets. My first thought was to look at common data structures in Java like arrays, lists, and sets. Each of those had some merits, but once I thought about the kind of operations the system would be doing regularly, like quickly checking if a car is already parked, retrieving a vehicle's info using its ID, or removing a vehicle when it leaves – HashMap stood out.

The main reason I went with a HashMap was its efficiency. It gives me an average time complexity of $O(1)$ for insertions, lookups, and deletions [7], which is exactly the kind of performance I need for a system that might be accessed frequently and grow in size over time. The constant-time access makes it really practical when you're trying to simulate real-time car park activity.

Another thing that made HashMap the right fit is how it works with key-value pairs. In my application, every vehicle has a unique identifier, either a number plate or barcode, and this pairs perfectly with a map-based structure. So in the *CarPark* class, I've got one HashMap that links the vehicle IDs to their *Vehicle* objects, and another that links those same IDs to their active *Ticket* objects. This setup makes it super straightforward to manage vehicle entries and exits, issue tickets, and validate them when the vehicle leaves.

As mentioned earlier, I also considered using an ArrayList, but indexing by position doesn't make sense when you're working with unique IDs. A HashSet crossed my mind too, but it wouldn't have given me the quick retrieval or easy access to associated ticket info that HashMap does.

The only real downside with HashMap is that in rare worst-case scenarios, like when a lot of keys hash to the same value, it can drop to $O(n)$ performance [7]. But honestly, with good hash functions (and Java's built-in handling), it's not something I'm worried about for this use case.

So overall, using HashMap in the Car Park application made a lot of sense. It's fast, works really well with how vehicle IDs are used and scales nicely if the number of vehicles grows, which is pretty much everything I was looking for.

Unit Test Documentation

This section presents unit tests developed using Junit for a selection of core classes within the system. The focus has been placed on methods that represent critical functionality, such as parking logic, barcode generation, and system interaction flow.

CarParkTest

This section documents the unit tests written for the *CarPark* class, which is central to managing vehicle parking, ticketing, and space allocation.

Test Methods and Explanations

1. `parkVehicle_NewVehicle_VehicleShouldBeAdded`

Purpose: Verifies that a new vehicle can be parked successfully. Asserts that it is added to the parked vehicle list and the available space count decreases.

```
@Test
public void parkVehicle_NewVehicle_VehicleShouldBeAdded(){

    Object vehicleId = "AB12UOK";
    Vehicle vehicle = new Car(make: "Toyota", model: "Corolla", year: 2022);

    carPark.parkVehicle(vehicleId, vehicle);

    assertTrue(carPark.getParkedVehicles().containsKey(vehicleId));
    assertEquals(expected: 9, carPark.spacesLeft());

    // Check that the vehicle info was printed (from the park message)
    assertTrue(outputStream.toString().trim().contains("with ID " + vehicleId + " successfully parked"));
}
```

Figure 19: parkVehicle_NewVehicle_VehicleShouldBeAdded()

2. unparkVehicle_VehicleExists_VehicleShouldBeRemoved

Purpose: Ensures that an existing vehicle can be removed correctly, updating the space count and internal data structure.

```
@Test
public void unparkVehicle_VehicleExists_VehicleShouldBeRemoved(){

    Object vehicleId = "AB12UOK";
    Vehicle vehicle = new Car(make: "Toyota", model: "Corolla", year: 2022);
    // Start by parking a vehicle
    carPark.parkVehicle(vehicleId, vehicle);

    // Unpark the vehicle
    carPark.unparkVehicle(vehicleId);

    //Confirm that the vehicle was removed from the car park
    assertEquals(expected: 0, carPark.getParkedVehicles().size(), message: "Should be no more vehicles left");
    assertFalse(carPark.getParkedVehicles().containsKey(vehicleId));
    assertEquals(expected: 10, carPark.spacesLeft()); // free spaces should be restored
}
```

Figure 20: unparkVehicle_VehicleExists_VehicleShouldBeRemoved()

3. unparkVehicle_VehicleDoesNotExist_ShouldShowError

Purpose: Confirms that attempting to unpark a non-existent vehicle results in an appropriate error message.

```

@Test 1 Jayvibee
public void unparkVehicle_VehicleDoesNotExist_ShouldShowError(){
    Object vehicleId = "NONEXISTENT123";
    carPark.unparkVehicle(vehicleId);

    //Confirm that the vehicle was not removed and the spaces left is still down by 1
    assertFalse(carPark.getParkedVehicles().containsKey(vehicleId)); // Should still be empty
    assertTrue(outputStream.toString().trim().contains("Exit denied"));
}

```

Figure 21: unparkVehicle_VehicleDoesNotExist_ShouldShowError()

4. unparkVehicle_PayableVehicle_ShouldCalculateFee

Purpose: Checks if a vehicle that implements the Payable interface is charged a fee upon exit.

```

@Test 1 Jayvibee
public void unparkVehicle_PayableVehicle_ShouldCalculateFee(){
    Object vehicleId = "0Y45U0K";
    Vehicle payableVehicle = new Car(make: "Hyundai", model: "i30", year: 2017); //Create an in

    carPark.parkVehicle(vehicleId, payableVehicle);

    carPark.unparkVehicle(vehicleId);

    assertTrue(outputStream.toString().trim().contains("Parking fee: £"));
}

```

Figure 22: unparkVehicle_PayableVehicle_ShouldCalculateFee()

5. createVehicle_WhenUserSelectsCar_ShouldReturnCar

Purpose: Tests the creation of a car using mocked input values.

```

@Test
public void createVehicle_WhenUserSelectsCar_ShouldReturnCar() {
    // try-with-resource
    try (MockedStatic<InputValidator> mockedInput = Mockito.mockStatic(InputValidator.class)) {
        // Simulate user input for vehicle type selection and details
        mockedInput.when(() -> InputValidator.getNumberInput(Mockito.anyString())).thenReturn(1, Mockito.anyString());
        mockedInput.when(() -> InputValidator.getStringInput(Mockito.anyString())).thenReturn("Toyota", "Corolla");

        // Call the method
        Vehicle vehicle = carPark.createVehicle();

        // Assertions
        assertNotNull(vehicle);
        assertTrue(vehicle instanceof Car);
        assertEquals("expected: 'Car: Toyota Corolla 2022', vehicle.displayInfo()");
    }
}

```

Figure 23: createVehicle_WhenUserSelectsCar_ShouldReturnCar()

6. createVehicle_WhenUserSelectsMotorcycle_ShouldReturnMotorcycle

Purpose: Tests the correct construction of a motorcycle instance via simulated input.

```

@Test
public void createVehicle_WhenUserSelectsMotorcycle_ShouldReturnMotorcycle() {
    try (MockedStatic<InputValidator> mockedInput = Mockito.mockStatic(InputValidator.class)) {
        mockedInput.when(() -> InputValidator.getNumberInput(Mockito.anyString())).thenReturn(2, Mockito.anyString());
        mockedInput.when(() -> InputValidator.getStringInput(Mockito.anyString())).thenReturn("Honda", "CBR500R");

        Vehicle vehicle = carPark.createVehicle();

        assertNotNull(vehicle);
        assertTrue(vehicle instanceof Motorcycle);
        assertEquals("expected: 'Motorcycle: Honda CBR500R 2021', vehicle.displayInfo()");
    }
}

