

Formal Report: Best Practice Implementation for OOP Car Park Solution

Justin Hendy

May 2025

Abstract

The following report discusses best practices in object-oriented programming (OOP) through a literature review and the design and implementation of a car park simulator. It demonstrates how the application adheres to these best practices through code excerpts and a wider discussion of the applied principles, including descriptions of class attributes, their methods, their relationships with other classes and memory allocation choices in Java data structures. An overview of the unit test results highlights the application of a good and robust testing strategy. Finally, a brief user guide provides instructions on how to run and assess the code.

Contents

1. Abstract
2. Contents
3. Best Practice Summary
4. Application of Best Practices
5. Class Design Overview
6. Testing
7. User Guide
8. References

Best Practice Summary

This section outlines best practices in object-oriented programming, providing a brief description of each principle. The principles discussed include:

- Inheritance
- Interfaces
- Abstract Classes
- Polymorphism and Virtual Methods
- Storage Structures
- Generics

Inheritance as a principle provides a structure for new classes to inherit the properties or behaviours of an existing parent class, enabling code reuse and creating specialised class structures. Classes that inherit from a parent class extend the parent class. Best practices in inheritance include using inheritance only when there is a clear "is-a" relationship between classes, avoiding overly complex inheritance hierarchies, and favouring composition over inheritance when appropriate [1]. As an alternative to inheritance, composition allows for more flexible and maintainable designs by combining simple classes to create complex behaviour, best deployed when a clear "has-a" relationship exists between classes [2]. The "is-a" and "has-a" relationships are key to the successful use of inheritance and composition, demonstrated by examples such as a "Dog is-an Animal" or a "Computer has-a hard drive/graphics card/motherboard, etc...".

Interfaces provide contracts of behaviour to any class that wants to implement that behaviour. Well-designed interfaces are small and focused, provide no implementation details, and have descriptive names. Classes adhere to the contract by providing implementation details for all methods contained within the implemented interface; they cannot choose which methods to implement. Interfaces are best employed where a particular behaviour is inherent across multiple object types, such as actions like flying or walking [3]. Many classes derived from an Animal class will walk, but so would many from a Person class. Similarly, any object derived from an Aeroplane parent class would fly, but so would a Bird class derived from an Animal class.

Abstract classes provide a way to define a scaffolding or design structure for a concrete class that extends the abstract class [3]. Continuing the animal example, an animal is an abstract idea that becomes concrete once defined by further characteristics, such as the characteristics inherent to a cat or a dog. When adhering to best practice principles there should be no need to instantiate the abstract class within the code as any concrete class that extends an abstract class will override the required behaviours as intended.

Methods defined by abstract classes and interfaces are virtual methods. These methods can be overridden in derived classes, allowing for dynamic use of methods throughout the codebase. When employed successfully, virtual methods define common behaviour for related classes; however, where necessary, the final access modifier on certain classes should be used to prevent the unintended propagation of any virtual methods.

Polymorphism is the ability of various classes to provide specific behaviour to the same method call, achieved by method overriding. Best practices include using polymorphism to

simplify code through code reusability, avoiding type checks, and using interfaces or abstract classes to define common behaviour [4].

Storage structures such as lists, maps, and sets efficiently manage collections of objects. When selecting a storage structure, it is good practice to consider the purpose or use of the stored data, whether the structure uses static or dynamic memory allocation and the performance implications [5].

Generics allow code to be type-safe, reusable and flexible through parameterised types. When employed, classes, interfaces and methods can accept and use objects of various types but continue to do so in a type-safe manner. Type safety prevents errors at compile time resulting from type incompatibility. To ensure the best performance when working with generics, the constructed generic class must be as abstract as possible [6]. When used, Java already contains many generic classes that can significantly improve the application's ability to adapt to implementation changes.

Application of Best Practices

The following section discusses how the car park simulator application applies the best practices detailed previously. The application simulates a car park with various components, including barriers, scanners, ticket production, and a data store. The design employs inheritance, interfaces, abstract classes, polymorphism, and generics to create a flexible and maintainable codebase.

Inheritance, Abstract Classes and Composition

The primary use of inheritance within the application involves an abstract Barrier class extended by the two concrete classes, entranceBarrier and exitBarrier. The abstract class defines the common behaviour of both classes, which is to open and close the barrier. The concrete classes implement the specific behaviour of each barrier type.

```
1 public abstract class Barrier {
2
3     private boolean barrierOpen;
4
5     public Barrier(boolean barrierOpen) {
6         this.barrierOpen = barrierOpen;
7     }
8
9     public boolean isBarrierOpen() {
10         return barrierOpen;
11     }
12
13     public void setBarrierOpen(boolean barrierOpen) {
14         this.barrierOpen = barrierOpen;
15     }
16
17     public abstract String open();
18
19     public abstract String close();
20 }
```

