

Best Practice Use in The Implementation of Object Oriented Solutions

Bradley Watkins

April 2025

Contents

1	Formal Report on Object-Oriented Programming Principles	3
1.1	Inheritance and Polymorphism	3
1.2	Interfaces and Abstract Classes	4
1.3	Generics and Storage Structures	5
1.4	Memory Allocation in Java	5
1.5	Conclusion	6
2	Implementation Analysis	6
2.1	Code Snippet 1: Inheritance and Abstract Classes	7
2.2	Code Snippet 2: Polymorphism	11
2.3	Code Snippet 3: Generics and Storage Structures	13
2.4	Code Snippet 4: Interfaces	16
3	Class Design and Relationship UML	18
4	Testing	19
A	Appendix: Additional Screenshots and User Guide	21
A.1	Youtube Link	21
A.2	User Guide	21
A.3	Running Solution	24

B	Appendix: Full Source Code	28
B.1	abstracts	28
B.2	classes	30
B.3	config	40
B.4	enums	43
B.5	exceptions	44
B.6	factories	44
B.7	interfaces	55
B.8	logger	57
B.9	runnable	60
B.10	org.main	72
B.11	TESTS	75
B.12	abstracts	75
B.13	classes	77
B.14	config	95
B.15	factories	99
B.16	runnable	115
B.17	stubs	128

Introduction

This report examines the application of core object-oriented programming (OOP) principles in the development of a car park application. The principles discussed include inheritance, polymorphism, interfaces, abstract classes, generics, and memory management.

The aim of this report is to demonstrate how these principles have been implemented in the application to create a solution that is efficient, scalable, and maintainable. Code snippets and examples from the application are provided to illustrate key design decisions and their alignment with best practices. Additionally, the report describes the relationships between the major classes in the application and explains how and why storage structures were chosen to support functionality.

1 Formal Report on Object-Oriented Programming Principles

1.1 Inheritance and Polymorphism

Inheritance in object-oriented programming enables related classes to share common implementation details. A base class can define shared properties and methods, which are then inherited by derived classes. Derived classes can provide their own implementations for methods in the base class by overriding those methods. They can also define new methods and properties that are not present in the base class. This reduces code duplication by allowing shared functionality to be reused across multiple classes. [1]

Inheritance support the open-closed principle, where classes are expected to be open for extension but closed for modification. Extending a class allows new functionality to be added while avoiding the need to modify the existing structure. This has the added benefit of keeping existing code working while allowing experimental changes to the newly derived class. These introductions and changes can also be tested separately, while existing tests will continue to function. [2] [1]

Polymorphism provides another beneficial use case for inheritance and further supports the open-closed principle. With polymorphism, any derived class can be used in place of classes higher in the inheritance hierarchy. This increases abstraction within a solution, as the least specific type definition can be used, allowing more specific behaviour to be swapped in and out. [2] During runtime, the correct method to call is determined by the type in use, allowing behaviour to change over time when different types are available. [3]

The polymorphism described above is known as dynamic polymorphism, where function overloading allows derived classes to provide more specific implementations. In languages like C++, methods must be explicitly

declared as virtual to enable polymorphism. A method marked as virtual can be overridden in derived classes, allowing them to provide more specific implementation. [4]

Java, on the other hand, treats all public and protected methods as virtual by default. To prevent a method from being overridden, it must be marked as final or static. Static methods are shared by all instances of a class and are not tied to a specific object, so they are not polymorphic. Classes can also be marked as final to stop them from being extended. [5]

There is also static polymorphism. Static polymorphism is achieved through method overloading, where methods share a common name but have different input parameters. This allows logically similar methods to be grouped, providing different implementations for specific scenarios. The compiler selects the correct method to use based on number and type of the provided parameters.

1.2 Interfaces and Abstract Classes

Interfaces group functionality that is not directly related to a class hierarchy. They allow for flexible designs by acting as connectors between different classes. An interface allows for the definition of method signatures without implementation. The behaviour for these methods must be provided by any class that implements the interface.

Abstract classes, in contrast, provide a common base for a group of related classes. They cannot themselves be instantiated but can be used as the type for polymorphic behaviour. They act as a base class in an inheritance hierarchy where there is shared behaviour but where more specific implementation is required. An abstract class may contain both abstract and concrete methods, with abstract methods requiring implementation in derived classes. [6] [3]

Abstract classes can be easier to update than interfaces. Adding a method to an interface requires providing implementation to all implementing classes. In an abstract class, when providing concrete methods, only the abstract class and any classes that need to override the default implementation need to be updated. This allows for greater flexibility in making changes to the class definition but also means that the class hierarchy must be carefully designed to avoid breaking changes. [6]

Both interfaces and abstract classes support dynamic polymorphism during runtime. The ability to group and store objects using these types is often critical. The decision to make use of one or the other will typically come down to the specific requirements of a given class and the level of shared responsibility between objects. If a group of unrelated objects must share method definitions, then interfaces are likely the correct choice. However, if default implementation and shared parameters are required for a group of related objects, then

abstract classes can provide greater flexibility.

1.3 Generics and Storage Structures

Storage structures are critical to the creation of useful object-oriented solutions. They allow for the grouping of objects during runtime and provide useful operations on the stored data. For example, dynamically sized structures like lists and sets can be used with lists providing out of the box sorting capabilities and sets ensuring uniqueness for stored data. There are also structures that provide rapid data access and placement, such as the first in, first out queues and key value pair storage through maps. Using Java as an example, many default storage structure implementations are built into the language. The Java Collections library has all of the above options, all providing good performance out of the box. [7]

What sets these storage structures apart in Java, making them flexible and generally useful, is the usage of generics. Generics allow for the definitions of a type parameter at the class level. When instantiating the class, a type is provided as this generic parameter, binding that object to usage of that type. That means that different instantiations of the same class can be made to work with many different types. This provides a level of type safety during compile time, with an exception being thrown if the wrong type is used after instantiation, while still providing flexibility. [8]

The reason generics are used heavily in storage structures is due to the stored type not impacting the operation of the structure. Using queues as an example, when adding an item to the back of the queue or removing an item from the front of the queue, it is not important what that type is. In most cases, no specific behaviour from the stored type will be invoked or used. What is important is that the correct type is stored and returned during operation, a problem generics gracefully solve.

1.4 Memory Allocation in Java

Memory allocation in Java is handled in both the stack and the heap. The heap is where Java handles dynamic memory allocation, creating and destroying objects as the program runs. Its size adjusts dynamically, growing and shrinking based on the program's needs. Objects in the heap are accessed through memory references, which act like pointers to the object's location in memory. This makes it efficient to share and pass objects around since only a copy of the reference is needed, not a copy of the object itself.

To manage unused objects and prevent memory leaks, Java uses garbage collection. Garbage collection typically involves two steps: the marking phase and the sweeping phase. In the marking phase, the garbage collector scans the heap and marks objects that are still in use. In the sweeping phase, objects that were not

marked are removed, freeing up memory. [9]

Garbage collection may also include further tuning, such as the usage of generational partitioning. With heap generations, new objects are placed in the nursery where a minor garbage collector will run frequently. This cleans up dead objects and moves long lived objects into an older generation. The older generational space will see less frequent but more thorough and resource-intensive garbage collection take place. [10] [9]

The stack is used for static memory allocation, where the size of the memory is fixed and determined at compile time. The stack is last in, first out, where method calls and local variables are placed as they are declared and automatically popped when they are no longer needed. This is a highly efficient memory management process where necessary memory is allocated and freed rapidly. The stack is generally small and isolated and is not suited to larger, longer lived objects. There is also the risk of a stack overflow occurring, where more memory is required than the stack has allocated. This can lead to crashes and unusual behaviour, so developers must be careful when making use of stack-intensive operations such as recursion.

The heap and stack provide an efficient system for memory management in Java. Fine tuning and modifying these processes can be time consuming and error prone. Thankfully, they are largely abstracted away from developers and only need attention in specialised cases, such as big data applications or high-performance systems. [10]

1.5 Conclusion

This report has looked at how key object-oriented programming principles help create scalable and maintainable software. Inheritance and polymorphism promote code reuse and abstraction, cutting down duplication and supporting the open-closed principle. Interfaces and abstract classes allow for modular and extensible designs, balancing shared functionality with custom behaviours. Generics provide type safety while making storage structures more generally useful. Java's memory model, with its heap and stack, ensures effective memory management through garbage collection and generational partitioning.

By applying these principles and tools, developers can create systems that are easier to extend, test, and maintain. The following sections illustrate how these principles have been used in the provided solution.

2 Implementation Analysis

This section focuses on showcasing specific examples from the Car Park application. The aim is to illustrate the implementation of object-oriented principles and adherence to best practices. Relevant code snippets and design

elements will be selected to demonstrate these concepts. Each example will be accompanied by a discussion explaining its alignment with best practices.

Comments and imports have been omitted from the code snippets to reduce clutter. See the full source code in the appendix for complete details.

Lombok annotations are used in the code snippets to reduce boilerplate code.

2.1 Code Snippet 1: Inheritance and Abstract Classes

There are multiple cases of inheritance in the solution. This section will focus on the base class **RunnableGate** and the derived classes **Entrance** and **Exit**.

The **RunnableGate** class is an abstract class that implements the **Runnable** interface. **RunnableGate** is not intended to be instantiated and used directly. It defines the common properties for gates in the car park, which are injected through the constructor. There are two methods available to the class, **arrive** and **run**.

arrive is a concrete method that is used to add vehicles to the queue, providing a default implementation to derived classes. **run** is an abstract method that must be implemented by the derived classes and is where the actual business logic for the gates is expected to be. The choice for **run** to be abstract in this case was due to there not being a default behaviour that would apply to all gates.

The derived classes **Entrance** and **Exit** extend the **RunnableGate** class and provide specific functionality within the **run** method. Entrances and exits are both gates, but they have different behaviours. An entrance handles objects entering the car park, polling the car park for availability and admitting vehicles where possible. Exits handle objects leaving the car park, releasing them when appropriate.

Despite their differences, both classes are fundamentally linked and can be logically grouped. The usage of the **RunnableGate** class allows for code reuse and a cleaner design, reducing duplicated code and improving maintainability. They also benefit from dynamic polymorphism during runtime, allowing the program to treat both classes as instances of the **RunnableGate**.