```

Figure 24: createVehicle_WhenUserSelectsMotorcycle_ShouldReturnMotorcycle()

7. createVehicle_WhenUserSelectsTruck_ShouldReturnTruck

Purpose: Validates correct creation of a truck from user selection.

```

@Test
public void createVehicle_WhenUserSelectsTruck_ShouldReturnTruck() {
    try (MockedStatic<InputValidator> mockedInput = Mockito.mockStatic(InputValidator.class)) {
        mockedInput.when(() -> InputValidator.getNumberInput(Mockito.anyString())).thenReturn(3, Mockito.anyString());
        mockedInput.when(() -> InputValidator.getStringInput(Mockito.anyString())).thenReturn("Ford", "F-150");

        Vehicle vehicle = carPark.createVehicle();

        assertNotNull(vehicle);
        assertTrue(vehicle instanceof Truck);
        assertEquals("expected: 'Truck: Ford F-150 2020', vehicle.displayInfo()");
    }
}

```

Figure 25: createVehicle_WhenUserSelectsTruck_ShouldReturnTruck()

8. createVehicle_WhenUserSelectsInvalidOption_ShouldReturnCar

Purpose: Ensures that an invalid input defaults to creating a Car.

```
@Test
public void createVehicle_WhenUserSelectsInvalidOption_ShouldReturnCar() {
    try (MockedStatic<InputValidator> mockedInput = Mockito.mockStatic(InputValidator.class)) {
        mockedInput.when(() -> InputValidator.getNumberInput(Mockito.anyString())).thenReturn(99, 2018);
        mockedInput.when(() -> InputValidator.getStringInput(Mockito.anyString())).thenReturn("Default", "Car");

        Vehicle vehicle = carPark.createVehicle();

        assertNotNull(vehicle);
        assertTrue(vehicle instanceof Car); // Default case should return a Car
        assertEquals("Car: Default Car 2018", vehicle.displayInfo());
    }
}
```

Figure 26: createVehicle_WhenUserSelectsInvalidOption_ShouldReturnCar()

9. issueTicket_WhenCalledWithValidVehicleId_ShouldReturnTicket

Purpose: Confirms that issuing a ticket adds it to the internal map with the correct ID.

```
@Test
public void issueTicket_WhenCalledWithValidVehicleId_ShouldReturnTicket() {
    String vehicleId = "0Y45U0K";
    carPark.issueTicket(vehicleId);

    // Assert
    assertTrue(carPark.getActiveTickets().containsKey(vehicleId), "Ticket should be stored in activeTickets");
    assertEquals(vehicleId, carPark.getActiveTickets().get(vehicleId).getId(), "Ticket should have the correct vehicleId");
}
```

Figure 27: issueTicket_WhenCalledWithValidVehicleId_ShouldReturnTicket()

10. issueTicket_WhenCalled_ShouldPrintCorrectTicketDetails

Purpose: Validates that ticket details (ID, car park name, etc.) are printed correctly.

```
@Test
public void issueTicket_WhenCalled_ShouldPrintCorrectTicketDetails() {
    String vehicleId = "XY34ABC";

    carPark.issueTicket(vehicleId);

    // Get the printed output
    String printedOutput = outputStream.toString().trim();

    // Assert that the printed ticket contains essential details
    assertTrue(printedOutput.contains("PARKING TICKET"), message: "Ticket header should be present");
    assertTrue(printedOutput.contains("Car Park: Test Park"), message: "Car park name should be printed");
    assertTrue(printedOutput.contains("ID: " + vehicleId), message: "Vehicle ID should be printed");
    assertTrue(printedOutput.contains("Keep this ticket safe!"), message: "Safety message should be printed");
}
```

Figure 28: issueTicket_WhenCalled_ShouldPrintCorrectTicketDetails()

11. removeTicket_TicketExists_ShouldRemoveTicketAndPrintMessage

Purpose: Tests the successful removal of an existing ticket.

```
@Test
public void removeTicket_TicketExists_ShouldRemoveTicketAndPrintMessage() {
    String vehicleId = "XY34ABC";
    carPark.issueTicket(vehicleId); // Issue a ticket first to ensure it exists

    carPark.removeTicket(vehicleId);

    assertFalse(carPark.getActiveTickets().containsKey(vehicleId), message: "Ticket should be removed");
    assertTrue(outputStream.toString().trim().contains("Ticket removed for vehicle with ID: " + vehicleId), message: "Ticket removed message should be printed");
}
```

Figure 29: removeTicket_TicketExists_ShouldRemoveTicketAndPrintMessage()

12. removeTicket_TicketDoesNotExist_ShouldPrintErrorMessage

Purpose: Ensures that an appropriate message is shown when attempting to remove a non-existent ticket.

```

@Test
public void removeTicket_TicketDoesNotExist_ShouldPrintErrorMessage() {
    String nonExistentVehicleId = "ZZ34ABC";
    carPark.removeTicket(nonExistentVehicleId);

    assertTrue(outputStream.toString().trim().contains("No active ticket found for ID: " + nonExistentVehicleId), "message");
}

```

Figure 30: removeTicket_TicketDoesNotExist_ShouldPrintErrorMessage()

Test Results Summary

Below is a screenshot showing all tests in *CarParkTest* passing successfully.

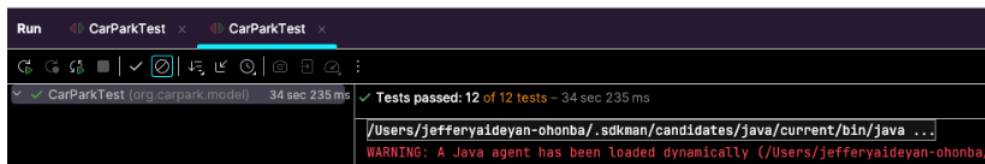


Figure 31: Unit test result for CarPark class

BarcodeReaderTest

This section outlines the unit tests for the *BarcodeReader* class, responsible for generating barcodes, reading user input, and displaying messages based on user location (entrance or exit).