The relationship between the abstract barrier and concrete entranceBarrier/exitBarrier is a clear case of an "is-a" relationship. The abstract class provides a structure for both classes, allowing for code reuse and reducing duplication. Below is an example of the exitBarrier class, which extends the abstract Barrier class with the open and close methods overridden with the exit implementation. Note in the image above, the use of a constructor on the abstract class, which is called by the concrete classes to set the initial state of the barrier via a super() call.

```
1 public ExitBarrier(boolean barrierOpen) {
2     super(barrierOpen);
3 }
4
5 @Override
6 public String open() {
7     if (isBarrierOpen()) {
8         return "Barrier is already open";
9     } else {
10        setBarrierOpen(true);
11        return "*".repeat(13) + "Exit Barrier is open" + "*".repeat(13);
12    }
13 }
14
15 @Override
16 public String close() {
17     if (!isBarrierOpen()) {
18         return "Barrier is already closed";
19     } else {
20        setBarrierOpen(false);
21        return "*".repeat(12) + "Exit barrier is closed" + "*".repeat(12);
22    }
23 }
```

The CompositionRoot class employs composition principles as the class instantiates all the dependencies for the application and injects them into a series of factories that encapsulate the object creation, in the particular case of the CarParkComponentsFactory, which aggregates all the key car park components, such as the entrance and exit barriers, the car park itself, etc. The CarParkComponentsFactory is a clear "has-a" relationship, as this class contains all the components required to build the car park. Here is one of the methods taken from the CompositionRoot class used for instantiating the CarParkComponentsFactory.

```
1 public class CompositionRoot {
2
3     public static CarParkComponentsFactory createCarParkComponentsFactory(int
4 carParkCapacity) {
5         CarPark carPark = new CarPark(carParkCapacity);
6         EntranceBarrier entranceBarrier = new EntranceBarrier(false, new Random
7 ());
8         ExitBarrier exitBarrier = new ExitBarrier(false);
9         ANPRReader anprReader = new ANPRReader();
10        BarcodeReader barcodeReader = new BarcodeReader();
11        FullSign fullSign = new FullSign(false);
12        DurationOfStay durationOfStay = new DurationOfStay();
13        return new CarParkComponentsFactory(carPark, entranceBarrier,
14 exitBarrier, anprReader, barcodeReader, fullSign, durationOfStay);
15    }
16
17    // more methods for other factories...
```

```
15 }
```

Interfaces

Interfaces are employed extensively throughout the application. At one end, two simple interfaces, `IScannable` and `IDetectable`, contain one method each. The `EntranceBarrier` implements `IDetectable`, and the `BarcodeReader` and `ANPRReader` implement `IScannable`. Interfaces below:

```
1 public interface IScannable {
2
3     boolean scan(String string);
4 }
5
6 public interface IDetectable {
7
8     boolean vehiclePresent();
9 }
```

Implementation:

```
1 public final class EntranceBarrier extends Barrier implements IDetectable {
2
3     // rest of the class...
4     @Override
5     public boolean vehiclePresent() {
6         if (getRandomBool()) {
7             System.out.println("""
8                 Vehicle detected at entrance barrier
9                 --> initialise ANPR system""");
10            return true;
11        } else {
12            System.out.println("No vehicle present at entrance barrier");
13            return false;
14        }
15    }
16 }
17
18
19 public class BarcodeReader implements IScannable {
20
21     @Override
22     public boolean scan(String barcode) {
23         return isValidBarcode(barcode);
24     }
25
26     private boolean isValidBarcode(String barcode) {
27         String BARCODE_REGEX = "~\\|\\d\\|\\|\\|\\d{5}\\|\\|\\|\\d{4}\\|\\|\\|\\d$";
28         return barcode.matches(BARCODE_REGEX);
29     }
30 }
31
32 public class ANPRReader implements IScannable {
33
34     @Override
35     public boolean scan(String numberPlate) {
36         if (isValidNumberPlate(numberPlate)) {
```

```

37         System.out.println("---> Valid UK number plate");
38         return true;
39     }
40     System.out.println("Not a valid UK number plate");
41     return false;
42 }
43
44 private boolean isValidNumberPlate(String numberPlate) {
45     String UK_PLATE_REGEX = "[A-Z]{2}[0-9]{2} [A-Z]{3}$";
46     return numberPlate.toUpperCase().matches(UK_PLATE_REGEX);
47 }
48
49 }

```

These examples demonstrate the use of interfaces to define a behaviour contract that any class wanting to execute that behaviour can implement.

The application contains many other interfaces, the largest and most significant being the `IUpdatable` interface. Implemented by the `CarParkUpdateService` and passed to the `CarParkUpdateSteps` controller via constructor-based dependency injection, the interface allows the service class to define all the logic required to update the car park on each simulation iteration. Here is the interface definition:

```

1     public boolean update() {
2         iUpdatable.reduceRemainingHours();
3         iUpdatable.handleQueue();
4         iUpdatable.handleCapacity();
5         iUpdatable.handleVehiclePresent();
6         iUpdatable.handleVehicle();
7         iUpdatable.handleScanner();
8         iUpdatable.handleDuration();
9         iUpdatable.handleParking();
10        iUpdatable.handleTicket();
11        iUpdatable.parkingStatus();
12        iUpdatable.setTotalVehiclesInCarPark();
13        iUpdatable.carParkStatus();
14        iUpdatable.exitCarPark();
15        iUpdatable.setVehicleToNull();
16        return iUpdatable.isVehicleQueueEmpty() && iUpdatable.isCarParkEmpty()
17    ;
18 }

```

The `IUpdatable` interface defines the contract for the `CarParkUpdateService` class, which implements all the methods. Using the interface as a constructor-based dependency injection for the `CarParkUpdateSteps` defines which methods from the `CarParkUpdateService` are available and allows for a clear separation of concerns, making it easy to test the controller without instantiating the service class.