Listing 1: RunnableGate Abstract Class

```
@Getter
@RequiredArgsConstructor
public abstract class RunnableGate implements Runnable {
    protected final String name;
    protected final IDReader<Identifiable> idReader;
    protected final Sensor<Identifiable> sensor;
```

```

protected final Barrier barrier;

protected final Carpark<Vehicle> carpark;

protected final CarparkGateQueue<Vehicle> carparkGateQueue;

public void arrive(Vehicle arrival) {
    carparkGateQueue.enqueue(arrival);
}

public abstract void run();
}

```

Listing 2: Entrance Class, Extends RunnableGate

```

public class Entrance extends RunnableGate {

    public Entrance(String name, IDReader<Identifiable> idReader, Sensor<Identifiable> sensor, Barrier
        barrier, Carpark<Vehicle> carpark, CarparkGateQueue<Vehicle> carparkGateQueue) {
        super(name, idReader, sensor, barrier, carpark, carparkGateQueue);
    }

    @Override
    public void run() {
        do {
            if (sensor.carPresent(carparkGateQueue)) {
                tryAdmit();
            }

            sleep(POLL_INTERVAL);
        } while (KEEP_RUNNING);
    }

    private void tryAdmit() {
        if (carpark.spacesLeft() > 0) {
            try {

```

```

        String id = idReader.read(carparkGateQueue);
        barrier.raise();
        Vehicle item = carparkGateQueue.peek();

        if (item != null) {
            carpark.admit(id, item);
            // remove vehicle from gate queue after successful admittance
            carparkGateQueue.dequeue();
        }
    } catch (FailureToAdmitException e) {
        // Carpark is not ready for admittance; allow entrance to try again
    } catch (IllegalArgumentException e) {
        // Assume something is wrong with the item in the queue and remove it
        carparkGateQueue.dequeue();
    }

    barrier.lower();
}

protected void sleep(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

Listing 3: Exit Class, Extends RunnableGate

```

public class Exit extends RunnableGate {
    public Exit(String name, IDReader<Identifiable> idReader, Sensor<Identifiable> sensor, Barrier
        barrier, Carpark<Vehicle> carpark, CarparkGateQueue<Vehicle> carparkGateQueue) {

```

```

        super(name, idReader, sensor, barrier, carpark, carparkGateQueue);
    }

    @Override
    public void run() {
        do {
            if (sensor.carPresent(carparkGateQueue)) {
                tryRelease();
            }

            sleep(POLL_INTERVAL);
        } while (KEEP_RUNNING);
    }

    protected void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void tryRelease() {
        try {
            String id = idReader.read(carparkGateQueue);

            barrier.raise();
            carpark.release(id);
        } finally {
            // Always dequeue to handle invalid or improperly released vehicles
            carparkGateQueue.dequeue();
            barrier.lower();
        }
    }
}

```

}

2.2 Code Snippet 2: Polymorphism

Within the solution, both dynamic and static polymorphism are demonstrated. Dynamic polymorphism can be seen in the **CarParkOrchestrator**, which works with the **RunnableGate** class. The orchestrator is responsible for managing **Entrance** and **Exit** threads, and it uses the **RunnableGate** as a base to create a single list of both to iterate through. In this case the choice to keep the gates separated by their actual classes was made due to **Cars** needing to access the lists to select a random entrance and exit. If this behaviour were designed out, then all gates could be stored in a single list of **RunnableGate**. This could even be taken further, removing the need for specific orchestrators and instead having a list of **Runnable** where all threads are managed.

Static polymorphism can be seen in the **ConfigLoader** class, which uses method overloading to provide different ways to load configuration files. The method name **getInstance** is overloaded to allow for both a custom configuration path or for the default to be used if none is provided. This is a case where the ability to reuse the method name is useful, as both behaviours are fundamentally linked but are used in different contexts.

Listing 4: CarparkOrchestrator Class

```
@RequiredArgsConstructor
@Getter

public class CarParkOrchestrator implements Runnable {

    private final List<Entrance> entrances;

    private final List<Exit> exits;

    @Override
    public void run() {

        List<RunnableGate> gates = new ArrayList<>();

        gates.addAll(entrances);

        gates.addAll(exits);

        for (RunnableGate gate : gates) {

            Thread thread = new Thread(gate);

            thread.start();
```

```
    }  
  }  
}
```

Listing 5: ConfigLoader Class

```
public class ConfigLoader {  
    private final static String DEFAULT_CONFIG = "default.json";  
  
    @Getter  
    private final List<Config> configs;  
  
    private static ConfigLoader configLoader;  
  
    private ConfigLoader(String configPath) throws IOException {  
        if (configPath == null || configPath.isEmpty()) {  
            configPath = DEFAULT_CONFIG;  
        }  
  
        ObjectMapper mapper = new ObjectMapper();  
        configs = List.of(mapper.readValue(new File(configPath), Config[].class));  
    }  
  
    public static ConfigLoader getInstance() throws IOException {  
        return getInstance(DEFAULT_CONFIG);  
    }  
  
    public static ConfigLoader getInstance(String configPath) throws IOException {  
        if (configLoader == null) {  
            synchronized (ConfigLoader.class) {  
                if (configLoader == null) {  
                    configLoader = new ConfigLoader(configPath);  
                }  
            }  
        }  
    }  
}
```

```

    }

    return configLoader;
}

protected static void resetInstance() {
    configLoader = null;
}
}

```

2.3 Code Snippet 3: Generics and Storage Structures

Throughout the solution there are various instances of storage structures being used from within the Java Collections library (JCL). Each of these was selected with care to ensure scalability and maintainability. Below you will find various examples of the selected data structures and their justifications.

The data structure **Queue** was chosen for the storage of vehicles in the gates. This is due to its first in, first out approach to data management, which is ideal in this instance. When a vehicle arrives at the gate, it is added to the end of the queue. When a vehicle is released from the gate, it is removed from the front of the queue. This gives an average time complexity of $O(1)$ for both enqueue and dequeue operations.

A custom implementation of a queue was created to provide a common interface for gates to use. **CarparkGateQueue** makes use of the JCL queue, which is an implementation that makes use of linked lists. Instantiating the queue in the constructor allows the queue to be easily replaced with a different implementation if needed in the future, while abstracting the internal choice of data structure from the gates. The operations available to the queue are also restricted to those that are needed for the car park simulation in this case. By injecting the **CarparkGateQueue** as a dependency to other classes, mocks can be used for ease of testing.

A generic type parameter was used to allow for the queue to be used with any type of object. In this case, it may have been pertinent to use a more specific type, such as **Vehicle**, but this would have limited the flexibility of the queue. As the **RunnableGate** binds the type parameter as **Vehicle**, it did not feel necessary to update the gate queue. On reflection, if the solution were to be redone, many of the generics in use would be designed out, as they add a level of dependency management complexity that was unnecessary for an application of this scope.

Another example of a data structure used in the solution is the **ConcurrentHashMap**. This is used within

the **VehicleCarpark** class to manage vehicles currently in the car park. This is a key-value pair data structure, where the key is the vehicle ID and the value is the vehicle object itself. The `ConcurrentHashMap` is a thread-safe implementation of the `HashMap`, which allows for concurrent access to the data structure without the need for external synchronisation. This is ideal for the car park application, as multiple threads may be trying to access the car park at the same time.

The **VehicleCarpark** provides a limited interface into the internal `HashMap`, providing only the ability to add, remove and check remaining capacity. The admit and release (add and remove) operations are both $O(1)$ on average, as the `HashMap` uses a hash function to determine the location of the object in memory. The remaining capacity operation has a time complexity of $O(n)$ as it requires iterating through the entire map to count the number of objects. It is possible to remove the need for an $O(n)$ operation by tracking the number of vehicles as an external variable. However, this would be difficult to manage in a multithreaded environment and would be prone to lost updates without locking and synchronisation.

Listing 6: `CarparkGateQueue<E>` Class With Generic Type Parameter

```
public class CarparkGateQueue<E> {
    private Queue<E> queue;

    public CarparkGateQueue() {
        queue = new LinkedList<>();
    }

    public boolean isEmpty() {
        return queue.isEmpty();
    }

    public E peek() {
        return queue.peek();
    }

    public void enqueue(E element) {
        queue.add(element);
    }
}
```

```
public E dequeue() {  
    return queue.poll();  
}  
}
```

Listing 7: VehicleCarpark Class

@Getter

```
public class VehicleCarpark implements Carpark<Vehicle> {  
    protected int capacity;  
    protected ConcurrentHashMap<String, Vehicle> vehicles;  
    protected String name;  
  
    public VehicleCarpark(String name, int capacity) {  
        this.capacity = capacity;  
        this.name = name;  
        this.vehicles = new ConcurrentHashMap<>();  
    }  
}
```

@Override

```
public int spacesLeft() {  
    return capacity - vehicles.size();  
}
```

@Override

```
public synchronized void admit(String id, Vehicle vehicle) throws FailureToAdmitException {  
    if (vehicles.containsKey(id)) {  
        throw new IllegalArgumentException("Vehicle with id " + id + " already exists");  
    }  
  
    if (spacesLeft() == 0) {  
        throw new FailureToAdmitException("Maximum capacity achieved, failed to admit vehicle " +  
            id);  
    }  
}
```

```

        vehicles.put(id, vehicle);
        vehicle.setAdmitted(true);
    }

    @Override
    public void release(String id) {
        Vehicle vehicle = vehicles.remove(id);

        if (vehicle != null) {
            vehicle.setAdmitted(false);
        }
    }
}

```

2.4 Code Snippet 4: Interfaces

Interfaces are used liberally within the solution. A good example of interface usage is through the **IDReader** interface. This interface provides a generic type parameter and a single method, **read**.

The intention of this class is to take a queue of **Identifiable** objects and return the ID of the first object in the queue. This allows for the implementation of ID readers that work with different objects and different ID types. For example, within the solution there is a generic **IdentifiableIDReader** which does no ID verification. There is also a **NumberplateIDReader** which will throw an exception if the ID read is not in the correct format.

By making use of the IDReader interface, the required implementation can be swapped in and out as needed. In this case, a base implementation of IDReader and usage of inheritance in place of interfaces would probably have made sense. All of the IDReaders are logically related and work within a hierarchy of more and less specialised behaviour. As before, a change here would be worth considering if a redesign were to take place.

Listing 8: IDReader Interface With Bounded Generic Type Parameter

```

public interface IDReader<T extends Identifiable> {
    String read(CarparkGateQueue<? extends T> cars);
}

```

Listing 9: IdentifiableIDReader Class Implements IDReader Interface

```
public class IdentifiableIDReader implements IDReader<Identifiable> {

    @Override

    public String read(CarparkGateQueue<? extends Identifiable> cars) {
        String result = null;

        Identifiable item = cars.peek();
        if (item != null) {
            result = item.getId();
        }

        return result;
    }
}
```

Listing 10: NumberplateIDReader Class Implements IDReader Interface

```
public class NumberplateIDReader implements IDReader<Identifiable> {

    @Override

    public String read(CarparkGateQueue<? extends Identifiable> cars) {
        String result = null;

        Identifiable item = cars.peek();
        if (item != null) {
            result = item.getId();
            verify(result);
        }

        return result;
    }

    private void verify(String id) {
        Pattern pattern = Pattern.compile(Regex.NUMBERPLATE);
    }
}
```

```

    Matcher matcher = pattern.matcher(id);

    if (!matcher.matches()) {

        throw new IllegalArgumentException("Invalid ID: " + id);

    }

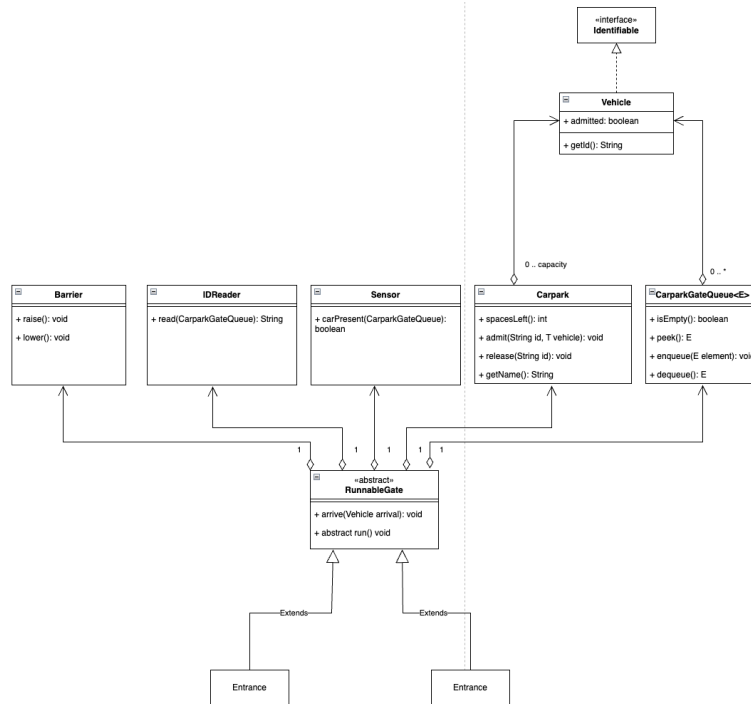
}
}

```

3 Class Design and Relationship UML

The below UML diagram shows the class design and relationships within the Car Park application. This is not a complete representation of the entire application, but it does show the key classes and how they are composed. The solution makes use of aggregation, with gates being provided all of the necessary components for smooth operation of the carpark. Vehicles are moved through the application via the **CarparkGateQueue** and **Carpark**. The **IDReader**, **Sensor**, and **Barrier** work to ensure that operation only occurs when relevant and that only valid vehicles are moved through the carpark.

Figure 1: Solution Class Relationship UML



4 Testing

The solution contains a comprehensive test suite that covers all aspects of the application. Most classes achieve 100% code coverage, with some exceptions. One of these exceptions is the **Main** class, which would require integration testing for full coverage. Due to the high test coverage throughout the application, there is a high level of confidence in the correct operation of the carpark.

Each class has been designed to be testable in isolation by leveraging dependency injection and inversion. By focusing on abstractions and passing dependencies into objects, behaviour can be mocked, and a focus can be placed on the class under test. This is especially important in a multithreaded environment, where the inability to control a behaviour can cause testing to be impossible in some cases.

Unit tests were written using JUnit 5 and Mockito and can be run through the terminal with Maven or through an IDE directly. The images below show the test coverage for the solution and the test results for a passing build.

Within the solution, 142 unit tests were produced, providing a total class coverage of 89% and a total method coverage of 86%. These numbers drop when looking at line and branch coverage, with 62% and 69% respectively. This can be attributed to classes such as the CarparkLogger and Main receiving no unit tests due to not containing testable business logic and having high line counts. Some branches were also impractical to test, such as methods that have been overwritten to remove some functionality for testing or where a threading exception is thrown. Where this is the case, care was taken to ensure that there were not edge cases where missing tests would mean losing confidence in the application.

Figure 2: Solution Test Results