Test Methods and Explanations

1. generateBarcode_ShouldReturnNumberBetween1000And9999

Purpose: Ensures that the generated barcode is within the expected range (100000–999999).

```

@Test
void generateBarcode_ShouldReturnNumberBetween1000And9999(){
    int generatedBarcode = barcodeReader.generateBarcode();

    // Assert that the generated barcode is a four-digit number
    assertTrue(condition generatedBarcode >= 100000 && generatedBarcode <= 999999, message: "Generated barcode should be a four-digit number");
}

```

Figure 32: generateBarcode_ShouldReturnNumberBetween1000And9999()

2. readId_WhenGivenValidInput_ShouldReturnParsedBarcode

Purpose: Verifies that a mocked readId() call returns the expected barcode value.

```
@Test 1 Jayvibee
public void readId_WhenGivenValidInput_ShouldReturnParsedBarcode(){
    doReturn(toBeReturned: 1234).when(spyBarcodeReader).readId();

    assertEquals(expected: 1234, spyBarcodeReader.readId());
}
```

Figure 33: readId_WhenGivenValidInput_ShouldReturnParsedBarcode()

3. displayEntranceOrExitMessage_WhenLocationIsEntrance_ShouldPrintEntranceMessage

Purpose: Confirms that the correct informational message is printed when the location is "Entrance".

```
@Test 1 Jayvibee
public void displayEntranceOrExitMessage_WhenLocationIsEntrance_ShouldPrintEntranceMessage() {
    barcodeReader.displayEntranceOrExitMessage(location: "Entrance");

    String output = outputStream.toString().trim();
    String expectedOutput = "\t\t✔ A barcode will be generated for your vehicle.\n\t\tIt will be printed on your ticket. Please keep your ticket safe.".trim();

    assertEquals(expectedOutput, output);
}
```

Figure 34: displayEntranceOrExitMessage_WhenLocationIsEntrance_ShouldPrintEntranceMessage()

4. displayEntranceOrExitMessage_WhenLocationIsExit_ShouldPrintExitMessage

Purpose: Verifies that the correct message is shown when the user is at the exit.

```
@Test 1 Jayvibee
public void displayEntranceOrExitMessage_WhenLocationIsExit_ShouldPrintExitMessage() {
    barcodeReader.displayEntranceOrExitMessage(location: "Exit");

    String output = outputStream.toString().trim();
    String expectedOutput = "Please enter the barcode on your ticket to exit>>.".trim();

    assertEquals(expectedOutput, output);
}
```

Figure 35: displayEntranceOrExitMessage_WhenLocationIsExit_ShouldPrintExitMessage()

Test Results Summary

All *BarcodeReaderTest* unit tests executed successfully, ensuring reliability of barcode-related functionality.

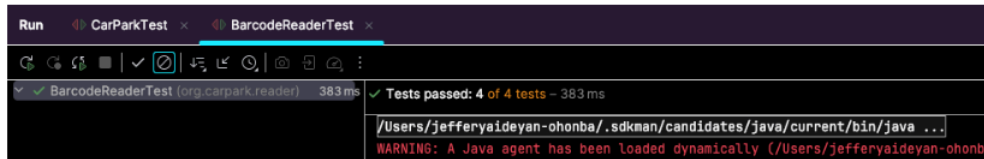


Figure 36: Unit test result for *BarcodeReader* class unit test

CarParkSystemTest

The *CarParkSystemTest* class focuses on validating user interaction through car park selection. It ensures correct behaviour for both valid and invalid user inputs using mocked Scanner objects.

Test Methods and Explanations

1. chooseCarPark_ValidInputType_ShouldReturnSelectedCarPark

Purpose: Verifies that valid user input selects the correct car park.

```
@Test
void chooseCarPark_ValidInputType_ShouldReturnSelectedCarPark() {
    // Mock scanner to simulate user choosing "0"
    Scanner mockScanner = mock(Scanner.class);
    when(mockScanner.nextLine()).thenReturn("1"); //simulated user input

    CarPark selectedCarPark = carParkSystem.chooseCarPark(mockScanner);

    // Verify the expected car park was chosen
    assertNotNull(selectedCarPark);
    assertEquals("expected: \"NCP - Orchard Street\", selectedCarPark.getName());
}
```

Figure 37: chooseCarPark_ValidInputType_ShouldReturnSelectedCarPark()

2. chooseCarPark_InvalidInput_ShouldPromptAgain

Purpose: Ensures the system handles invalid input by prompting the user again until a valid option is provided.

```
@Test
void chooseCarPark_InvalidInput_ShouldPromptAgain(){
    Scanner mockScanner = mock(Scanner.class);
    when(mockScanner.nextLine()).thenReturn("3", "6");

    CarPark selectedCarPark = carParkSystem.chooseCarPark(mockScanner);

    // Verify Scanner was called at least twice (for invalid then valid input)
    verify(mockScanner, atLeast(2)).nextLine();
    assertEquals("NCP - The City Gates", selectedCarPark.getName());
}
```

Figure 38: chooseCarPark_InvalidInput_ShouldPromptAgain()

Test Results Summary

All tests in *CarParkSystemTest* passed successfully, confirming robust handling of user selection and input validation.

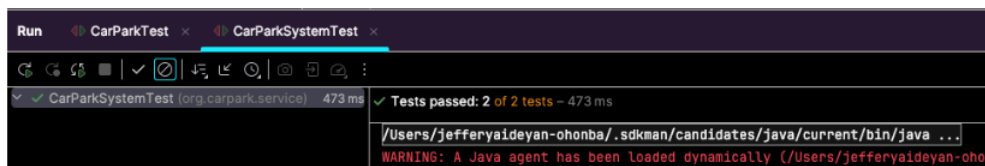


Figure 39: Unit test result for CarParkSystem class

Working Program Demonstration

This section includes screenshots demonstrating the live, running version of the car park simulation system. It highlights key interactions such as starting the application, selecting a car park, parking and unparking vehicles, and ticket generation. For a more detailed demonstration of the system in action, a video walkthrough is available here: <https://youtu.be/3rte3z1DCyA>.

1. Program Startup

```
/Users/jefferyaideyan-ohonba/.sdkman/candidates,  
Welcome to the Car Park Simulation!  
Select a car park from the list below:  
0. NCP - The City Gates  
1. NCP - Orchard Street
```

Figure 40: Working program demo - program startup

2. Choosing a Car Park

```
/Users/jefferyaideyan-ohonba/.sdkman/candidates/java/curre  
Welcome to the Car Park Simulation!  
Select a car park from the list below:  
0. NCP - The City Gates  
1. NCP - Orchard Street  
1  
You have selected NCP - Orchard Street located in Swansea  
  
*****Simulation Starts*****  
Is a vehicle at the entrance? (yes/no):
```

Figure 41: Working program demo – choosing a car park

3. Parking a vehicle

```
*****Simulation Starts*****
Is a vehicle at the entrance? (yes/no):
yes
Sensor: Vehicle detected at the entrance!
Enter your vehicle number plate (e.g AB12 XYZ or CD34LMN) to enter.
0Y54YHJ
🔍 Checking Number Plate : 0Y54YHJ...
Select Vehicle Type:
1. Car
2. Motorcycle
3. Truck
Enter your choice (1-3):
1
Enter vehicle make:
Hyundai
Enter vehicle model:
i30
Enter vehicle year:
2017
🔴 Entrance barrier raising...✅ Barrier fully raised!
Car: Hyundai i30 2017 with ID 0Y54YHJ successfully parked! Remaining spaces: 4
🔴 Entrance barrier lowering...✅ Barrier fully lowered!
Sensor: Turning off entrance sensor.
```