Virtual Methods and Polymorphism

The above examples illustrate the use of both virtual methods and polymorphism. The open and close methods overridden by the concrete `Barrier` classes are virtual methods. The type of object used at runtime determines the specific implementation of the method. This implementation is also a clear example of polymorphism, as the same method call can result in different behaviours depending on the object used.

Storage Structures

Before looking at the specific storage structures used within the application, here is a quick overview of static and dynamic memory allocation. Static memory is fixed in size, allocated at compile time, and lasts for the entirety of the application's runtime. Simple data types and Arrays use static memory allocation. Dynamic memory size is allocated at runtime and can change during runtime. Larger data structures and complex objects such as ArrayLists, HashMaps, and LinkedLists use dynamic memory allocation.

The application uses a variety of storage structures, including LinkedLists, HashMaps, ArrayLists and the simple Array. Below is a summary of the key storage structures used in the application:

- **LinkedList:** Used to store the queue of vehicles waiting to enter the car park.
- **HashMap:** Used to store the vehicle data, including the vehicle's number plate and duration of stay.
- **ArrayList:** Used to store the list of vehicles currently parked in the car park.
- **Array:** Used to store an array of integer values used to represent the numbers of a barcode.

LinkedList

A LinkedList implementing the Java Queue collection was an obvious choice to represent a queue of vehicles waiting to enter the car park. On a basic level, the first in, first out (FIFO) principle mimics the behaviour of vehicles waiting to enter a car park; the LinkedList also provides a simple way to add and remove vehicles from the queue. The following code snippet shows how the LinkedList is used to add and remove vehicles from the queue:

```
1  public class VehicleQueue implements IParkable {
2
3      @Override
4      public void pollVehicleFromQueue() {
5          this.vehicleCurrentlyParking = getVehicleQueue().poll();
6      }
7
8      public void generateVehicles(Vehicle vehicle, List<String> numberPlates,
9      int numberOfVehicles) {
10
11          if (addVehicle(vehicle, numberPlates.removeFirst())) {
12              System.out.printf("Vehicle with number plate %s successfully
13      created%n", vehicle.getNumberPlate());
14          } else {
15              System.out.println("Vehicle failed to create");
16              System.out.println("Vehicle not added");
17          }
18
19          if (getVehicleQueueSize() == numberOfVehicles) {
20
21              System.out.printf("%d %s been created%n", getVehicleQueueSize(),
22              getVehicleQueueSize() == 1 ? "vehicle has" : "vehicles
23      have");
24
25              // Print a separator for better console output readability.
26              System.out.println("-".repeat(55));
```

```

23     }
24 }
25
26
27 private boolean addVehicle(Vehicle vehicle, String numberPlate) {
28     vehicle.setNumberPlate(numberPlate);
29     if (!findVehicle(vehicle)) {
30         getVehicleQueue().add(vehicle);
31         return true;
32     }
33     return false;
34 }
35
36 private boolean findVehicle(Vehicle vehicle) {
37     for (Vehicle ve : getVehicleQueue()) {
38         if (ve.getNumberPlate().equals(vehicle.getNumberPlate())) {
39             System.out.println("Vehicle already exists");
40             return true;
41         }
42     }
43     return false;
44 }

```

Several other methods are used within this vehicle queue class. The primary method is `generateVehicles`, which creates several vehicles and adds them to the queue in conjunction with the private methods `addVehicle` and `findVehicle`. The `pollVehicleFromQueue` method removes a vehicle from the queue when it is ready to enter the car park.

HashMap and ArrayList

The `CarParkVehicleDataStore` stores and manages the `HashMap` and `ArrayList` car park data. The purpose of this class is to hold all the data relating to the vehicle during its stay in the car park. Upon entering the car park, the `ArrayList` receives a vehicle number plate string, and the `HashMap` adds the same number plate, along with a duration of stay integer, as key-value pairs. On each iteration of the simulation, the duration of stay held in the map against each vehicle number plate is decremented by one. The use of the `ArrayList` allows for the implementation of a for-each loop to iterate through the vehicles in the car park, while the `HashMap` provides quick access to the duration of stay for each vehicle. The following code snippet shows the `HashMap` and `ArrayList` implementation:

```

1 public boolean addVehicle(String numberPlate, int durationOfStay) {
2     if (!findVehicle(numberPlate)) {
3         getVehiclesInCarPark().add(numberPlate);
4         getVehicleDurationMap().put(numberPlate, durationOfStay);
5         return true;
6     }
7     return false;
8 }
9
10
11 private boolean findVehicle(String numberPlate) {
12     for (String num : getVehiclesInCarPark()) {
13         if (num.equals(numberPlate)) {
14             System.out.println("Vehicle already parked in car park");
15             return true;