```
[INFO] Results:
[INFO]
[INFO] Tests run: 142, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.911 s
[INFO] Finished at: 2025-04-21T20:03:39+01:00
[INFO] -----
```

Figure 3: Solution Test Coverage

Element	Class, %	Method, %	Line, %	Branch, %
org.main	88% (25/28)	86% (59/68)	68% (190/272)	62% (55/88)
abstracts	100% (2/2)	100% (2/2)	100% (3/3)	100% (0/0)
RunnableGate	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
Vehicle	100% (1/1)	100% (1/1)	100% (2/2)	100% (0/0)
classes	100% (7/7)	100% (18/18)	100% (46/46)	100% (14/14)
BarcodeIDReader	100% (1/1)	100% (1/1)	100% (5/5)	100% (2/2)
Barrier	100% (1/1)	100% (3/3)	100% (4/4)	100% (0/0)
CarParkGateQueue	100% (1/1)	100% (5/5)	100% (6/6)	100% (0/0)
IdentifiableIDReader	100% (1/1)	100% (1/1)	100% (5/5)	100% (2/2)
IdentifiableSensor	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
NumberplateIDReader	100% (1/1)	100% (2/2)	100% (10/10)	100% (4/4)
VehicleCarPark	100% (1/1)	100% (5/5)	100% (15/15)	100% (6/6)
config	100% (1/1)	100% (4/4)	100% (13/13)	87% (7/8)
Config	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ConfigLoader	100% (1/1)	100% (4/4)	100% (13/13)	87% (7/8)
enums	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
IdentifierOption	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
exceptions	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
FailureToAdmitException	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
factories	100% (8/8)	100% (17/17)	100% (70/70)	100% (16/16)
CarFactory	100% (2/2)	100% (8/8)	100% (30/30)	100% (6/6)
CarOrchestratorFactory	100% (1/1)	100% (1/1)	100% (4/4)	100% (2/2)
CarParkOrchestratorFactory	100% (1/1)	100% (3/3)	100% (16/16)	100% (4/4)
EntranceFactory	100% (1/1)	100% (1/1)	100% (5/5)	100% (0/0)
ExitFactory	100% (1/1)	100% (1/1)	100% (5/5)	100% (0/0)
ReaderFactory	100% (2/2)	100% (3/3)	100% (10/10)	100% (4/4)
interfaces	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
CarPark	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Identifiable	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IDReader	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Sensor	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
logger	0% (0/1)	0% (0/5)	0% (0/45)	0% (0/22)
CarParkLogger	0% (0/1)	0% (0/5)	0% (0/45)	0% (0/22)
runnable	100% (5/5)	100% (15/15)	94% (53/56)	81% (18/22)
Car	100% (1/1)	100% (5/5)	93% (14/15)	83% (5/6)
CarOrchestrator	100% (1/1)	100% (1/1)	100% (3/3)	100% (2/2)
CarParkOrchestrator	100% (1/1)	100% (1/1)	100% (6/6)	100% (2/2)
Entrance	100% (1/1)	100% (4/4)	94% (17/18)	75% (6/8)
Exit	100% (1/1)	100% (4/4)	92% (13/14)	75% (3/4)
Main	0% (0/1)	0% (0/4)	0% (0/34)	0% (0/6)
Regex	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)

References

- [1] A. Taivalsaari, “On the notion of inheritance,” *ACM Computing Surveys*, vol. 28, no. 3, pp. 438–479, Sep. 1996.
- [2] R. Plösch, J. Bräuer, C. Körner, and M. Saft, “Measuring, assessing and improving software quality based on object-oriented design principles,” *Open Computer Science*, vol. 6, no. 1, pp. 187–207, Jan. 2016.
- [3] H. Ali, “Exploring polymorphism: Flexibility and code reusability in object-oriented programming,” *Passer Journal of Basic and Applied Sciences*, vol. 6, no. Special Issue, Jan. 2024.
- [4] cppreference. (2025) Virtual Function Specifier. Accessed on: 2025-04-19. [Online]. Available: <https://en.cppreference.com/w/cpp/language/virtual>
- [5] Oracle. (2024) Writing Final Classes and Methods. Accessed on: 2025-04-19. [Online]. Available: <https://docs.oracle.com/javase/tutorial/java/IandI/final.html>
- [6] D. NICOLAU, “Demitizarea interfețelor în programarea orientată obiect,” *Revista Română de Informatică și Automatică*, vol. 30, no. 4, pp. 73–84, Dec. 2020.

- [7] P. Patrik PERHÁČ and S. ŠIMOŇÁK, “Algorithms and data structure libraries for java,” *Acta Electrotechnica et Informatica*, vol. 20, no. 1, pp. 39–48, Apr. 2020.
- [8] P. Soha and N. Pataki, “Instantiation of java generics,” *Acta Cybernetica*, vol. 25, no. 4, pp. 897–908, Jan. 2022.
- [9] Oracle. (2016) Understanding Memory Management. Accessed on: 2025-04-21. [Online]. Available: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html
- [10] S. Sahin, W. Cao, Q. Zhang, and L. Liu, “Jvm configuration management and its performance impact for big data applications,” in *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE, Jun. 2016.

A Appendix: Additional Screenshots and User Guide

A.1 Youtube Link

<https://www.youtube.com/watch?v=CNMemJ0cWFg>

A.2 User Guide

Listing 11: README.md

```
# Java OOP Carpark

Java OOP Carpark is a multithreaded solution that simulates the operation of a carpark with entrances
and exit queues.

Cars manage their journey to an entrance and subsequent exit from the carpark after admittance.
The project makes use of object-oriented programming principles in Java and employs various tools and
libraries for development and testing.

## Features

- Multithreaded Simulation: Cars, entrances, and exits operate in a concurrent environment.
- Queue Management: Admittance and release of cars are handled through entrance and exit queues.
- Carpark Map: A logical representation of the carpark is maintained to track car locations.
```

- **Java OOP Principles**: Designed with object-oriented programming concepts for clarity and modularity.

Technologies Used

- **Java**
- **JUnit**
- **Mockito**
- **Maven**

Getting Started

Prerequisites

Ensure you have the following installed on your system:

1. **Java Development Kit (JDK)**: Version 21.
2. **Maven**: For build and dependency management.
3. **IntelliJ IDEA** (Optional)

Loading Project Dependencies

Using Maven (Command Line)

1. Open a terminal and navigate to the project's root directory.
2. Run the following command to download and install all required dependencies:

```
```bash
mvn clean install
```
```

Using IntelliJ IDEA

1. Open IntelliJ IDEA and import the project as a Maven project.
2. IntelliJ will automatically detect the ``pom.xml`` file and prompt you to load the Maven dependencies.
3. If not automatically loaded, open the Maven tool window (on the right-hand side), click on the refresh button, or right-click the project and select ``Reload Maven Project``.

Running the Application

Command Line

1. Navigate to the root directory ``oop-carpark/``
2. Compile the project:

```
```bash
mvn compile
```
```
3. Run the application using the following command:

```
```bash
mvn exec:java -Dexec.mainClass="org.main.Main"
```
```

IntelliJ IDEA

1. Open IntelliJ IDEA and load the project.
2. Locate ``Main.java`` class in the ``src/main/java/org.main/`` directory.
3. Right-click ``Main.java`` and select ``Run``.

Running Tests

Command Line

1. Run the test suite using Maven:

```
```bash
mvn test
```
```

IntelliJ IDEA

1. Right-click on the ``src/test/java`` directory and select ``Run``.

— — —

A.3 Running Solution

Figure 4: Solution Starting

[illegible]

Figure 5: Soltuon After Five Seconds

| | |
|----------|----------|
| ----- | |
| NCP | |
| ----- | |
| HF84 LVU | E092 BAL |
| ----- | ----- |
| QK78 JHU | IB00 LPI |
| ----- | ----- |
| ZU42 EYN | NA21 ARO |
| ----- | ----- |
| MX13 OUV | HC57 GNA |
| ----- | ----- |
| NH85 APR | AN40 RYZ |
| ----- | ----- |
| IG63 BJH | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | ///// |
| ----- | ----- |

Figure 6: Solution After Ten Seconds

| | |
|----------|----------|
| ----- | |
| NCP | |
| ----- | |
| E092 BAL | FK13 XIM |
| ----- | ----- |
| QK78 JHU | MX13 OUV |
| ----- | ----- |
| DD43 ZFE | IG63 BJH |
| ----- | ----- |
| BZ19 KYJ | IB00 LPI |
| ----- | ----- |
| ZU42 EYN | NA21 ARO |
| ----- | ----- |
| JZ24 BUT | PX34 XIW |
| ----- | ----- |
| AN40 RYZ | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | ///// |
| ----- | ----- |

Figure 7: Solution After Twenty Seconds

| | |
|----------|----------|
| ----- | |
| NCP | |
| ----- | |
| DX35 LWL | FK13 XIM |
| ----- | ----- |
| QK78 JHU | CR88 EDW |
| ----- | ----- |
| XE41 JWW | XX47 GEQ |
| ----- | ----- |
| EI30 QQY | RA76 JNK |
| ----- | ----- |
| UU71 YOX | IG63 BJH |
| ----- | ----- |
| FP20 FMZ | LL41 EJJ |
| ----- | ----- |
| PX34 XIW | UK68 ENL |
| ----- | ----- |
| UW07 UGH | KU82 TYW |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | EMPTY |
| ----- | ----- |
| EMPTY | ///// |
| ----- | ----- |

B Appendix: Full Source Code

Find below the full source code listing for the car park solution. Subsections will indicate folder structure.

B.1 abstracts

```
package org.main.abstracts;

import lombok.Getter;
import lombok.RequiredArgsConstructor;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

/**
 * The {@code RunnableGate} class is an abstract representation of a car park gate.
 * It implements the {@code Runnable} interface and provides a basic structure
 * for gates intended to be run in a Thread.
 *
 * <ul>
 * <li>{@code Vehicle}s may arrive at the gate</li>
 * <li>{@code Vehicle}s that are at the gate will be managed and admitted to the {@code Carpark}
 * in a logical manner</li>
 * </ul>
 */
@Getter
@RequiredArgsConstructor
public abstract class RunnableGate implements Runnable {
    protected final String name;
    protected final IDReader<Identifiable> idReader;
    protected final Sensor<Identifiable> sensor;
```

```

protected final Barrier barrier;
protected final Carpark<Vehicle> carpark;
protected final CarparkGateQueue<Vehicle> carparkGateQueue;