Figure 42: Working program demo - parking a vehicle

4. Issuing a Ticket

```
● Printing ticket...
|=====|
|      🚗 PARKING TICKET      |
|=====|
| 🚗 Car Park: NCP - Orchard Street |
| ⌚ Entry: 2025-04-10T21:51:02.092982 |
| 🆔 ID: OY54YHJ              |
|=====|
|      💎 Keep this ticket safe! 💎      |
|=====|

Is a vehicle at the exit? (yes/no):
```

Figure 43: Working program demo - issuing a ticket

5. Unparking a Vehicle

```
Is a vehicle at the exit? (yes/no):
yes
Sensor: Vehicle detected at the exit!
Enter your vehicle number plate to exit.
OY54YHJ
🔍 Checking Number Plate : OY54YHJ...
● Exit barrier raising...✅ Barrier fully raised!
Car: Hyundai i30 2017 with ID OY54YHJ successfully exited the car park! Remaining spaces: 5
💰 Parking fee: £5.0
● Exit barrier lowering...✅ Barrier fully lowered!
Sensor: Turning off exit sensor.
Ticket removed for vehicle with ID: OY54YHJ
```

Figure 44: Working program demo - unparking a vehicle

Conclusion

Overall, this project allowed me to apply core OOP principles in a practical and hands-on way. The design leverages encapsulation to manage class responsibilities, inheritance and polymorphism to support different vehicle types, and abstraction to separate concerns across components such as input handling and ticket generation. Unit tests were implemented for key classes to verify correctness and behaviour. While adding features such as persistent storage would take the system even further, the current implementation already does a solid job demonstrating strong object-oriented design and development practices.

Video Demo Link

<https://www.youtube.com/watch?v=3rte3z1DCyA>

Bibliography

- [1] M. O. Elish, "Measuring inheritance coupling in object-oriented systems," ProQuest Dissertations & Theses, 1999.
- [2] F. Fallucchi and M. Gozzi, "Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in Object-Oriented Programming," *Applied Sciences*, vol. 14, (12), pp. 5083, 2024. Available: <https://www.proquest.com/scholarly-journals/puzzle-pattern-systematic-approach-multiple/docview/3072252234/se-2>. DOI: <https://doi.org/10.3390/app14125083>.
- [3] D. Mumaugh, *Abstract Classes: Base Classes Providing a Common Interface*, DePaul University. [Online]. Available: <https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/abcpvf.pdf>
- [4] H. Ali, M. Hamza, M. Y. Rashid, and T. A. Ahmed, "Exploring Polymorphism: Flexibility and Code Reusability in Object-Oriented Programming," *Passer Journal of Basic and Applied Sciences*, vol. 6, Special Issue, pp. 502-512, 2024, doi: 10.24271/psr.2024.189667.
- [5] Péter Soha and N. Pataki, "Instantiation of Java Generics," *Acta Cybernetica*, vol. 25, (4), pp. 897-908, 2022. Available: <https://www.proquest.com/scholarly-journals/instantiation-java-generics/docview/2781702705/se-2>. DOI: <https://doi.org/10.14232/actacyb.284073>.
- [6] N. V. Patil and P. S. Irabashetti, "Dynamic Memory Allocation: Role in Memory Management," *International Journal of Current Engineering and Technology*, vol. 4, no. 2, pp. 531-535, Apr. 2014.
- [7] I. Chivers and J. Sleightholme, "An Introduction to Algorithms and the Big O Notation," in *Introduction to Programming with Fortran*, Cham: Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-75502-1_23
- [8] Batten, "Big-O Notation," *Massachusetts Institute of Technology*, 2001. [Online]. Available: https://web.mit.edu/16.070/www/lecture/big_o.pdf

Appendix A: Source Code

A.1 CarParkSystem.java

```
package org.carpark.service;

import org.carpark.model.CarPark;
import org.carpark.reader.BarcodeReader;
import org.carpark.reader.NumberPlateReader;
import org.carpark.util.InputValidator;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class CarParkSystem {
    private List<CarPark> carParks;

    public CarParkSystem() {
        this.carParks = new ArrayList<>();
    }

    public void startSimulation() {
        System.out.println("Welcome to the Car Park Simulation!");
        System.out.println("Select a car park from the list below:");
        Scanner scanner = new Scanner(System.in);
        CarPark chosenCarPark = this.chooseCarPark(scanner);

        System.out.printf("You have selected %s located in %s\n",
chosenCarPark.getName(), chosenCarPark.getLocation());

        // Check what type of id reader is used in the chosen car park so
that the user can be informed
        if(chosenCarPark.getIdReader() instanceof BarcodeReader){
            System.out.println("The car park you chose uses a barcode
system. Your vehicle ID will be generated for you.");
        }

        System.out.println();
        System.out.println("*****Simulation Starts*****");
        // Start the simulation loop.
        int simulationIterations = 15;
        for(int i = 0; i < simulationIterations; i++){
            chosenCarPark.manageOperations();
            System.out.println("Iteration " + (i + 1) + " of " +
simulationIterations + ": polling car park components and reacting to
state");

            // Ask the user if they want to continue only when they are not
on the second to last iteration
            if(i != simulationIterations - 1){
                String continueChoice = InputValidator.getYesOrNoInput("Do
you want to continue to the next car park operation? (yes/no):");
                if (continueChoice.equalsIgnoreCase("no")) {
                    System.out.println("Simulation ended.");
                    break; // Exit the loop
                }
            }
        }
    }
}
```

```

    }

    public CarPark chooseCarPark(Scanner scanner) {
        carParks.add(new CarPark("NCP - The City Gates", "Swansea", 5, new
BarcodeReader()));
        carParks.add(new CarPark("NCP - Orchard Street", "Swansea", 5, new
NumberPlateReader()));

        for (int i =0; i<carParks.size(); i++) {
            System.out.println(i + ". " + carParks.get(i).getName());
        }

        while (true) {
            int input = InputValidator.validateNumber(scanner.nextLine());
// Validates the user input to make sure it's a number

            if (input >= 0 && input < carParks.size()) {
                return carParks.get(input);
            }

            System.out.println("Invalid option. Try again.");
        }
    }
}

```

A.2 CarPark.java

```

package org.carpark.model;

import org.carpark.interfaces.Payable;
import org.carpark.reader.BarcodeReader;
import org.carpark.reader.IDReader;
import org.carpark.util.InputValidator;

import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;
import java.util.HashMap;

public class CarPark {
    private String name;
    private String location;
    private int capacity;
    private int freeSpaces;
    private Barrier entryBarrier;
    private Barrier exitBarrier;
    private Sensor entrySensor;
    private Sensor exitSensor;
    private FullSign fullSign;
    private IDReader<?> idReader;
    private HashMap<Object, Vehicle> parkedVehicles;
    private HashMap<Object, Ticket<?>> activeTickets;

    public CarPark(String name, String location, int capacity, IDReader
idReader) {
        this.name = name;
        this.location = location;
        this.capacity = capacity;
        this.freeSpaces = this.capacity;
    }
}