```

```

16     }
17     }
18     return false;
19 }
20
21
22 public void reduceRemainingHours() {
23     for (String num : getVehiclesInCarPark()) {
24         int currentRemainingHours = getVehicleDurationMap().get(num);
25         currentRemainingHours--;
26         getVehicleDurationMap().put(num, currentRemainingHours);
27     }
28 }

```

Array

A string of digits represents the barcode attached to a ticket for each vehicle, and the Array is used to store the individual numbers of the barcode. Using an Array allows for easy manipulation of the barcode, such as checking whether it is valid. The following code snippet shows how the Array is used to represent the barcode:

```

1  public String randomBarcode() {
2      Random rand = new Random();
3      int[] barcodeArray = new int[11];
4      for (int i = 0; i < 11; i++) {
5          int numbers = rand.nextInt(10);
6          barcodeArray[i] = numbers;
7      }
8
9      return ""
10         .formatted("%d|%d%d%d%d|%d%d%d|%d",
11         barcodeArray[0], barcodeArray[1], barcodeArray[2],
12         barcodeArray[3], barcodeArray[4], barcodeArray[5], barcodeArray[6],
13         barcodeArray[7], barcodeArray[8], barcodeArray[9], barcodeArray[10]);
14
15     }
16 }

```

This use of an array to hold individual digits allowed for creating a barcode string containing pipes to mimic the look of a barcode and randomly generated integers.

Class Design Overview

This section provides an overview of the class design, including a description of the key classes and their relationships. The class diagram in Appendix A provides a visual representation of the class design.

Main

- Contains three final constants that set the simulation's speed, the car park capacity, and the number of vehicles to generate.

- Calls the CompositionRoot class to instantiate the CarParkComponentsFactory, CarParkDataFactory and CarParkSimFactory.
- Instantiates the CarParkSimulation and calls the run() method to start the simulation.

CompositionRoot

- Contains three static methods that instantiate CarParkComponentsFactory, CarParkDataFactory and the CarParkSimFactory called createCarParkComponentsFactory, createCarParkDataFactory and createCarParkSimFactory respectively.
- The createCarParkComponentsFactory takes an integer parameter for the car park capacity and instantiates all the components related to the "physical" car park.
- The createCarParkDataFactory instantiates the CarParkVehicleDataStore and the TicketDataStore.
- The createCarParkSimFactory completes a number of steps in order to instantiate a new CarParkSimFactory, these steps will be detailed under their own heading below.

CreateCarParkSimFactory

- The createCarParkSimFactory takes three parameters, first the number of vehicles, second the newly instantiated CarParkComponentsFactory, and third the newly instantiated CarParkDataFactory.
- It then instantiates a VehicleQueueGenerator passing an integer for the number of vehicles and three suppliers for a VehicleQueue, a NumberPlateGenerator, which also takes an integer for the number of vehicles, and a Vehicle. A supplier is a functional interface and provides a single get() method that allows any calling class to use the supplier object type.
- The createVehicleQueue method is called on the VehicleQueueGenerator, which generates a queue of vehicles and assigns this vehicle queue to the VehicleQueue within the createCarParkSimFactory method.
- A new instance of CarParkUpdateMessages is created and passed to the CarParkUpdateService, along with the newly generated CarParkComponentsFactory, CarParkDataFactory, and VehicleQueue.
- A new instance of CarParkUpdateSteps is created and passed the CarParkUpdateService.
- Finally, a new instance of CarParkSimFactory is created and returned, passing the CarParkUpdateSteps.

CarParkSimulation

- The constructor takes the instantiated CarParkSimFactory and an integer parameter that sets the simulation speed.

- Contains a while loop that runs until the simulation is complete.
- The CarParkSimFactory has a method called getCarParkUpdateSteps() that calls the update() method on the CarParkUpdateSteps controller. Each iteration of the while loop calls this method.
- When the simulation is complete, the getCarParkUpdateSteps() returns true, causing the while loop to exit.

Factories

- The CarParkComponentsFactory implements the IComponents interface and passes encapsulated relevant data regarding the physical parts of the car park, such as the barriers, scanners, and the current space in the car park, to the CarParkUpdateService when required.
- The CarParkDataFactory implements the IData interface and passes relevant, encapsulated data to the CarParkUpdateService when required, including vehicle data, such as the vehicle number plate and duration of stay, and ticket data, including the barcode.
- The CarParkSimFactory only holds the instantiated CarParkUpdateSteps and calls the update() method on each simulation iteration, returning a boolean value.