/**
 * Places arrival on the back of the gate queue
 *
 * @param arrival
 */
public void arrive(Vehicle arrival) {
    carparkGateQueue.enqueue(arrival);
}

/**
 * Poll for items in the gate queue and perform necessary logic
 */
public abstract void run();
}

```

```

package org.main.abstracts;

import lombok.Getter;
import lombok.Setter;
import org.main.interfaces.Identifiable;

/**
 * The {@code Vehicle} class is an abstract representation of a vehicle.
 * It implements the {@code Identifiable} interface and provides a basic structure
 * for vehicles with an admitted status.
 *
 * <ul>
 * <li>Vehicles track whether they have been conceptually admitted</li>
 * <li>Vehicle are identifiable by an Id</li>

```

```

* </ul>
*/
@Getter
@Setter
public abstract class Vehicle implements Identifiable {
    protected boolean admitted;

    public Vehicle() {
        this.admitted = false;
    }

    @Override
    public abstract String getId();
}

```

B.2 classes

```

package org.main.classes;

import org.main.interfaces.Identifiable;

import java.util.UUID;

/**
 * A reader that extracts IDs in barcode format (UUID) from the carpark gate queue.
 * <p>
 * This class is designed to read and validate IDs in the form of UUIDs
 * </p>
 */
public class BarcodeIDReader extends IdentifiableIDReader {
    /**
     * Attempts to read the Id from the first item in the carpark gate queue
     * without removing the item

```

```

*
* @param cars
* @throws IllegalArgumentException when id is not in barcode (UUID) format
* @return null if no car in gate queue, String Id if car is present in queue with valid Id
*/
@Override
public String read(CarparkGateQueue<? extends Identifiable> cars) {
    String result = null;

    Identifiable item = cars.peek();

    if (item != null) {
        // UUID from string will fail if not in UUID format
        result = String.valueOf(UUID.fromString(item.getId()));
    }

    return result;
}
}

```

```

package org.main.classes;

import lombok.Getter;

/**
 * Represents a barrier used in a gate system, which can be raised or lowered.
 *
 * The barrier starts in a lowered position by default and can be raised or lowered
 * using the provided methods. This class is useful for controlling access in systems
 * like car parks or gated areas.
 */
@Getter
public class Barrier {
    private boolean up;

```

```

public Barrier() {
    up = false;
}

public void raise() {
    up = true;
}

public void lower() {
    up = false;
}
}

```

```

package org.main.classes;

import java.util.LinkedList;
import java.util.Queue;

/**
 * A simple queue implementation for use in car park simulations.
 * <p>
 * This class provides basic queue operations such as adding,
 * removing, and checking the first item in the queue. It is
 * designed to simulate the behavior of a queue at car park gates.
 * </p>
 * @param <E> The type of elements stored in the queue.
 */
public class CarparkGateQueue<E> {
    private Queue<E> queue;

    public CarparkGateQueue() {
        queue = new LinkedList<>();
    }
}

```

```

}

/**
 * Checks if the queue is empty.
 *
 * @return {@code true} if the queue has no elements, {@code false} otherwise.
 */
public boolean isEmpty() {
    return queue.isEmpty();
}

/**
 * Retrieves, but does not remove, the first element in the queue.
 *
 * @return The first element in the queue, or {@code null} if the queue is empty.
 */
public E peek() {
    return queue.peek();
}

/**
 * Adds a new element to the end of the queue.
 *
 * @param element The element to add to the queue.
 */
public void enqueue(E element) {
    queue.add(element);
}

/**
 * Removes and returns the first element from the queue.
 *
 * @return The first element in the queue, or {@code null} if the queue is empty.
 */

```

```

    public E dequeue() {
        return queue.poll();
    }
}

```

```

package org.main.classes;

import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;

/**
 * A basic ID reader for retrieving the ID objects that implement the {@link Identifiable} interface.
 * <p>
 * This class is designed to read the ID of the first object in a car park gate queue
 * without removing the object from the queue.
 * </p>
 */
public class IdentifiableIDReader implements IDReader<Identifiable> {

    /**
     * Reads the first item in the queue and retrieves its ID.
     * If the queue is empty, the method returns {@code null}.
     *
     * @param cars The car park gate queue containing objects that implement {@link Identifiable}.
     * @return The ID of the first item in the queue as a {@code String}, or {@code null}
     *         if the queue is empty.
     */
    @Override
    public String read(CarparkGateQueue<? extends Identifiable> cars) {
        String result = null;

        Identifiable item = cars.peek();
        if (item != null) {

```

```

        result = item.getId();
    }

    return result;
}
}

```

```

package org.main.classes;

import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

/**
 * A basic sensor for checking presence of Identifiable objects in car park gate queue
 */
public class IdentifiableSensor implements Sensor<Identifiable> {

    /**
     * Check gate queue for car presence
     * @param items
     * @return true if car present, false otherwise
     */
    @Override
    public boolean carPresent(CarparkGateQueue<? extends Identifiable> items) {
        return !items.isEmpty();
    }
}

```

```

package org.main.classes;

import org.main.Regex;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;

```

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * A reader for extracting and validating IDs in number plate format.
 */
public class NumberplateIDReader implements IDReader<Identifiable> {

    /**
     * Reads the ID of the first item in the car park gate queue.
     * <p>
     * This method retrieves the ID of the first car in the queue and verifies
     * that the ID matches the number plate format.
     * </p>
     *
     * @param cars The car park gate queue containing objects that implement {@link Identifiable}.
     * @return The ID of the first item in the queue as a {@code String}, or {@code null}
     *         if the queue is empty.
     * @throws IllegalArgumentException If the ID does not match the number plate format.
     */
    @Override
    public String read(CarParkGateQueue<? extends Identifiable> cars) {
        String result = null;

        Identifiable item = cars.peek();
        if (item != null) {
            result = item.getId();
            verify(result);
        }

        return result;
    }
}

```

```

/**
 * Verifies that the given ID matches the number plate format.
 *
 * <p>
 * This method checks if the provided ID conforms to the expected format
 * defined in {@link Regex#NUMBERPLATE}. If the ID does not match the format,
 * an {@code IllegalArgumentException} is thrown.
 *
 * </p>
 * @param id The ID to verify.
 * @throws IllegalArgumentException If the ID does not match the number plate format.
 */
private void verify(String id) {
    Pattern pattern = Pattern.compile(Regex.NUMBERPLATE);

    Matcher matcher = pattern.matcher(id);
    if (!matcher.matches()) {
        throw new IllegalArgumentException("Invalid ID: " + id);
    }
}
}

```

```

package org.main.classes;

import lombok.Getter;
import org.main.abstracts.Vehicle;
import org.main.exceptions.FailureToAdmitException;
import org.main.interfaces.Carpark;

import java.util.concurrent.ConcurrentHashMap;

/**
 * <p>A class for managing the storage of vehicles in a car park.</p>
 *
 * <p>This class provides functionality to admit and release vehicles in a car park,

```

```

    * while ensuring that the capacity and vehicle uniqueness constraints are respected.
    * It keeps track of all admitted vehicles using a thread-safe data structure.</p>
    */
    @Getter
    public class VehicleCarpark implements Carpark<Vehicle> {
        protected int capacity;
        protected ConcurrentHashMap<String, Vehicle> vehicles;
        protected String name;

        /**
         * <p>Creates a new car park with the specified name and capacity.</p>
         *
         * @param name    <p>The name of the car park.</p>
         * @param capacity <p>The maximum number of vehicles the car park can hold.</p>
         */
        public VehicleCarpark(String name, int capacity) {
            this.capacity = capacity;
            this.name = name;
            this.vehicles = new ConcurrentHashMap<>();
        }

        @Override
        public int spacesLeft() {
            return capacity - vehicles.size();
        }

        @Override
        public String getName() {
            return name;
        }

        /**
         * <p>Attempts to admit a vehicle into the car park.</p>
         *

```

```

* <p>If successful, the vehicle is added to the car park and marked as admitted.
* Admission will fail if:</p>
* <ul>
*   <li>The car park is full.</li>
*   <li>A vehicle with the same ID already exists in the car park.</li>
* </ul>
*
* @param id    <p>The unique ID of the vehicle to admit.</p>
* @param vehicle <p>The vehicle object to admit.</p>
* @throws FailureToAdmitException <p>If the car park is at full capacity.</p>
* @throws IllegalArgumentException <p>If a vehicle with the same ID already exists.</p>
*/
@Override
public synchronized void admit(String id, Vehicle vehicle) throws FailureToAdmitException {
    if (vehicles.containsKey(id)) {
        throw new IllegalArgumentException("Vehicle with id " + id + " already exists");
    }

    if (spacesLeft() == 0) {
        throw new FailureToAdmitException("Maximum capacity achieved, failed to admit vehicle " +
            id);
    }

    vehicles.put(id, vehicle);
    vehicle.setAdmitted(true);
}

/**
* <p>Attempts to release a vehicle from the car park using its ID.</p>
*
* <p>If a vehicle with the provided ID exists, it is removed from the car park
* and marked as no longer admitted. If no vehicle with the ID exists, the method
* does nothing.</p>
*

```

```

    * @param id <p>The unique ID of the vehicle to release.</p>
    */
    @Override
    public void release(String id) {
        Vehicle vehicle = vehicles.remove(id);

        if (vehicle != null) {
            vehicle.setAdmitted(false);
        }
    }
}

```

B.3 config

```

package org.main.config;

import lombok.Data;
import org.main.enums.IdentifierOption;

@Data
public class Config {
    private String name;
    private int capacity;
    private int numberOfEntrances;
    private int numberOfExits;
    private int numberOfCars;
    private int logInterval;
    private IdentifierOption carIdentifier;
    private IdentifierOption gateIdentifier;
}

```

```

package org.main.config;

```

```

import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.Getter;

import java.io.File;
import java.io.IOException;
import java.util.List;

/**
 * <p>A singleton class for loading configuration data from a JSON file.</p>
 *
 * <p>Provides functionality to load and parse configuration data
 * into a list of {@link Config} objects. It uses a singleton pattern to ensure
 * that only one instance of the class is created and shared throughout the application.</p>
 *
 * <p>The default configuration file is named "default.json", but a custom configuration
 * file path can also be specified.</p>
 */
public class ConfigLoader {
    private final static String DEFAULT_CONFIG = "default.json";

    @Getter
    private final List<Config> configs;

    private static ConfigLoader configLoader;

    /**
     * <p>Returns the singleton instance of {@code ConfigLoader} using the default configuration
     * file.</p>
     *
     * <p>If the instance does not exist, it will be created and initialized with the default
     * configuration file.</p>
     *
     * @return <p>The singleton instance of {@code ConfigLoader}.</p>
     * @throws IOException <p>If the default configuration file cannot be read or parsed.</p>

```

```

*/
public static ConfigLoader getInstance() throws IOException {
    return getInstance(DEFAULT_CONFIG);
}

/**
 * <p>Private constructor to initialize the {@code ConfigLoader} with a configuration file
 * path.</p>
 *
 * <p>If the provided path is {@code null} or empty, the default configuration file will be
 * used.</p>
 *
 * @param configPath <p>The path to the configuration file.</p>
 * @throws IOException <p>If the configuration file cannot be read or parsed.</p>
 */
private ConfigLoader(String configPath) throws IOException {
    if (configPath == null || configPath.isEmpty()) {
        configPath = DEFAULT_CONFIG;
    }

    ObjectMapper mapper = new ObjectMapper();
    configs = List.of(mapper.readValue(new File(configPath), Config[].class));
}

/**
 * <p>Returns the singleton instance of {@code ConfigLoader} using a specified configuration
 * file.</p>
 *
 * <p>If the instance does not exist, it will be created and initialized with the specified
 * configuration file.</p>
 *
 * @param configPath <p>The path to the configuration file.</p>
 * @return <p>The singleton instance of {@code ConfigLoader}.</p>
 * @throws IOException <p>If the specified configuration file cannot be read or parsed.</p>