```

```

        this.entryBarrier = new Barrier();
        this.exitBarrier = new Barrier();
        this.entrySensor = new Sensor();
        this.exitSensor = new Sensor();
        this.fullSign = new FullSign();
        this.idReader = idReader;
        this.parkedVehicles = new HashMap<>();
        this.activeTickets = new HashMap<>();
    }

    // Manage/process car park operations(car park cycle)
    public void manageOperations(){
        String carAtEntrance = InputValidator.getYesOrNoInput("Is a vehicle
at the entrance? (yes/no):"); // Validate that the user chooses either yes
or no.

        // Handle vehicle entrance
        if(carAtEntrance.equalsIgnoreCase("yes")){
            entrySensor.detectCar("entrance");

            // Check if there is space in the car park first before trying
to let them in
            if(fullSign.isLightOn()){
                fullSign.displayFullSign();
                entrySensor.reset("entrance"); //turn entry sensor off
            }
            else{
                idReader.displayEntranceOrExitMessage("entrance");
                // If the car park uses a barcode reader, a barcode can be
generated for the vehicle. if not read and validate their id
                Object vehicleId = idReader instanceof BarcodeReader ?
((BarcodeReader) idReader).generateBarcode() : idReader.readId(); //
Explicit type casting is needed because generateBarcode() is a method
exclusive to the org.carpark.readers.BarcodeReader class and is not
available in the org.carpark.readers.IDReader class.

                // after getting the vehicleId from the user or generating
one, attempt to park the vehicle
                if(!parkedVehicles.containsKey(vehicleId)){ // Check that a
vehicle with the id doesn't exist
                    // Create a new vehicle object
                    Vehicle vehicle = createVehicle();

                    parkVehicle(vehicleId, vehicle);
                }else{
                    System.out.println("❌ Duplicate entry detected!
Vehicle with ID " + vehicleId + " is already parked.");
                }
            }
        }

        // Handle vehicle exit
        String carAtExit = InputValidator.getYesOrNoInput("Is a vehicle at
the exit? (yes/no):");
        if(carAtExit.equalsIgnoreCase("yes")){
            exitSensor.detectCar("exit");
            idReader.displayEntranceOrExitMessage("exit");
            Object vehicleId = idReader.readId();

            if(parkedVehicles.containsKey(vehicleId)){

```

```

        unparkVehicle(vehicleId);
    }else{
        System.out.printf("❌ Exit denied. A vehicle with the ID - %s is not in the car park.\n", vehicleId);
    }
}

// Check if the car park is full after doing the entry and exit steps
System.out.println();
if(spacesLeft() <= 0){
    fullSign.switchOn();
}else{
    fullSign.switchOff();
}
}

public void parkVehicle(Object vehicleId, Vehicle vehicle){
    entryBarrier.raise("Entrance");
    parkedVehicles.put(vehicleId, vehicle);
    decrementSpaces();
    System.out.println(vehicle.displayInfo() + " with ID " + vehicleId + " successfully parked! Remaining spaces: " + spacesLeft());
    entryBarrier.lower("Entrance");
    entrySensor.reset("entrance");
    // Generate a ticket with the vehicle ID
    System.out.println();
    issueTicket(vehicleId);
    System.out.println();
}

public void unparkVehicle(Object vehicleId){
    Vehicle removedVehicle = parkedVehicles.remove(vehicleId);
    //Extra fail-safe check to make sure that an invalid vehicle ID is not passed in
    if(removedVehicle == null){
        System.out.println("Exit denied. A vehicle with the ID - " + vehicleId + " is not in the car park.");
        return; //Exit early to prevent NullPointerException
    }
    exitBarrier.raise("Exit");
    incrementSpaces();
    System.out.println(removedVehicle.displayInfo() + " with ID " + vehicleId + " successfully exited the car park! Remaining spaces: " + spacesLeft());

    // Calculate parking fee if applicable
    if(removedVehicle instanceof Payable && activeTickets.containsKey(vehicleId)){
        LocalDateTime entryTime = activeTickets.get(vehicleId).getEntryTime(); // get the entry time from the ticket issued to the vehicle
        LocalDateTime exitTime = LocalDateTime.now().plusHours(2);
        //always add 2 hours to the exit time by default (for testing)
        long hoursParked = ChronoUnit.HOURS.between(entryTime, exitTime);
        double fee = ((Payable) removedVehicle).calculateFee(hoursParked);
        System.out.println("💰 Parking fee: f" + fee);
    }
}

```

```

        exitBarrier.lower("Exit");
        exitSensor.reset("exit");
        // Remove the ticket for that vehicle id
        removeTicket(vehicleId);
    }

    public Vehicle createVehicle(){
        System.out.println("Select Vehicle Type: ");
        System.out.println("1. Car");
        System.out.println("2. Motorcycle");
        System.out.println("3. Truck");

        int choice = InputValidator.getNumberInput("Enter your choice (1-3):");
        String make = InputValidator.getStringInput("Enter vehicle make:");
        String model = InputValidator.getStringInput("Enter vehicle model:");
        int year = InputValidator.getNumberInput("Enter vehicle year:");

        switch (choice) {
            case 1:
                return new Car(make, model, year);
            case 2:
                return new Motorcycle(make, model, year);
            case 3:
                return new Truck(make, model, year);
            default:
                System.out.println("Invalid choice. Defaulting to Car.");
                return new Car(make, model, year);
        }
    }

    public <T> void issueTicket(T vehicleId){
        Ticket<T> ticket = new Ticket<>(vehicleId, this.name); // The ticket created will be the same type as the vehicle id passed in (Integer, String)
        activeTickets.put(vehicleId, ticket);
        ticket.printTicket();
    }

    public <T> void removeTicket(T vehicleId){
        if(activeTickets.containsKey(vehicleId)){
            activeTickets.remove(vehicleId);
            System.out.println("Ticket removed for vehicle with ID: " + vehicleId);
        }
        else{
            System.out.println("No active ticket found for ID: " + vehicleId);
        }
    }

    void incrementSpaces(){
        freeSpaces++;
    }

    void decrementSpaces(){
        freeSpaces--;
    }

```

```

    }

    int spacesLeft() {
        return freeSpaces;
    }

    public String getName() {
        return name;
    }

    public String getLocation() {
        return location;
    }

    public IDReader getIdReader() {
        return idReader;
    }

    public HashMap<Object, Vehicle> getParkedVehicles() {
        return parkedVehicles;
    }

    public HashMap<Object, Ticket<?>> getActiveTickets() {
        return activeTickets;
    }
}

```

A.3 Ticket.java

```

package org.carpark.model;

import java.time.LocalDateTime;

public class Ticket <T>{
    private T id;
    private LocalDateTime entryTime;
    private String carParkName;

    public Ticket(T id, String carParkName){
        this.id = id;
        this.entryTime = LocalDateTime.now();
        this.carParkName = carParkName;
    }

    public T getId(){
        return id;
    }

    public LocalDateTime getEntryTime(){
        return entryTime;
    }

    public void printTicket(){
        System.out.print("\u25cf Printing ticket");
        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(700);
                System.out.print(".");
            } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
}
System.out.println();
String border = "|=====|";
String separator = "|=====|";
String footer = "|=====|";

System.out.println(border);
System.out.println("| 🚗 PARKING TICKET |");
System.out.println(separator);
System.out.printf("| 🚗 Car Park: %-20s |\n", carParkName);
System.out.printf("| 🕒 Entry: %-20s |\n", entryTime);
System.out.printf("| 📄 ID: %-20s |\n", id);
System.out.println(separator);
System.out.println("| 💎 Keep this ticket safe! 💎 |");
System.out.println(footer);
}
}