VehicleQueue

- Implements the IParkable interface and contains methods for adding a vehicle to the queue.
- Contains a method called pollVehicleFromQueue() that removes a vehicle from the queue when it is ready to enter the car park and stores this object in a private instance field called vehicleCurrentlyParking.
- Contains a series of other methods that interact with the CarParkUpdateService that both set and provide data regarding the vehicle currently parking.

CarParkUpdateMessages

- Implements the IPrintable interface.
- Contains a series of string methods called and printed to the console by the CarParkUpdateService to provide feedback to the user on the current state of the car park and the vehicles within it.

CarParkUpdateService

- Contains the logic for updating the car park on each simulation iteration and implements the IUpdatable interface.

- The constructor takes four instantiated classes as parameters that each implement one of the following interfaces: IComponents, IData, IPrintable, and IParkable. The methods provided via the implemented interfaces provide access the relevant data to carry out specific business logic, which triggers the relevant strings from the CarParkUpdateMessages.
- The CarParkUpdateSteps controller calls the methods in this class to update the car park during each simulation iteration.

CarParkUpdateSteps

- Contains the main loop of the simulation
- The constructor takes a method that implements the IUpdatable interface as a parameter.
- Contains a single update() method that triggers each IUpdatable method to update the car park in the correct order. It returns true if both the VehicleQueue and CarParkDataStore are empty, which triggers the end of the simulation.

Other Classes

- A collection of model classes representing the various components of the car park, including the barriers, scanners, vehicles and the car park itself, with instance fields containing relevant information regarding each instance.
- The generator classes generate data such as vehicle number plates to assign to each vehicle, barcodes to assign to each ticket, and a vehicle queue for generating the vehicle objects ready for use by the car park. The VehicleQueueGenerator is a Record type that automatically generates the getters, setters, and instance fields for the parameters passed via the constructor.
- The solitary enums class defines the different allowable durations of stay.

Vehicle Journey

Once the VehicleQueue generates and the simulation begins, a random boolean from the EntranceBarrier class determines whether a vehicle is present at the entrance barrier. If a vehicle is present, a series of updates are triggered to allow that vehicle to enter the car park. Each iteration after this point will deduct one hour from the remaining hours for the vehicle until they reach zero; at this point, the vehicle leaves the car park, triggering methods that deal with the exit process.

Testing

Extensive unit testing has been carried out on the application using JUnit 5. The tests cover all the key classes and methods within the application. Main has no tests, and although the CompositionRoot class has tests, these are not included in the report as they are not relevant

to the overall functionality of the application. Appendix C contains the coverage results for all tests. There are 172 tests, and all branches, lines, and methods have 100% coverage.

The Mockito mocking framework isolates each class during testing. Appendix C includes a selection of results highlighting full coverage on the tested unit but zero coverage on all other units.

The tests run using an IDE's built-in test runner or the following command from the command line:

- `mvn clean test`

User Guide

- Once you have opened the project in your IDE, navigate to the `src/main/java/org/main` directory.
- Open the `Main.java` file.
- If you wish to change them, set the variables for the speed of the simulation, the car park capacity and the number of vehicles to be simulated.
- Run the `Main` class to start the simulation.
- Observe the console output for the simulation progress and results.
- The simulation will run until the vehicle queue is empty and all vehicles have exited the car park.

Alternatively you can run the application from the command line by navigating to the target directory and running the following command:

- `cd src/main/java`
- `javac org/main/Main.java`
- `java org.main.Main`

See Appendix D for images of the console output.

Conclusion

The car park simulator application successfully demonstrates good object-oriented programming. Through the use of inheritance, abstract classes, and extensive use of interfaces, the application is both flexible and maintainable. Composition, aggregation, and dependency injection allow each unit to maintain clear separation and provide ample opportunity for rigorous testing. This is highlighted by the extensive testing carried out, which demonstrates that all key methods and classes are covered. The application's user-friendly design has a clear console output that provides feedback on the simulation's progress and results.

References

- [1] B. Meyer, “The many faces of inheritance: a taxonomy of taxonomy,” *Computer*, vol. 29, no. 5, pp. 105–108, May 1996.
- [2] O. L. Madsen, “Open issues in object-oriented programming—a scandinavian perspective,” *Software: Practice and Experience*, vol. 25, no. S4, pp. 3–43, Dec. 1995.
- [3] M.-A. Sicilia, “Strategies for teaching object-oriented concepts with java,” *Computer Science Education*, vol. 16, no. 1, pp. 1–18, Mar. 2006.
- [4] GeeksForGeeks. (2025, May) Polymorphism in java. [Online]. Available: <https://www.geeksforgeeks.org/polymorphism-in-java/>
- [5] D. Costa, A. Andrzejak, J. Seboek, and D. Lo, “Empirical study of usage and performance of java collections,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE 17. ACM, Apr. 2017.
- [6] P. Soha and N. Pataki, “Instantiation of java generics,” *Acta Cybernetica*, vol. 25, no. 4, pp. 897–908, Jan. 2022.