```

```

    */
    public static ConfigLoader getInstance(String configPath) throws IOException {
        if (configLoader == null) {
            synchronized (ConfigLoader.class) {
                if (configLoader == null) {
                    configLoader = new ConfigLoader(configPath);
                }
            }
        }

        return configLoader;
    }

    /**
     * <p>Resets the singleton instance of {@code ConfigLoader}.</p>
     *
     * <p>This method is primarily intended for testing purposes to allow reinitialization
     * of the singleton with a different configuration file.</p>
     */
    protected static void resetInstance() {
        configLoader = null;
    }
}

```

B.4 enums

```

package org.main.enums;

public enum IdentifierOption {
    ANY,
    NUMBERPLATE,
    BARCODE
}

```

B.5 exceptions

```
package org.main.exceptions;

public class FailureToAdmitException extends Exception {
    public FailureToAdmitException(String message) {
        super(message);
    }
}
```

B.6 factories

```
package org.main.factories;

import lombok.RequiredArgsConstructor;
import org.main.enums.IdentifierOption;
import org.main.runnable.Entrance;
import org.main.runnable.Car;
import org.main.runnable.CarParkOrchestrator;
import org.main.runnable.Exit;

import java.util.List;
import java.util.Random;
import java.util.UUID;

/**
 * <p>A factory class for creating {@link Car} objects.</p>
 *
 * <p>This class generates cars with a unique identifier, a randomly assigned entrance,
 * and a randomly assigned exit. The identifier format (e.g., number plate or barcode)
 * is determined by the {@link IdentifierOption} provided.</p>
 */
@RequiredArgsConstructor
public class CarFactory {
```

```

private final CarParkOrchestrator carParkOrchestrator;
private final IdentifierOption identifierOption;

/**
 * <p>Creates a new {@link Car} with a unique identifier, a random entrance,
 * and a random exit from the {@link CarParkOrchestrator}</p>
 *
 * @return <p>A newly created {@link Car} object.</p>
 */
public Car createCar() {
    Entrance entrance = getRandomEntrance();
    Exit exit = getRandomExit();
    String id = generateId();

    return new Car(id, entrance, exit);
}

/**
 * <p>Selects a random entrance from the available entrances in the car park orchestrator.</p>
 *
 * @return <p>A random {@link Entrance} object, or {@code null} if no entrances are available.</p>
 */
private Entrance getRandomEntrance() {
    Random random = new Random();

    List<Entrance> entrances = carParkOrchestrator.getEntrances();

    if (entrances.isEmpty()) {
        throw new IllegalArgumentException("No entrance available");
    }

    return entrances.get(random.nextInt(entrances.size()));
}

```

```

/**
 * <p>Selects a random exit from the available exits in the car park orchestrator.</p>
 *
 * @return <p>A random {@link Exit} object, or {@code null} if no exits are available.</p>
 */
private Exit getRandomExit() {
    Random random = new Random();

    List<Exit> exits = carParkOrchestrator.getExits();

    if (exits.isEmpty()) {
        throw new IllegalArgumentException("No entrance available");
    }

    return exits.get(random.nextInt(exits.size()));
}

/**
 * <p>Generates a unique identifier for a car based on the {@link IdentifierOption}.</p>
 *
 * @return <p>A unique identifier as a {@link String}.</p>
 */
private String generateId() {
    switch (identifierOption) {
        case NUMBERPLATE:
            return generateRandomNumberplate();
        case BARCODE:
            return UUID.randomUUID().toString();
        default:
            return UUID.randomUUID().toString().substring(0, 5);
    }
}

/**

```

```

* <p>Generates a random vehicle number plate in the format "AA11 CCC".</p>
*
* @return <p>A randomly generated number plate as a {@link String}.</p>
*/
private String generateRandomNumberplate() {
    Random random = new Random();

    // using string concat instead of String.value of to prevent
    // ascii values being summed and replaced
    return ""
        + getChar(random)
        + getChar(random)
        + getNumeric(random)
        + getNumeric(random)
        + " "
        + getChar(random)
        + getChar(random)
        + getChar(random);
}

/**
* <p>Generates a random uppercase character (A-Z).</p>
*
* @param random <p>The {@link Random} instance to use for generating the character.</p>
* @return <p>A random uppercase character.</p>
*/
private char getChar(Random random) {
    return (char) ('A' + random.nextInt(26));
}

/**
* <p>Generates a random single-digit number (0-9).</p>
*
* @param random <p>The {@link Random} instance to use for generating the number.</p>

```

```

    * @return <p>A random single-digit number.</p>
    */
    private int getNumeric(Random random) {
        return random.nextInt(10);
    }
}

```

```

package org.main.factories;

import lombok.RequiredArgsConstructor;
import org.main.runnable.Car;
import org.main.runnable.CarOrchestrator;

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

/**
 * <p>A factory class for creating {@link CarOrchestrator} instances.</p>
 *
 * <p>This class uses a {@link CarFactory} to generate a specified number of cars
 * and organises them into a {@link CarOrchestrator}.</p>
 */
@RequiredArgsConstructor
public class CarOrchestratorFactory {
    private final CarFactory carFactory;

    /**
     * <p>Creates a {@link CarOrchestrator} with a specified number of cars.</p>
     *
     * @param carCount <p>The number of cars to include in the orchestrator.</p>
     * @return <p>A new {@link CarOrchestrator} instance containing the specified number of cars.</p>
     */
    public CarOrchestrator create(int carCount) {

```

```

        List<Car> cars = new ArrayList<>();

        for (int i = 0; i < carCount; i++) {
            cars.add(carFactory.createCar());
        }

        return new CarOrchestrator(cars);
    }
}

```

```

package org.main.factories;

import lombok.RequiredArgsConstructor;
import org.main.abstracts.Vehicle;
import org.main.interfaces.Carpark;
import org.main.runnable.CarParkOrchestrator;
import org.main.runnable.Entrance;
import org.main.runnable.Exit;

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

/**
 * <p>A factory class for creating {@link CarParkOrchestrator} instances.</p>
 *
 * <p>This factory generates orchestrators for managing entrances and exits
 * for a specific car park. Entrances and exits are created based on the
 * specified number using the provided {@link EntranceFactory} and {@link ExitFactory}.</p>
 */
@RequiredArgsConstructor
public class CarparkOrchestratorFactory {
    private final Carpark<Vehicle> carpark;
    private final EntranceFactory entranceFactory;

```

```

private final ExitFactory exitFactory;

/**
 * <p>Creates a {@link CarParkOrchestrator} for the given car park.</p>
 *
 * @param numberOfEntrances <p>The number of entrances to create for the car park.</p>
 * @param numberOfExits <p>The number of exits to create for the car park.</p>
 * @return <p>A new {@link CarParkOrchestrator} instance configured with the specified entrances
 *         and exits.</p>
 */
public CarParkOrchestrator createCarparkOrchestrator(int numberOfEntrances, int numberOfExits) {
    System.out.println("Creating CarParkOrchestrator for " + carpark.getName() + " with: " +
        numberOfEntrances + " entrances and " + numberOfExits + " exits");
    List<Entrance> entrances = createEntrances(numberOfEntrances);
    List<Exit> exits = createExits(numberOfExits);

    return new CarParkOrchestrator(entrances, exits);
}

/**
 * <p>Creates a list of entrances for the car park.</p>
 *
 * @param numberOfEntrances <p>The number of entrances to create.</p>
 * @return <p>A list of {@link Entrance} objects.</p>
 */
private List<Entrance> createEntrances(int numberOfEntrances) {
    List<Entrance> entrances = new ArrayList<>();
    for (int i = 0; i < numberOfEntrances; i++) {
        String name = carpark.getName() + "Entrance" + i;

        Entrance entrance = entranceFactory.createEntrance(name, carpark);
        entrances.add(entrance);
    }
}

```

```

        return entrances;
    }

    /**
     * <p>Creates a list of exits for the car park.</p>
     *
     * @param numberOfExits <p>The number of exits to create.</p>
     * @return <p>A list of {@link Exit} objects.</p>
     */
    private List<Exit> createExits(int numberOfExits) {
        List<Exit> exits = new ArrayList<>();
        for (int i = 0; i < numberOfExits; i++) {
            String name = carpark.getName() + "Exit" + i;

            Exit exit = exitFactory.createExit(name, carpark);
            exits.add(exit);
        }

        return exits;
    }
}

```

```

package org.main.factories;

import lombok.RequiredArgsConstructor;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.classes.IdentifiableSensor;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

```

```

import org.main.runnable.Entrance;

/**
 * <p>A factory class for creating {@link Entrance} instances for a car park.</p>
 */
@RequiredArgsConstructor
public class EntranceFactory {
    private final ReaderFactory readerFactory;

    /**
     * <p>Creates a new {@link Entrance} for a car park.</p>
     *
     * <p>The entrance is configured with a name, an ID reader, a sensor, a barrier,
     * a car park reference, and a gate queue.</p>
     *
     * @param name <p>The name of the entrance (e.g., "MainEntrance").</p>
     * @param carpark <p>The car park that this entrance will manage.</p>
     * @return <p>A new {@link Entrance} instance.</p>
     */
    public Entrance createEntrance(String name, Carpark<Vehicle> carpark) {
        IDReader<Identifiable> idReader = readerFactory.create();
        Sensor<Identifiable> sensor = new IdentifiableSensor();
        Barrier barrier = new Barrier();
        CarparkGateQueue<Vehicle> queue = new CarparkGateQueue<>();

        return new Entrance(name, idReader, sensor, barrier, carpark, queue);
    }
}

package org.main.factories;

import lombok.RequiredArgsConstructor;
import org.main.abstracts.Vehicle;

```

```

import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.classes.IdentifiableSensor;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;
import org.main.runnable.Exit;

/**
 * <p>A factory class for creating {@link Exit} instances for a car park.</p>
 */
@RequiredArgsConstructor
public class ExitFactory {
    private final ReaderFactory readerFactory;

    /**
     * <p>Creates a new {@link Exit} for a car park.</p>
     *
     * <p>The exit is configured with a name, an ID reader, a sensor, a barrier,
     * a car park reference, and a gate queue.</p>
     *
     * @param name <p>The name of the exit (e.g., "MainExit").</p>
     * @param carpark <p>The car park that this exit will manage.</p>
     * @return <p>A new {@link Exit} instance.</p>
     */
    public Exit createExit(String name, Carpark<Vehicle> carpark) {
        IDReader<Identifiable> idReader = readerFactory.create();
        Sensor<Identifiable> sensor = new IdentifiableSensor();
        Barrier barrier = new Barrier();
        CarparkGateQueue<Vehicle> queue = new CarparkGateQueue<>();

        return new Exit(name, idReader, sensor, barrier, carpark, queue);
    }
}

```

```
}
```

```
package org.main.factories;

import org.main.classes.BarcodeIDReader;
import org.main.classes.IdentifiableIDReader;
import org.main.classes.NumberplateIDReader;
import org.main.enums.IdentifierOption;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;

/**
 * <p>A factory class for creating {@link IDReader} instances based on the {@link
 * IdentifierOption}</p>
 *
 * <p>This class provides flexibility to generate different types of ID readers,
 * such as readers for number plates, barcodes, or general identifiable objects,
 * depending on the specified {@link IdentifierOption}</p>
 */
public class ReaderFactory {
    private final IdentifierOption identifierOption;

    /**
     * <p>Constructs a {@link ReaderFactory} with the given {@link IdentifierOption}</p>
     *
     * <p>If the provided {@code identifierOption} is {@code null}, the default option
     * {@link IdentifierOption#ANY} is used.</p>
     *
     * @param identifierOption <p>The {@link IdentifierOption} that determines the type
     * of ID reader to create.</p>
     */
    public ReaderFactory(IdentifierOption identifierOption) {
        if (identifierOption == null) {
```

```

        identifierOption = IdentifierOption.ANY;
    }

    this.identifierOption = identifierOption;
}

/**
 * <p>Creates an {@link IDReader} instance based on the configured {@link IdentifierOption}</p>
 *
 * <p>The type of reader created depends on the {@link IdentifierOption}:
 * <ul>
 * <li>{@link IdentifierOption#NUMBERPLATE}: Creates a {@link NumberplateIDReader}</li>
 * <li>{@link IdentifierOption#BARCODE}: Creates a {@link BarcodeIDReader}</li>
 * <li>{@link IdentifierOption#ANY}: Creates a {@link IdentifiableIDReader}</li>
 * </ul>
 * </p>
 *
 * @return <p>An appropriate {@link IDReader} instance for the specified {@link
 *         IdentifierOption}</p>
 */
public IDReader<Identifiable> create() {
    IDReader<Identifiable> idReader;

    switch (identifierOption) {
        case NUMBERPLATE -> idReader = new NumberplateIDReader();
        case BARCODE -> idReader = new BarcodeIDReader();
        default -> idReader = new IdentifiableIDReader();
    }

    return idReader;
}
}