```

A.4 Vehicle.java

```

package org.carpark.model;

public abstract class Vehicle {
    protected String make;
    protected String model;
    protected int year;

    public Vehicle(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public abstract String displayInfo();
}

```

A.5 Car.java

```

package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Car extends Vehicle implements Payable {

    public Car(String make, String model, int year) {
        super(make, model, year);
    }

    @Override
    public String displayInfo(){
        return "Car: " + make + " " + model + " " + year;
    }

    @Override

```

```

        public double calculateFee(long hours){ // Method from payable
interface
        return hours * 2.50; // £2.50 per hour
    }
}

```

A.6 Motorcycle.java

```

package org.carpark.model;

public class Motorcycle extends Vehicle {
    public Motorcycle(String make, String model, int year) {
        super(make, model, year);
    }

    @Override
    public String displayInfo(){
        return "Motorcycle: " + make + " " + model + " " + year;
    }
}

```

A.7 Truck.java

```

package org.carpark.model;

import org.carpark.interfaces.Payable;

public class Truck extends Vehicle implements Payable {

    public Truck(String make, String model, int year) {
        super(make, model, year);
    }

    @Override
    public String displayInfo(){
        return "Truck: " + make + " " + model + " " + year;
    }

    @Override
    public double calculateFee(long hours) {
        return hours * 5.00; // Trucks pay £5 per hour
    }
}

```

A.8 InputValidator.java

```

package org.carpark.util;

import java.util.Scanner;

public class InputValidator {

    private static Scanner scanner = new Scanner(System.in);
}

```

```

    public static String getYesOrNoInput(String prompt) {
        System.out.println(prompt);
        String input = scanner.nextLine();
        while(true){
            if(input.equalsIgnoreCase("yes") ||
input.equalsIgnoreCase("no")){
                return input;
            }
            System.out.println("Please enter either yes or no");
            input = scanner.nextLine(); //Take the user input again
        }
    }

    public static int validateNumber(String input){

        int number = -1;
        boolean isValidNumber = false;

        while(!isValidNumber){
            try{
                number = Integer.parseInt(input);
                isValidNumber = true;
            }catch (NumberFormatException e){
                System.out.println("Invalid number! try again");
                input = scanner.nextLine();
            }
        }
        return number;
    }

    public static int getNumberInput(String prompt){
        System.out.println(prompt);
        return validateNumber(scanner.nextLine());
    }

    public static String getStringInput(String prompt){
        System.out.println(prompt);
        while(true){
            String input = scanner.nextLine();
            if(!input.equalsIgnoreCase("")){
                return input;
            }
            System.out.println("Please enter a value>");
        }
    }
}

```

A.9 IDReader.java

```

package org.carpark.reader;

import java.util.Scanner;

public abstract class IDReader<T> {
    protected Scanner scanner;

    public IDReader() {

```

```

        this.scanner = new Scanner(System.in);
    }

    public abstract T readId();

    public abstract void displayEntranceOrExitMessage(String location);
}

```

A.10 BarcodeReader.java

```

package org.carpark.reader;
import org.carpark.util.InputValidator;

import java.util.Random;

public class BarcodeReader extends IDReader<Integer>{

    public BarcodeReader() {
        super();
    }

    // Generate a random barcode for entry to simulate a real world barcode
    reader machine that prints out a ticket with barcode
    public int generateBarcode() {
        Random random = new Random();
        return 100000 + (random.nextInt(900000));
    }

    @Override
    public Integer readId(){
        // Validate barcode format first here before sending it to get
        checked in the records
        String input = scanner.nextLine().trim();
        System.out.println("\n Checking Barcode : " + input + "...");
        int barcode = InputValidator.validateNumber(input);
        return barcode;
    }

    @Override
    public void displayEntranceOrExitMessage(String location){
        if(location.equalsIgnoreCase("entrance")) {
            System.out.println("\n\t\t✅ A barcode will be generated for your
            vehicle.\n\t\tIt will be printed on your ticket. Please keep your ticket
            safe.");
        }
        else{
            System.out.println("Please enter the barcode on your ticket to
            exit>>.");
        }
    }
}

```

A.11 NumberPlateReader.java

```

package org.carpark.reader;

```

```

public class NumberPlateReader extends IDReader<String> {

    public NumberPlateReader() {
        super();
    }

    @Override
    public String readId() {
        // Validate number plate
        while (true) {
            String numberPlate = scanner.nextLine().trim();
            System.out.println("🔍 Checking Number Plate : " + numberPlate
+ "...");
            if (numberPlate.matches("^[A-Z]{2}\\d{2}\\s?[A-Z]{3}$")) {
                return numberPlate;
            }
            System.out.println("Invalid number plate format! Try again");
        }
    }

    @Override
    public void displayEntranceOrExitMessage(String location) {
        if (location.equalsIgnoreCase("entrance")) {
            System.out.println("Enter your vehicle number plate (e.g AB12
XYZ or CD34LMN) to enter.");
        } else {
            System.out.println("Enter your vehicle number plate to exit.");
        }
    }
}

```

A.12 Payable.java

```

package org.carpark.interfaces;

public interface Payable {
    double calculateFee(long hours);
}

```

A.13 Main.java

```

package org.carpark.main;

import org.carpark.service.CarParkSystem;

public class Main {
    public static void main(String[] args) {
        CarParkSystem carParkSystem = new CarParkSystem();
        carParkSystem.startSimulation();
    }
}

```

A.14 Barrier.java

```

package org.carpark.model;

```

```

public class Barrier {
    private boolean up;

    public Barrier() {
        this.up = false; //Barrier starts in the down position
    }

    void raise(String location){
        if(!isUp()){
            System.out.print("🔴 " + location + " barrier raising");
            for (int i = 0; i < 3; i++) {
                try {
                    Thread.sleep(700);
                    System.out.print(".");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("✅ Barrier fully raised!");
            this.up = true;
        }
    }

    void lower(String location){
        if(isUp()){
            System.out.printf("🔴 " + location + " barrier lowering");
            for (int i = 0; i < 3; i++) {
                try {
                    Thread.sleep(700);
                    System.out.print(".");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("✅ Barrier fully lowered!");
            this.up = false;
        }
    }

    boolean isUp(){
        return this.up;
    }
}

```

A.15 FullSign.java

```

package org.carpark.model;

public class FullSign {
    private boolean lightOn;

    public FullSign() {
        lightOn = false;
    }

    void switchOn(){
        lightOn = true;
    }
}

```

```

    void switchOff(){
        lightOn = false;
    }

    boolean isLightOn() {
        return lightOn;
    }

    void displayFullSign() {
        System.out.println("🚗 FULL SIGN ON: Car park is full!");
    }
}