Appendix B - CompositionRoot Class Diagram

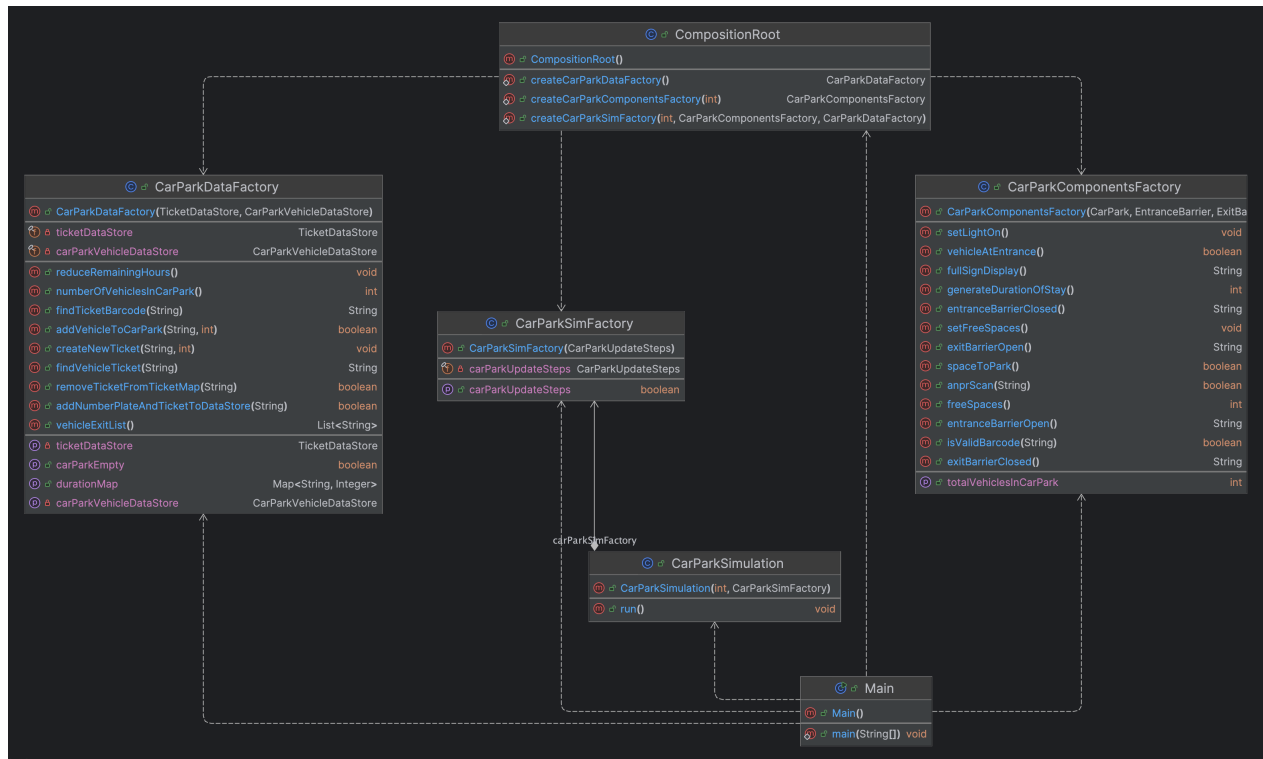


Figure 2: CompositionRoot

Appendix C - Testing

Coverage testpackage ×				
Element ^	Class, %	Method, %	Line, %	Branch, %
CarParkUpdateSteps	100% (1/1)	100% (2/2)	100% (17/17)	100% (2/2)
data	100% (2/2)	100% (18/18)	100% (59/59)	100% (24/24)
CarParkVehicleDataStore	100% (1/1)	100% (8/8)	100% (30/30)	100% (16/16)
TicketDataStore	100% (1/1)	100% (10/10)	100% (29/29)	100% (8/8)
enums	100% (1/1)	100% (3/3)	100% (9/9)	100% (6/6)
DurationOfStayEnums	100% (1/1)	100% (3/3)	100% (9/9)	100% (6/6)
factories	100% (3/3)	100% (31/31)	100% (41/41)	100% (2/2)
CarParkComponentsFactory	100% (1/1)	100% (15/15)	100% (22/22)	100% (2/2)
CarParkDataFactory	100% (1/1)	100% (14/14)	100% (16/16)	100% (0/0)
CarParkSimFactory	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
generators	100% (3/3)	100% (7/7)	100% (36/36)	100% (16/16)
BarcodeGenerator	100% (1/1)	100% (1/1)	100% (9/9)	100% (2/2)
NumberPlateGenerator	100% (1/1)	100% (4/4)	100% (20/20)	100% (12/12)
VehicleQueueGenerator	100% (1/1)	100% (2/2)	100% (7/7)	100% (2/2)
interfaces	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IComponents	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IData	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IDetectable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IParkable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IPrintable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IScannable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IUpdatable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
models	100% (9/9)	100% (38/38)	100% (76/76)	100% (18/18)
barriers	100% (3/3)	100% (11/11)	100% (29/29)	100% (10/10)
readers	100% (2/2)	100% (4/4)	100% (10/10)	100% (2/2)
CarPark	100% (1/1)	100% (6/6)	100% (9/9)	100% (0/0)
FullSign	100% (1/1)	100% (4/4)	100% (7/7)	100% (2/2)
Ticket	100% (1/1)	100% (8/8)	100% (10/10)	100% (0/0)
Vehicle	100% (1/1)	100% (5/5)	100% (11/11)	100% (4/4)
services	100% (1/1)	100% (21/21)	100% (59/59)	100% (32/32)
CarParkUpdateService	100% (1/1)	100% (21/21)	100% (59/59)	100% (32/32)
simulation	100% (3/3)	100% (16/16)	100% (41/41)	100% (18/18)
CarParkSimulation	100% (1/1)	100% (2/2)	100% (8/8)	100% (2/2)
DurationOfStay	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
VehicleQueue	100% (1/1)	100% (13/13)	100% (31/31)	100% (16/16)
util	100% (1/1)	100% (7/7)	100% (24/24)	100% (10/10)
CarParkUpdateMessages	100% (1/1)	100% (7/7)	100% (24/24)	100% (10/10)