```

B.7 interfaces

```
package org.main.interfaces;

import org.main.exceptions.FailureToAdmitException;

public interface Carpark<T> {
    String getName();
    void admit(String id, T vehicle) throws FailureToAdmitException;
    void release(String id);
    int spacesLeft();
}
```

```
package org.main.interfaces;

public interface Identifiable {
    String getId();
}
```

```
package org.main.interfaces;

import org.main.classes.CarparkGateQueue;

public interface IDReader<T extends Identifiable> {
    String read(CarparkGateQueue<? extends T> cars);
}
```

```
package org.main.interfaces;

import org.main.classes.CarparkGateQueue;

public interface Sensor<T extends Identifiable> {
    boolean carPresent(CarparkGateQueue<? extends T> items);
}
```

B.8 logger

```
package org.main.logger;

import lombok.RequiredArgsConstructor;
import org.main.abstracts.Vehicle;
import org.main.classes.VehicleCarpark;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Threaded logger for writing carpark to console periodically
 */
@RequiredArgsConstructor
public class CarparkLogger implements Runnable {
    private final VehicleCarpark carpark;
    private final int interval;

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(interval);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }

            printCarpark();
        }
    }

    private void printCarpark() {
```

```

ConcurrentHashMap<String, Vehicle> vehicles = carpark.getVehicles();

int maxLength = vehicles.keySet().stream().mapToInt(String::length).max().orElse(10);
int columnWidth = maxLength + 2; // Adding padding

StringBuilder builder = new StringBuilder();
appendTopHorizontalBoundary(builder, columnWidth);
builder.append("|").append(center(carpark.getName(), columnWidth * 2 + 5)).append("|\\n");
appendTopHorizontalBoundary(builder, columnWidth);

int count = 0;
for (Map.Entry<String, Vehicle> entry : vehicles.entrySet()) {
    if(count % 2 == 0) {
        builder.append("|");
    }
    builder.append(center(entry.getKey(), columnWidth));

    if(count % 2 == 0) {
        builder.append(" ");
    }

    if (count % 2 == 1) {
        builder.append("|\\n");
        appendHorizontalBoundary(builder, columnWidth);
    }
    count++;
}

// Fill remaining rows with empty spaces
while (count < carpark.getCapacity()) {
    if(count % 2 == 0) {
        builder.append("|");
    }
    builder.append(center("EMPTY", columnWidth));
}

```

```

        if(count % 2 == 0) {
            builder.append(" ");
        }

        if (count % 2 == 1) {
            builder.append("\n");
            appendHorizontalBoundary(builder, columnWidth);
        }
        count++;
    }

    if (count % 2 != 0) {
        String lastSpace = "EMPTY";

        if (carpark.getCapacity() % 2 != 0 ) {
            // space is not available
            lastSpace = "/".repeat(columnWidth - 4);
        }

        builder.append(center(lastSpace, columnWidth)).append("\n");
        appendHorizontalBoundary(builder, columnWidth);
    }

    System.out.print(builder);
    System.out.println();
}

private String center(String text, int width) {
    int padding = (width - text.length()) / 2;
    if (padding > 0) {
        return " ".repeat(padding) + text + " ".repeat(width - text.length() - padding);
    } else {
        return text;
    }
}

```

```

    }

    private void appendTopHorizontalBoundary(StringBuilder builder, int columnWidth) {
        builder.append("|").append("-".repeat(columnWidth * 2 + 5)).append("|\\n");
    }

    private void appendHorizontalBoundary(StringBuilder builder, int width) {
        builder.append("|").append("-".repeat(width)).append("|
        |").append("-".repeat(width)).append("|\\n");
    }
}

```

B.9 runnable

```

package org.main.runnable;

import lombok.Getter;
import lombok.RequiredArgsConstructor;
import lombok.Setter;
import org.main.abstracts.Vehicle;
import org.main.interfaces.Identifiable;

import java.util.Random;

/**
 * <p>A runnable implementation of a {@link Vehicle} that tracks its journey
 * through a car park, including arriving at an entrance, waiting for admittance,
 * and departing through an exit.</p>
 *
 * <p>The {@code Car} class models the behavior of a vehicle as it interacts
 * with car park gates, simulating real-time delays using random sleep intervals.</p>
 */
@Getter

```

```

@RequiredArgsConstructor
public class Car extends Vehicle implements Identifiable, Runnable {

    private final String id;

    private final Entrance entrance;

    private final Exit exit;

    /**
     * <p>The upper bound for random sleep intervals, in milliseconds.</p>
     * <p>Used to simulate travel time to the car park gates.</p>
     */
    private int SLEEP_RANDOM_BOUND = 5000;

    /**
     * <p>A flag to control polling behavior while waiting for admittance.</p>
     * <p>If set to {true}, the car will stop polling for admittance.</p>
     */
    @Setter
    private boolean DO_NOT_POLL_IF_NOT_ADMITTED = false;

    /**
     * <p>Sets the upper bound for random sleep intervals.</p>
     *
     * <p>If the provided bound is less than 1, it will default to 1 to
     * ensure a valid sleep interval.</p>
     *
     * @param bound <p>The upper bound for random sleep intervals, in milliseconds.</p>
     */
    public void setSleepRandomBound(int bound) {
        this.SLEEP_RANDOM_BOUND = Math.max(bound, 1);
    }

    /**
     * <p>Executes the car's journey through the car park.</p>
     */

```

```

* <p>The journey involves the following steps:</p>
* <ul>
*   <li>Traveling to the entrance (simulated by a random sleep interval).</li>
*   <li>Arriving at the entrance and waiting for admittance.</li>
*   <li>Traveling through the car park (simulated by another random sleep interval).</li>
*   <li>Departing through the exit if admitted.</li>
* </ul>
*/
@Override
public void run() {
    // Journey to entrance
    sleepRandom();

    entrance.arrive(this);

    waitUntilAdmitted();

    // Time in car park
    sleepRandom();
    if (admitted) {
        exit.arrive(this);
    }
}

/**
 * <p>Simulates a random sleep interval to represent idle time.</p>
 */
private void sleepRandom() {
    Random random = new Random();

    int timeToArrival = random.nextInt(SLEEP_RANDOM_BOUND) + 1;
    sleep(timeToArrival);
}

/**

```

```

    * <p>Forces the thread to sleep for the specified duration.</p>
    *
    * @param millis <p>The duration to sleep, in milliseconds.</p>
    */
protected void sleep(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

/**
 * <p>Waits until the car is admitted to the car park.</p>
 *
 * <p>The car will repeatedly check its admission status, sleeping
 * for a short interval between checks. If the {@code DO_NOT_POLL_IF_NOT_ADMITTED}
 * flag is set to {@code true}, the polling will stop prematurely.</p>
 */
protected void waitUntilAdmitted() {
    while (!isAdmitted()) {
        sleep(20); // Poll for admitted check
        if (DO_NOT_POLL_IF_NOT_ADMITTED) {
            break;
        }
    }
}
}

package org.main.runnable;

import lombok.Getter;
import lombok.RequiredArgsConstructor;

```

```

import java.util.List;

/**
 * <p>Orchestrates the management and execution of multiple {@link Car} objects in a car park
 * simulation.</p>
 */
@RequiredArgsConstructor
@Getter
public class CarOrchestrator implements Runnable {
    private final List<Car> cars;

    /**
     * <p>Runs all stored {@link Car} objects in separate threads.</p>
     *
     * <p>Each car is executed on a new thread, allowing them to operate concurrently
     * as part of the car park simulation.</p>
     */
    @Override
    public void run() {
        for (Car car : cars) {
            Thread thread = new Thread(car);
            thread.start();
        }
    }
}

package org.main.runnable;

import lombok.Getter;
import lombok.RequiredArgsConstructor;
import org.main.abstracts.RunnableGate;
import org.main.abstracts.Vehicle;

```

```

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

/**
 * <p>Orchestrates the management and execution of car park gates in a simulation.</p>
 *
 * <p>The {@code CarParkOrchestrator} class manages a collection of {@link Entrance} and {@link Exit}
 * gates, allowing them to run concurrently in separate threads. This helps simulate the real-time
 * operations of a car park.</p>
 */
@RequiredArgsConstructor
@Getter
public class CarParkOrchestrator implements Runnable {
    private final List<Entrance> entrances;
    private final List<Exit> exits;

    /**
     * <p>Runs all stored gates ({@link Entrance} and {@link Exit}) in separate threads.</p>
     */
    @Override
    public void run() {
        List<RunnableGate> gates = new ArrayList<>();
        gates.addAll(entrances);
        gates.addAll(exits);

        for (RunnableGate gate : gates) {
            Thread thread = new Thread(gate);
            thread.start();
        }
    }
}

```

```

package org.main.runnable;

import lombok.Setter;
import org.main.abstracts.RunnableGate;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.exceptions.FailureToAdmitException;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

/**
 * <p>An implementation of {@link RunnableGate} that manages the logic for admitting vehicles
 * into a car park through an entrance gate.</p>
 *
 * <p>This class continuously monitors the gate queue for vehicles, checks if the car park
 * has available spaces, and attempts to admit vehicles using the provided {@link IDReader},
 * {@link Sensor}, and {@link Barrier}. It runs on its own thread to simulate real-time
 * gate operation.</p>
 */
public class Entrance extends RunnableGate {
    /**
     * <p>Flag to determine whether the gate should keep running. Set to {@code false}
     * to stop the gate's operation.</p>
     */
    @Setter
    private boolean KEEP_RUNNING = true;

    /**
     * <p>The interval, in milliseconds, at which the gate polls the queue for vehicles.</p>
     */
    @Setter

```

```

private int POLL_INTERVAL = 200;

/**
 * @param name          <p>The name of the entrance gate.</p>
 * @param idReader      <p>The {@link IDReader} used to read identifiers for vehicles.</p>
 * @param sensor        <p>The {@link Sensor} used to detect vehicle presence.</p>
 * @param barrier       <p>The {@link Barrier} controlling the physical gate mechanism.</p>
 * @param carpark       <p>The {@link Carpark} that the gate manages vehicle entry for.</p>
 * @param carparkGateQueue <p>The {@link CarparkGateQueue} representing the queue of vehicles at
 *                          the gate.</p>
 */
public Entrance(String name, IDReader<Identifiable> idReader, Sensor<Identifiable> sensor, Barrier
    barrier, Carpark<Vehicle> carpark, CarparkGateQueue<Vehicle> carparkGateQueue) {
    super(name, idReader, sensor, barrier, carpark, carparkGateQueue);
}

/**
 * <p>Continuously polls the gate queue for vehicles and attempts to admit them into the car
 * park.</p>
 */
@Override
public void run() {
    do {
        if (sensor.carPresent(carparkGateQueue)) {
            tryAdmit();
        }

        sleep(POLL_INTERVAL);
    } while (KEEP_RUNNING);
}

/**
 * <p>Attempts to admit a vehicle into the car park if spaces are available.</p>
 *