```

A.16 Sensor.java

```

package org.carpark.model;

public class Sensor {
    private boolean carDetected;

    public Sensor(){
        this.carDetected = false;
    }

    void detectCar(String location){
        this.carDetected = true;
        System.out.println("Sensor: Vehicle detected at the " + location +
            "\n!");
    }

    public void reset(String location){
        this.carDetected = false;
        System.out.println("Sensor: Turning off " + location + " sensor.");
    }

    public boolean isCarPresent(){
        return this.carDetected;
    }
}

```

A.17 CarParkTest.java

```

package org.carpark.model;

import org.carpark.reader.IDReader;
import org.carpark.util.InputValidator;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import static org.junit.jupiter.api.Assertions.*;

```

```

public class CarParkTest {
    private IDReader mockIdReader; // Mocking IDReader to simulate vehicle
    ID reading
    private CarPark carPark;

    private final ByteArrayOutputStream outputStream = new
    ByteArrayOutputStream();
    private PrintStream originalSystemOut;

    @BeforeEach
    public void setUp() {
        mockIdReader = Mockito.mock(IDReader.class);
        carPark = new CarPark("Test Park", "Test Location", 10,
        mockIdReader);

        originalSystemOut = System.out; // Saves the original System.out so
        it can be restored after the test
        System.setOut(new PrintStream(outputStream));
    }

    @Test
    public void parkVehicle_NewVehicle_VehicleShouldBeAdded() {
        Object vehicleId = "AB12UOK";
        Vehicle vehicle = new Car("Toyota", "Corolla", 2022);

        carPark.parkVehicle(vehicleId, vehicle);

        assertTrue(carPark.getParkedVehicles().containsKey(vehicleId));
        assertEquals(9, carPark.spacesLeft());

        // Check that the vehicle info was printed (from the park message)
        assertTrue(outputStream.toString().trim().contains("with ID " +
        vehicleId + " successfully parked"));
    }

    @Test
    public void unParkVehicle_VehicleExists_VehicleShouldBeRemoved() {
        Object vehicleId = "AB12UOK";
        Vehicle vehicle = new Car("Toyota", "Corolla", 2022);
        // Start by parking a vehicle
        carPark.parkVehicle(vehicleId, vehicle);

        // Unpark the vehicle
        carPark.unParkVehicle(vehicleId);

        //Confirm that the vehicle was removed from the car park
        assertEquals(0, carPark.getParkedVehicles().size(), "Should be no
        more vehicles left");
        assertFalse(carPark.getParkedVehicles().containsKey(vehicleId));
        assertEquals(10, carPark.spacesLeft()); // free spaces should be
        restored
    }

    @Test
    public void unParkVehicle_VehicleDoesNotExist_ShouldShowError() {
        Object vehicleId = "NONEXISTENT123";
        carPark.unParkVehicle(vehicleId);
    }
}

```

```

        //Confirm that the vehicle was not removed and the spaces left is
        still down by 1
        assertFalse(carPark.getParkedVehicles().containsKey(vehicleId)); //
        Should still be empty
        assertTrue(outputStream.toString().trim().contains("Exit denied"));
    }

    @Test
    public void unParkVehicle_PayableVehicle_ShouldCalculateFee() {
        Object vehicleId = "OY45UOK";
        Vehicle payableVehicle = new Car("Hyundai", "i30", 2017); //Create
        an instance of a vehicle type that implements payable

        carPark.parkVehicle(vehicleId, payableVehicle);

        carPark.unParkVehicle(vehicleId);

        assertTrue(outputStream.toString().trim().contains("Parking fee:
        £"));
    }

    @Test
    public void createVehicle_WhenUserSelectsCar_ShouldReturnCar() {
        // try-with-resource
        try (MockedStatic<InputValidator> mockedInput =
        Mockito.mockStatic(InputValidator.class)) {
            // Simulate user input for vehicle type selection and details
            mockedInput.when(() ->
            InputValidator.getNumberInput(Mockito.anyString()))
            .thenReturn(1, 2022);
            mockedInput.when(() ->
            InputValidator.getStringInput(Mockito.anyString()))
            .thenReturn("Toyota",
            "Corolla");

            // Call the method
            Vehicle vehicle = carPark.createVehicle();

            // Assertions
            assertNotNull(vehicle);
            assertTrue(vehicle instanceof Car);
            assertEquals("Car: Toyota Corolla 2022",
            vehicle.displayInfo());
        }
    }

    @Test
    public void createVehicle_WhenUserSelectsMotorcycle_ShouldReturnMotorcycle() {
        try (MockedStatic<InputValidator> mockedInput =
        Mockito.mockStatic(InputValidator.class)) {
            mockedInput.when(() ->
            InputValidator.getNumberInput(Mockito.anyString()))
            .thenReturn(2, 2021);
            mockedInput.when(() ->
            InputValidator.getStringInput(Mockito.anyString()))
            .thenReturn("Honda",
            "CBR500R");

            Vehicle vehicle = carPark.createVehicle();

```

```

        assertNotNull(vehicle);
        assertTrue(vehicle instanceof Motorcycle);
        assertEquals("Motorcycle: Honda CBR500R 2021",
vehicle.displayInfo());
    }

    }

    @Test
    public void createVehicle_WhenUserSelectsTruck_ShouldReturnTruck() {
        try (MockedStatic<InputValidator> mockedInput =
Mockito.mockStatic(InputValidator.class)) {
            mockedInput.when(() ->
InputValidator.getNumberInput(Mockito.anyString()))
.thenReturn(3, 2020);
            mockedInput.when(() ->
InputValidator.getStringInput(Mockito.anyString()))
.thenReturn("Ford", "F-
150");

            Vehicle vehicle = carPark.createVehicle();

            assertNotNull(vehicle);
            assertTrue(vehicle instanceof Truck);
            assertEquals("Truck: Ford F-150 2020", vehicle.displayInfo());
        }
    }

    @Test
    public void
createVehicle_WhenUserSelectsInvalidOption_ShouldReturnCar() {
        try (MockedStatic<InputValidator> mockedInput =
Mockito.mockStatic(InputValidator.class)) {
            mockedInput.when(() ->
InputValidator.getNumberInput(Mockito.anyString()))
.thenReturn(99, 2018);
            mockedInput.when(() ->
InputValidator.getStringInput(Mockito.anyString()))
.thenReturn("Default",
"Car");

            Vehicle vehicle = carPark.createVehicle();

            assertNotNull(vehicle);
            assertTrue(vehicle instanceof Car); // Default case should
return a Car
            assertEquals("Car: Default Car 2018", vehicle.displayInfo());
        }
    }

    @Test
    public void
issueTicket_WhenCalledWithValidVehicleId_ShouldReturnTicket() {
        String vehicleId = "OY45UOK";
        carPark.issueTicket(vehicleId);

        // Assert
        assertTrue(carPark.getActiveTickets().containsKey(vehicleId),
"Ticket should be stored in activeTickets");
        assertEquals(vehicleId,
carPark.getActiveTickets().get(vehicleId).getId(), "Ticket should have the
correct vehicleId");
    }
}