Figure 3: Full Test Package

Coverage CarParkUpdateServiceTest				
Element	Class, %	Method, %	Line, %	Branch, %
org.main	3% (1/26)	14% (21/148)	15% (59/388)	25% (32/128)
composition	0% (0/1)	0% (0/4)	0% (0/18)	100% (0/0)
controllers	0% (0/1)	0% (0/2)	0% (0/17)	0% (0/2)
data	0% (0/2)	0% (0/18)	0% (0/59)	0% (0/24)
enums	0% (0/1)	0% (0/3)	0% (0/9)	0% (0/6)
factories	0% (0/3)	0% (0/31)	0% (0/41)	0% (0/2)
generators	0% (0/3)	0% (0/7)	0% (0/36)	0% (0/16)
models	0% (0/9)	0% (0/38)	0% (0/76)	0% (0/18)
services	100% (1/1)	100% (21/21)	100% (59/59)	100% (32/32)
CarParkUpdateService	100% (1/1)	100% (21/21)	100% (59/59)	100% (32/32)
simulation	0% (0/3)	0% (0/16)	0% (0/41)	0% (0/18)
util	0% (0/1)	0% (0/7)	0% (0/24)	0% (0/10)
Main	0% (0/1)	0% (0/1)	0% (0/8)	100% (0/0)

Figure 4: CarParkUpdateServiceTest Only

Coverage VehicleQueueTest				
Element	Class, %	Method, %	Line, %	Branch, %
org.main	3% (1/26)	8% (13/148)	7% (31/388)	12% (16/128)
composition	0% (0/1)	0% (0/4)	0% (0/18)	100% (0/0)
controllers	0% (0/1)	0% (0/2)	0% (0/17)	0% (0/2)
data	0% (0/2)	0% (0/18)	0% (0/59)	0% (0/24)
enums	0% (0/1)	0% (0/3)	0% (0/9)	0% (0/6)
factories	0% (0/3)	0% (0/31)	0% (0/41)	0% (0/2)
generators	0% (0/3)	0% (0/7)	0% (0/36)	0% (0/16)
models	0% (0/9)	0% (0/38)	0% (0/76)	0% (0/18)
services	0% (0/1)	0% (0/21)	0% (0/59)	0% (0/32)
simulation	33% (1/3)	81% (13/16)	75% (31/41)	88% (16/18)
CarParkSimulation	0% (0/1)	0% (0/2)	0% (0/8)	0% (0/2)
DurationOfStay	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
VehicleQueue	100% (1/1)	100% (13/13)	100% (31/31)	100% (16/16)
util	0% (0/1)	0% (0/7)	0% (0/24)	0% (0/10)
Main	0% (0/1)	0% (0/1)	0% (0/8)	100% (0/0)

Figure 5: VehicleQueueTest Only

Appendix D - Console Output

```
/Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java -java
Vehicle with number plate NH27 UBT successfully created
Vehicle with number plate YV43 DYL successfully created
Vehicle with number plate SF03 RGS successfully created
Vehicle with number plate AY10 HHA successfully created
Vehicle with number plate YC25 DVY successfully created
Vehicle with number plate LE14 AKW successfully created
Vehicle with number plate UP13 KWS successfully created
Vehicle with number plate WX41 JTB successfully created
Vehicle with number plate MD62 YPF successfully created
Vehicle with number plate UV23 RYX successfully created
Vehicle with number plate RT62 XDN successfully created
Vehicle with number plate XU12 EMH successfully created
Vehicle with number plate TA64 KLJ successfully created
Vehicle with number plate KH86 BFS successfully created
Vehicle with number plate AR55 WYB successfully created
Vehicle with number plate HY65 KUW successfully created
Vehicle with number plate TW85 ATH successfully created
Vehicle with number plate XE02 YMN successfully created
Vehicle with number plate PJ73 CDE successfully created
Vehicle with number plate TX10 SXJ successfully created
Vehicle with number plate AP60 NUJ successfully created
Vehicle with number plate YN70 MFB successfully created
Vehicle with number plate FX46 XTT successfully created
Vehicle with number plate KA46 YXM successfully created
Vehicle with number plate FS49 NFS successfully created
Vehicle with number plate VP67 DLA successfully created
Vehicle with number plate RY47 AJG successfully created
Vehicle with number plate XU44 AEV successfully created
Vehicle with number plate YV94 MRN successfully created
Vehicle with number plate EK44 BVM successfully created
30 vehicles have been created

-----
No vehicle present at entrance barrier

-----
Spaces available: 10
=====Car Park Status=====
Number Plate:      | Remaining Hours:
-----
No vehicle present at entrance barrier
```