```

```

* <p>The method raises the {@link Barrier}, reads the vehicle's identifier using the {@link
    IDReader},
* and admits the vehicle into the {@link Carpark} if valid. </p>
*/
private void tryAdmit() {
    if (carpark.spacesLeft() > 0) {
        try {
            String id = idReader.read(carparkGateQueue);
            barrier.raise();
            Vehicle item = carparkGateQueue.peek();

            if (item != null) {
                carpark.admit(id, item);
                // remove vehicle from gate queue after successful admittance
                carparkGateQueue.dequeue();
            }
        } catch (FailureToAdmitException e) {
            // Carpark is not ready for admittance; allow entrance to try again
        } catch (IllegalArgumentException e) {
            // Assume something is wrong with the item in the queue and remove it
            carparkGateQueue.dequeue();
        }

        barrier.lower();
    }
}

/**
* <p>Forces the thread to sleep for the specified duration.</p>
*
* @param millis <p>The duration to sleep, in milliseconds.</p>
*/
protected void sleep(int millis) {
    try {

```

```

        Thread.sleep(millis);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

```

package org.main.runnable;

import lombok.Setter;
import org.main.abstracts.RunnableGate;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

/**
 * <p>An implementation of {@link RunnableGate} that manages the logic for releasing vehicles
 * from a car park through an exit gate.</p>
 *
 * <p>This class continuously monitors the gate queue for vehicles, checks for their presence,
 * and attempts to release them from the car park using the provided {@link IDReader},
 * {@link Sensor}, and {@link Barrier}. It runs on its own thread to simulate real-time
 * gate operation.</p>
 */
public class Exit extends RunnableGate {
    /**
     * <p>Flag to determine whether the gate should keep running. Set to {@code false}
     * to stop the gate's operation.</p>
     */

```

```

@Setter

private boolean KEEP_RUNNING = true;

/**
 * <p>The interval, in milliseconds, at which the gate polls the queue for vehicles.</p>
 */
@Setter

private int POLL_INTERVAL = 200;

/**
 * @param name          <p>The name of the exit gate.</p>
 * @param idReader      <p>The {@link IDReader} used to read identifiers for vehicles.</p>
 * @param sensor        <p>The {@link Sensor} used to detect vehicle presence.</p>
 * @param barrier       <p>The {@link Barrier} controlling the physical gate mechanism.</p>
 * @param carpark       <p>The {@link Carpark} that the gate manages vehicle exits for.</p>
 * @param carparkGateQueue <p>The {@link CarparkGateQueue} representing the queue of vehicles at
 *                          the gate.</p>
 */
public Exit(String name, IDReader<Identifiable> idReader, Sensor<Identifiable> sensor, Barrier
    barrier, Carpark<Vehicle> carpark, CarparkGateQueue<Vehicle> carparkGateQueue) {
    super(name, idReader, sensor, barrier, carpark, carparkGateQueue);
}

/**
 * <p>Continuously polls the gate queue for vehicles and attempts to release them from the car
 * park.</p>
 */
@Override

public void run() {
    do {
        if (sensor.carPresent(carparkGateQueue)) {
            tryRelease();
        }
    }
}

```

```

        sleep(POLL_INTERVAL);
    } while (KEEP_RUNNING);
}

/**
 * <p>Forces the thread to sleep for the specified duration.</p>
 *
 * @param millis <p>The duration to sleep, in milliseconds.</p>
 */
protected void sleep(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

/**
 * <p>Attempts to release a vehicle from the car park.</p>
 *
 * <p>The method raises the {@link Barrier} to allow the vehicle to exit,
 * reads the vehicle's identifier using the {@link IDReader}, and updates the
 * {@link Carpark} to reflect the vehicle's release. Finally, it removes the vehicle
 * from the {@link CarparkGateQueue}</p>
 */
private void tryRelease() {
    try {
        String id = idReader.read(carparkGateQueue);

        barrier.raise();
        carpark.release(id);
    } finally {
        // Always dequeue to handle invalid or improperly released vehicles
        carparkGateQueue.dequeue();
    }
}

```

```
        barrier.lower();
    }
}
}
```

B.10 org.main

```
package org.main;

import org.main.abstracts.Vehicle;
import org.main.config.Config;
import org.main.config.ConfigLoader;
import org.main.logger.CarparkLogger;
import org.main.classes.VehicleCarpark;
import org.main.factories.CarFactory;
import org.main.factories.CarOrchestratorFactory;
import org.main.factories.CarparkOrchestratorFactory;
import org.main.factories.EntranceFactory;
import org.main.factories.ExitFactory;
import org.main.factories.ReaderFactory;
import org.main.interfaces.Carpark;
import org.main.runnable.CarOrchestrator;
import org.main.runnable.CarParkOrchestrator;

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

public class Main {
    public static final String filePath = null;
    public static void main(String[] args) {
        List<Config> configs;
        try {
            ConfigLoader loader;
```

```

    if (filePath == null) {
        loader = ConfigLoader.getInstance();
    } else {
        loader = ConfigLoader.getInstance(filePath);
    }

    configs = loader.getConfigs();
} catch (Exception e) {
    System.err.println("Failed to load configuration: " + e.getMessage());
    System.exit(1);
    return;
}

List<Thread> threads = new ArrayList<>();

for (Config config : configs) {
    try {
        VehicleCarpark carpark = new VehicleCarpark(config.getName(), config.getCapacity());
        CarparkLogger logger = new CarparkLogger(carpark, config.getLogInterval());

        CarParkOrchestrator carParkOrchestrator = buildCarParkOrchestrator(carpark, config);
        CarOrchestrator carOrchestrator = buildCarOrchestrator(carParkOrchestrator, config);

        Thread carparkThread = new Thread(carParkOrchestrator);
        Thread carThread = new Thread(carOrchestrator);
        Thread loggerThread = new Thread(logger);

        threads.add(carparkThread);
        threads.add(carThread);
        threads.add(loggerThread);

        carparkThread.start();
        carThread.start();
        loggerThread.start();
    } catch (Exception e) {

```

```

        System.err.println("Failed to initialize car park \"" + config.getName() + "\": " +
            e.getMessage());
    }
}

for (Thread thread : threads) {
    try {
        thread.join();
    } catch (InterruptedException e) {
        System.err.println("Simulation interrupted: " + e.getMessage());
    }
}
}

private static CarParkOrchestrator buildCarParkOrchestrator(Carpark<Vehicle> carpark, Config
    config) {
    ReaderFactory readerFactory = new ReaderFactory(config.getGateIdentifier());
    EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
    ExitFactory exitFactory = new ExitFactory(readerFactory);
    CarparkOrchestratorFactory orchestratorFactory = new CarparkOrchestratorFactory(carpark,
        entranceFactory, exitFactory);

    return orchestratorFactory.createCarparkOrchestrator(config.getNumberOfEntrances(),
        config.getNumberOfExits());
}

private static CarOrchestrator buildCarOrchestrator(CarParkOrchestrator carParkOrchestrator,
    Config config) {
    CarFactory carFactory = new CarFactory(carParkOrchestrator, config.getCarIdentifier());
    CarOrchestratorFactory carOrchestratorFactory = new CarOrchestratorFactory(carFactory);

    return carOrchestratorFactory.create(config.getNumberOfCars());
}
}

```

```
package org.main;

public class Regex {
    public static final String NUMBERPLATE = "[A-Z]{2}\\d{2} ?[A-Z]{3}$";
}
```

B.11 TESTS

B.12 abstracts

```
package org.main.abstracts;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;

import static org.mockito.Mockito.*;

class RunnableGateTest {

    private RunnableGate runnableGate;
    private IDReader<Identifiable> idReader;
    private Sensor<Identifiable> sensor;
    private Barrier barrier;
    private Carpark<Vehicle> carpark;
    private CarparkGateQueue<Vehicle> carparkGateQueue;
```

```

@BeforeEach
void setUp() {
    idReader = mock(IDReader.class);
    sensor = mock(Sensor.class);
    barrier = mock(Barrier.class);
    carpark = mock(Carpark.class);
    carparkGateQueue = mock(CarparkGateQueue.class);

    runnableGate = new TestRunnableGate("TestGate", idReader, sensor, barrier, carpark,
        carparkGateQueue);
}

@Test
void arrive_callsCarparkGateQueue_enqueue() {
    Vehicle vehicle = mock(Vehicle.class);
    runnableGate.arrive(vehicle);
    verify(carparkGateQueue).enqueue(vehicle);
}

private static class TestRunnableGate extends RunnableGate {
    public TestRunnableGate(String name, IDReader<Identifiable> idReader, Sensor<Identifiable>
        sensor, Barrier barrier, Carpark<Vehicle> carpark, CarparkGateQueue<Vehicle>
        carparkGateQueue) {
        super(name, idReader, sensor, barrier, carpark, carparkGateQueue);
    }

    @Override
    public void run() {
        // do nothing
    }
}
}

```

B.13 classes

```
package org.main.classes;

import org.junit.jupiter.api.Test;
import org.main.interfaces.Identifiable;
import stubs.TestIdentifiable;

import java.util.UUID;

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

class BarcodeIDReaderTest {

    @Test
    void read_returnsExpected() {
        String expected = UUID.randomUUID().toString();

        TestIdentifiable item = new TestIdentifiable();
        item.Id = expected;

        CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
        items.enqueue(item);

        BarcodeIDReader reader = new BarcodeIDReader();
        String actual = reader.read(items);

        assertEquals(actual, expected);
    }

    @Test
    void read_throwsIllegalArgumentException_whenNotUUID() {
        String failing = "TEST";
```

```

TestIdentifiable item = new TestIdentifiable();
item.Id = failing;

CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
items.enqueue(item);

BarcodeIDReader reader = new BarcodeIDReader();

assertThrows(IllegalArgumentException.class, () -> reader.read(items));
}

```

@Test

```

void read_throwsIllegalArgumentException_whenIdEmpty() {
    String failing = "";

    TestIdentifiable item = new TestIdentifiable();
    item.Id = failing;

    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
    items.enqueue(item);

    BarcodeIDReader reader = new BarcodeIDReader();

    assertThrows(IllegalArgumentException.class, () -> reader.read(items));
}

```

@Test

```

void read_idempotent() {
    TestIdentifiable item = new TestIdentifiable();
    item.Id = UUID.randomUUID().toString();
    TestIdentifiable item2 = new TestIdentifiable();
    item2.Id = UUID.randomUUID().toString();

    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
}

```

```

        items.enqueue(item);
        items.enqueue(item2);

        BarcodeIDReader reader = new BarcodeIDReader();
        String one = reader.read(items);
        String two = reader.read(items);

        assertEquals(one, two);
    }

    @Test
    void read_returnsNull_whenQueueIsEmpty() {
        CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
        BarcodeIDReader reader = new BarcodeIDReader();

        String actual = reader.read(items);

        assertNull(actual);
    }
}

package org.main.classes;

import org.junit.jupiter.api.Test;

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

class BarrierTest {

    @Test
    void raise() {
        Barrier barrier = new Barrier();

```

```

        barrier.raise();

        assertTrue(barrier.isUp());
    }

    @Test
    void lower() {
        Barrier barrier = new Barrier();

        barrier.lower();

        assertFalse(barrier.isUp());
    }
}

```

```

package org.main.classes;

import org.junit.jupiter.api.Test;

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

class CarparkGateQueueTest {

    @Test
    void isEmpty() {
        CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
        assertTrue(carparkGateQueue.isEmpty());
    }

    @Test
    void peek_returnsNull_whenEmpty() {
        CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
    }
}

```

```

        String result = carparkGateQueue.peek();

        assertNull(result);
    }

    @Test
    void peek() {
        String expected = "WV70 UFX";
        CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
        carparkGateQueue.enqueue(expected);