```

```

@Test
public void issueTicket_WhenCalled_ShouldPrintCorrectTicketDetails() {
    String vehicleId = "XY34ABC";

    carPark.issueTicket(vehicleId);

    // Get the printed output
    String printedOutput = outputStream.toString().trim();

    // Assert that the printed ticket contains essential details
    assertTrue(printedOutput.contains("📄 PARKING TICKET"), "Ticket
header should be present");
    assertTrue(printedOutput.contains("📍 Car Park: Test Park"), "Car
park name should be printed");
    assertTrue(printedOutput.contains("🚗 ID: " + vehicleId), "Vehicle
ID should be printed");
    assertTrue(printedOutput.contains("💎 Keep this ticket safe! 💎"),
"Safety message should be printed");
}

@Test
public void
removeTicket_TicketExists_ShouldRemoveTicketAndPrintMessage() {
    String vehicleId = "XY34ABC";
    carPark.issueTicket(vehicleId); // Issue a ticket first to ensure
it exists

    carPark.removeTicket(vehicleId);

    assertFalse(carPark.getActiveTickets().containsKey(vehicleId),
"Ticket should be removed");
    assertTrue(outputStream.toString().trim().contains("Ticket removed
for vehicle with ID: " + vehicleId), "Should print correct removal
message");
}

@Test
public void removeTicket_TicketDoesNotExist_ShouldPrintErrorMessage() {
    String nonExistentVehicleId = "ZZ34ABC";
    carPark.removeTicket(nonExistentVehicleId);

    assertTrue(outputStream.toString().trim().contains("No active
ticket found for ID: " + nonExistentVehicleId), "Should print error message
for missing ticket");
}

@AfterEach
public void restoreSystemOut() {
    System.setOut(originalSystemOut); //Clear the output stream after
each test
}
}

```

A.18 BarcodeReaderTest.java

```
package org.carpark.reader;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class BarcodeReaderTest {

    private BarcodeReader barcodeReader;
    private BarcodeReader spyBarcodeReader;
    private final ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
    private PrintStream originalSystemOut;

    @BeforeEach
    public void setUp() {
        barcodeReader = new BarcodeReader();
        spyBarcodeReader = Mockito.spy(barcodeReader); // Spy on the real
instance

        originalSystemOut = System.out; // Saves the original System.out so
it can be restored after the test
        System.setOut(new PrintStream(outputStream)); // Set the new
System.out to capture printed output
    }

    @Test
    void generateBarcode_ShouldReturnNumberBetween1000And9999(){
        int generatedBarcode = barcodeReader.generateBarcode();

        // Assert that the generated barcode is a four-digit number
        assertTrue(generatedBarcode >= 100000 && generatedBarcode <=
999999, "Generated barcode should be a four-digit number");
    }

    @Test
    public void readId_WhenGivenValidInput_ShouldReturnParsedBarcode(){
        doReturn(1234).when(spyBarcodeReader).readId();

        assertEquals(1234, spyBarcodeReader.readId());
    }

    @Test
    public void
displayEntranceOrExitMessage_WhenLocationIsEntrance_ShouldPrintEntranceMess
age() {
        barcodeReader.displayEntranceOrExitMessage("Entrance");
    }
}
```

```

        String output = outputStream.toString().trim();
        String expectedOutput = "\t\t✅ A barcode will be generated for
your vehicle.\n\t\tIt will be printed on your ticket. Please keep your
ticket safe.".trim();

        assertEquals(expectedOutput, output);
    }

    @Test
    public void
displayEntranceOrExitMessage_WhenLocationIsExit_ShouldPrintExitMessage() {
        barcodeReader.displayEntranceOrExitMessage("Exit");

        String output = outputStream.toString().trim();
        String expectedOutput = "Please enter the barcode on your ticket to
exit>>.".trim();

        assertEquals(expectedOutput, output);
    }

    @AfterEach
    void restoreSystemOut() {
        System.setOut(originalSystemOut);
    }
}

```

A.19 CarParkSystemTest.java

```

package org.carpark.service;

import org.carpark.model.CarPark;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Scanner;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class CarParkSystemTest {

    private CarParkSystem carParkSystem;

    @BeforeEach
    public void setUp() {
        carParkSystem = new CarParkSystem();
    }

    @Test
    void chooseCarPark_ValidInputType_ShouldReturnSelectedCarPark() {
        // Mock scanner to simulate user choosing "0"
        Scanner mockScanner = mock(Scanner.class);
        when(mockScanner.nextLine()).thenReturn("1"); //simulated user
input

        CarPark selectedCarPark = carParkSystem.chooseCarPark(mockScanner);
    }
}

```

```
        // Verify the expected car park was chosen
        assertNotNull(selectedCarPark);
        assertEquals("NCP - Orchard Street", selectedCarPark.getName());
    }

    @Test
    void chooseCarPark_InvalidInput_ShouldPromptAgain(){
        Scanner mockScanner = mock(Scanner.class);
        when(mockScanner.nextLine()).thenReturn("3", "0");

        CarPark selectedCarPark = carParkSystem.chooseCarPark(mockScanner);

        // Verify Scanner was called at least twice (for invalid then valid
input)
        verify(mockScanner, atLeast(2)).nextLine();
        assertEquals("NCP - The City Gates", selectedCarPark.getName());
    }
}
```

OOP Report

GRADEMARK REPORT

FINAL GRADE

76/100

GENERAL COMMENTS

Engagement with Literature (5 marks): 5/5

Range & credibility of sources 2/2

Correct in-text citations & IEEE formatting 2/2

Evidence of critical reading
1/1

~~~~~  
**Knowledge & Understanding (20 marks): 13/20**

Best practice analysis (e.g. OO principles) 4/6

Static vs dynamic memory explanation 4/7

Data-structure justification  
5/7

~~~~~  
Cognitive & Intellectual Skills (20 marks): 13/20

Critical evaluation of designs & literature 7/10

Justification of hypotheses/decisions 3/5

Synthesis of complex issues 3/5

~~~~~  
**Practical Application Skills (50 marks): 42/50**

Class design & file organisation 8/10

Implementation of  
core simulation 8/ 10

Inheritance & IDReader subclasses 4/5

Dynamic data-structure coding 4/5

Use of OO concepts (abstract classes,  
polymorphism,

generics,

inheritance,

interfaces) 8/ 10

Memory-based storage structures 5/5

Unit tests & harness 5/5

~~~~~

Transferable Skills (5 marks):3/5

Report structure & clarity 1/2

Academic writing & organisation 0/1

User guide clarity 1/1

Video presentation quality
1/1

Comment:

Overall a smashing piece of work,
Great work J.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5



Comment 1

Avoiding the first person in a technical report helps to maintain an objective, impersonal tone that emphasises the work itself rather than the author. By writing “The algorithm was implemented” instead of “I implemented the algorithm,” the focus remains on the methodology and results, lending your discussion greater authority and universality.

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22



Comment 2

this goes a bit further beyond just Java!



Comment 3

An example of when to use a static memory structure would have been nice to see too

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59