Figure 6: Console Output - Image One

```

C:\> python car_park.py
-----
Spaces available: 10
=====Car Park Status=====
Number Plate: | Remaining Hours:
-----
Vehicle detected at entrance barrier
--> initialise ANPR system
--> Valid UK number plate
*****Entrance Barrier is open*****
----> NH27 UBT has been added to the car park
----> Duration of stay = 4 hours
*****Entrance barrier is closed*****
-----
Spaces available: 9
=====Car Park Status=====
Number Plate: | Remaining Hours:
NH27 UBT      |         4
-----
No ticket to process
No vehicles are leaving
-----
Vehicle detected at entrance barrier
--> initialise ANPR system
--> Valid UK number plate
*****Entrance Barrier is open*****
----> YV43 DYL has been added to the car park
----> Duration of stay = 12 hours
*****Entrance barrier is closed*****
-----
Spaces available: 8
=====Car Park Status=====
Number Plate: | Remaining Hours:
NH27 UBT      |         3
YV43 DYL      |        12
-----
No ticket to process
No vehicles are leaving
-----
Vehicle detected at entrance barrier

```

Figure 7: Console Output - Image Two

```

No vehicle present at entrance barrier
-----
Spaces available: 6
=====Car Park Status=====
Number Plate: | Remaining Hours:
NH27 UBT      |         0
SF03 RGS      |         0
YV43 DYL      |         9
AY10 HHA      |        23
-----
Ticket for SF03 RGS processed successfully
NH27 UBT is leaving the car park
-----
Receipt:
Length of stay    --> 4 hours
Rate Charged     --> @£0.50p/h
NH27 UBT         --> £2.00

Scan barcode to exit:
[9][35494][4445][7

--> Scanning receipt barcode
----> Valid receipt barcode
-----
*****Exit Barrier is open*****
----> NH27 UBT is exiting the car park
*****Exit barrier is closed*****
-----
SF03 RGS is leaving the car park
-----
Receipt:
Length of stay    --> 2 hours
Rate Charged     --> @£0.50p/h
SF03 RGS         --> £1.00

Scan barcode to exit:
[5][49477][7141][2

--> Scanning receipt barcode
----> Valid receipt barcode

```

Figure 8: Console Output - Image Three

```

No ticket to process
No vehicles are leaving
-----
Vehicle detected at entrance barrier
--> initialise ANPR system
--> Valid UK number plate
*****Entrance Barrier is open*****
-----> KA46 YXM has been added to the car park
-----> Duration of stay = 2 hours
*****Entrance barrier is closed*****
-----
Car park is full
Spaces available: 0
=====Car Park Status=====
Number Plate: | Remaining Hours:
AR55 WYB | 0
PJ73 CDE | 0
KA46 YXM | 2
TW85 ATH | 2
RT62 XDN | 4
AP60 NUJ | 5
FX46 XTT | 5
YN70 MFB | 6
TX10 SXJ | 8
XE02 YMN | 15
-----
Ticket for PJ73 CDE processed successfully
AR55 WYB is leaving the car park
-----
Receipt:
Length of stay --> 12 hours
Rate Charged --> @£0.50p/h
AR55 WYB --> £6.00

Scan barcode to exit:
|1|19846|18741|19

--> Scanning receipt barcode
-----> Valid receipt barcode

```

Figure 9: Console Output - Image Four

```

Ticket for EK44 BVM processed successfully
YV94 MRN is leaving the car park
-----
Receipt:
Length of stay --> 8 hours
Rate Charged --> @£0.50p/h
YV94 MRN --> £4.00

Scan barcode to exit:
|5|149697|11795|14

--> Scanning receipt barcode
-----> Valid receipt barcode
-----
*****Exit Barrier is open*****
-----> YV94 MRN is exiting the car park
*****Exit barrier is closed*****
-----
EK44 BVM is leaving the car park
-----
Receipt:
Length of stay --> 6 hours
Rate Charged --> @£0.50p/h
EK44 BVM --> £3.00

Scan barcode to exit:
|8|17904|19990|14

--> Scanning receipt barcode
-----> Valid receipt barcode
-----
*****Exit Barrier is open*****
-----> EK44 BVM is exiting the car park
*****Exit barrier is closed*****
-----

*****END OF SIMULATION*****

Process finished with exit code 0

```

Figure 10: Console Output - Image Five