        String res = carparkGateQueue.peek();

        assertEquals(expected, res);
    }

    @Test
    void peek_firstIn_firstOut() {
        String expected = "WV70 UFX";
        CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
        carparkGateQueue.enqueue(expected);
        carparkGateQueue.enqueue("DF64 TLV");
        carparkGateQueue.enqueue("ABC 123");

        String res = carparkGateQueue.peek();

        assertEquals(expected, res);
    }

    @Test
    void peek_idempotent() {
        String expected = "WV70 UFX";
        CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
        carparkGateQueue.enqueue(expected);

```

```

    carparkGateQueue.enqueue("DF64 TLV");
    carparkGateQueue.enqueue("ABC 123");

    for(int i = 0; i < 10; i++) {
        String res = carparkGateQueue.peek();
        assertEquals(expected, res);
    }
}

@Test
void enqueue() {
    String expected = "WV70 UFX";
    CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
    carparkGateQueue.enqueue(expected);

    String res = carparkGateQueue.peek();
    assertEquals(expected, res);
}

@Test
void dequeue() {
    String expected = "WV70 UFX";
    CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
    carparkGateQueue.enqueue(expected);

    String res = carparkGateQueue.dequeue();
    assertEquals(expected, res);
}

@Test
void dequeue_firstIn_firstOut() {
    String expectedOne = "WV70 UFX";
    String expectedTwo = "WV70 UFX";
    String expectedThree = "WV70 UFX";

```

```

    CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
    carparkGateQueue.enqueue(expectedOne);
    carparkGateQueue.enqueue(expectedTwo);
    carparkGateQueue.enqueue(expectedThree);

    String one = carparkGateQueue.dequeue();
    String two = carparkGateQueue.dequeue();
    String three = carparkGateQueue.dequeue();

    assertEquals(expectedOne, one);
    assertEquals(expectedTwo, two);
    assertEquals(expectedThree, three);
}

@Test
void dequeue_null_whenQueueIsEmpty() {
    CarparkGateQueue<String> carparkGateQueue = new CarparkGateQueue<>();
    assertNull(carparkGateQueue.dequeue());
}
}

```

```

package org.main.classes;

import org.junit.jupiter.api.Test;
import org.main.interfaces.Identifiable;
import stubs.TestIdentifiable;

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

class IdentifiableIDReaderTest {
    @Test
    void read_returnsExpected() {

```

```

String expected = "TEST";

TestIdentifiable item = new TestIdentifiable();
item.Id = expected;

CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
items.enqueue(item);

IdentifiableIDReader reader = new IdentifiableIDReader();
String actual = reader.read(items);

assertEquals(actual, expected);
}

@Test
void read_idempotent() {
    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
    items.enqueue(new TestIdentifiable());
    items.enqueue(new TestIdentifiable());

    IdentifiableIDReader reader = new IdentifiableIDReader();
    String one = reader.read(items);
    String two = reader.read(items);

    assertEquals(one, two);
}

@Test
void read_returnsNull_whenQueueIsEmpty() {
    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
    IdentifiableIDReader reader = new IdentifiableIDReader();

    String actual = reader.read(items);

```

```
        assertNull(actual);
    }
}
```

```
package org.main.classes;

import org.junit.jupiter.api.Test;
import org.main.interfaces.Identifiable;
import stubs.TestIdentifiable;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
```

```
class IdentifiableSensorTest {

    @Test
    void carPresent_returnsTrue_whenCarPresent() {
        IdentifiableSensor identifiableSensor = new IdentifiableSensor();
        CarparkGateQueue<Identifiable> carparkGateQueue = new CarparkGateQueue<>();
        carparkGateQueue.enqueue(new TestIdentifiable());

        boolean actual = identifiableSensor.carPresent(carparkGateQueue);

        assertTrue(actual);
    }

    @Test
    void carPresent_returnsFalse_whenCarNotPresent() {
        IdentifiableSensor identifiableSensor = new IdentifiableSensor();
        CarparkGateQueue<Identifiable> carparkGateQueue = new CarparkGateQueue<>();
```

```

        boolean actual = identifiableSensor.carPresent(carparkGateQueue);

        assertFalse(actual);
    }
}

```

```

package org.main.classes;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.main.interfaces.Identifiable;
import stubs.TestIdentifiable;

import java.util.UUID;

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

class NumberplateIDReaderTest {

    @ParameterizedTest
    @CsvSource({"AB12 ABC", "WV70 UFX", "WV70UFX", "DF64 TLV"})
    void read_returnsExpected(String expected) {
        TestIdentifiable item = new TestIdentifiable();
        item.Id = expected;

        CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
        items.enqueue(item);

        NumberplateIDReader reader = new NumberplateIDReader();
        String actual = reader.read(items);
    }
}

```

```

        assertEquals(actual, expected);
    }

    @ParameterizedTest
    @CsvSource({"TEST", "bad", "wv70ufx", "abc "})
    void read_throwsIllegalArgumentException_whenNotNumberplate(String failing) {
        TestIdentifiable item = new TestIdentifiable();
        item.Id = failing;

        CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
        items.enqueue(item);

        NumberplateIDReader reader = new NumberplateIDReader();

        assertThrows(IllegalArgumentException.class, () -> reader.read(items));
    }

    @Test
    void read_throwsIllegalArgumentException_whenIdEmpty() {
        String failing = "";

        TestIdentifiable item = new TestIdentifiable();
        item.Id = failing;

        CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
        items.enqueue(item);

        NumberplateIDReader reader = new NumberplateIDReader();

        assertThrows(IllegalArgumentException.class, () -> reader.read(items));
    }

    @Test
    void read_idempotent() {

```

```

    TestIdentifiable item = new TestIdentifiable();
    item.Id = "WV70 UFX";
    TestIdentifiable item2 = new TestIdentifiable();
    item2.Id = "DF64 TLV";

    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
    items.enqueue(item);
    items.enqueue(item2);

    NumberplateIDReader reader = new NumberplateIDReader();
    String one = reader.read(items);
    String two = reader.read(items);

    assertEquals(one, two);
}

@Test
void read_returnsNull_whenQueueIsEmpty() {
    CarparkGateQueue<Identifiable> items = new CarparkGateQueue<>();
    NumberplateIDReader reader = new NumberplateIDReader();

    String actual = reader.read(items);

    assertNull(actual);
}
}

```

```

package org.main.classes;

import org.junit.jupiter.api.Test;
import org.main.abstracts.Vehicle;
import org.main.exceptions.FailureToAdmitException;
import org.main.interfaces.Identifiable;

```

```

import org.main.runnable.Car;

import java.util.concurrent.ConcurrentHashMap;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

class VehicleCarparkTest extends VehicleCarpark {
    public VehicleCarparkTest() {
        super("TEST", 10);
    }

    @Test
    void admit_persistsCarInMemory() throws FailureToAdmitException {
        String id = "WV70 UFX";
        VehicleCarpark carpark = new VehicleCarparkTest();

        Car car = mock(Car.class);
        when(car.getId()).thenReturn(id);

        carpark.admit(id, car);

        Identifiable result = carpark.vehicles.get(id);

        assertNotNull(result);
    }

    @Test
    void admit_throws_whenIdsClash() throws FailureToAdmitException {

```

```

String id = "WV70 UFX";

VehicleCarpark vehicleCarpark = new VehicleCarparkTest();
Car car = mock(Car.class);
when(car.getId()).thenReturn(id);

Car clash = mock(Car.class);
when(clash.getId()).thenReturn(id);

vehicleCarpark.admit(id, car);
assertThrows(IllegalArgumentException.class, () -> vehicleCarpark.admit(id, clash));
}

@Test
void admit_throws_whenCapacityReached() {
    String id = "abc";

    ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();
    map.put("123", mock(Vehicle.class));
    map.put("345", mock(Vehicle.class));
    map.put("567", mock(Vehicle.class));

    VehicleCarpark carpark = new VehicleCarparkTest();
    carpark.vehicles = map;
    carpark.capacity = 3;

    assertThrows(FailureToAdmitException.class, () -> carpark.admit(id, mock(Vehicle.class)));
}

@Test
void release_removesCarFromMemory() {
    String id = "WV70 UFX";

    Vehicle toRelease = mock(Vehicle.class);

    ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();

```

```

        map.put(id, toRelease);
        map.put("345", mock(Vehicle.class));

        VehicleCarpark carpark = new VehicleCarparkTest();
        carpark.vehicles = map;

        carpark.release(id);

        Identifiable result = carpark.vehicles.get(id);

        assertNull(result);
    }

    @Test
    void release_doesNotRemoveOtherCars() {
        String id = "WV70 UFX";
        Vehicle toRelease = mock(Vehicle.class);

        String toStayId = "DF64 TLV";
        Vehicle toStay = mock(Vehicle.class);

        ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();
        map.put(id, toRelease);
        map.put(toStayId, toStay);

        VehicleCarpark carpark = new VehicleCarparkTest();
        carpark.vehicles = map;

        carpark.release(id);

        Identifiable result = carpark.vehicles.get(toStayId);

        assertNotNull(result);
    }
}

```

```

@Test
void release_idempotent() {
    String id = "WV70 UFX";
    Vehicle toRelease = mock(Vehicle.class);

    ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();
    map.put(id, toRelease);

    VehicleCarpark carpark = new VehicleCarparkTest();
    carpark.vehicles = map;

    carpark.release(id);
    carpark.release(id);
    carpark.release(id);

    Identifiable result = carpark.vehicles.get(id);

    assertNull(result);
}

```

```

@Test
void spacesLeft_returnsFreeSpace() {
    ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();
    map.put("123", mock(Vehicle.class));
    map.put("345", mock(Vehicle.class));
    map.put("567", mock(Vehicle.class));

    VehicleCarpark carpark = new VehicleCarparkTest();
    carpark.vehicles = map;
    carpark.capacity = 10;

    int expected = 7; // 10 - 3
    int result = carpark.spacesLeft();
}

```

```

        assertEquals(expected, result);
    }

    @Test
    void spacesLeft_returnsZero_whenCapacityReached() {
        ConcurrentHashMap<String, Vehicle> map = new ConcurrentHashMap<>();
        map.put("123", mock(Vehicle.class));
        map.put("345", mock(Vehicle.class));
        map.put("567", mock(Vehicle.class));

        VehicleCarpark carpark = new VehicleCarparkTest();
        carpark.vehicles = map;
        carpark.capacity = 3;

        int expected = 0;
        int result = carpark.spacesLeft();

        assertEquals(expected, result);
    }

    @Test
    void testGetName() {
        String expected = "TEST";

        VehicleCarpark carpark = new VehicleCarpark(expected, 1);
        String actual = carpark.getName();

        assertEquals(expected, actual);
    }

    @Test
    void admit_callsSetAdmitted_whenSuccessful() throws FailureToAdmitException {
        String id = "WV70 UFX";

```

```

Vehicle vehicle = mock(Vehicle.class);

VehicleCarpark carpark = new VehicleCarparkTest();
carpark.admit(id, vehicle);

verify(vehicle).setAdmitted(true);
}

@Test
void admit_doesNotCallSetAdmitted_whenIdClash() throws FailureToAdmitException {
    ConcurrentHashMap<String, Vehicle> map = mock(ConcurrentHashMap.class);
    when(map.containsKey(anyString())).thenReturn(true);

    String id = "WV70 UFX";
    Vehicle vehicle = mock(Vehicle.class);

    VehicleCarpark carpark = new VehicleCarparkTest();
    carpark.vehicles = map;

    assertThrows(IllegalArgumentException.class, () -> carpark.admit(id, vehicle));
    verify(vehicle, times(0)).setAdmitted(true);
}

@Test
void admit_doesNotCallSetAdmitted_whenCapacityReached() throws FailureToAdmitException {
    int capacity = 3;

    ConcurrentHashMap<String, Vehicle> map = mock(ConcurrentHashMap.class);
    when(map.size()).thenReturn(capacity);

    String id = "WV70 UFX";
    Vehicle vehicle = mock(Vehicle.class);

    VehicleCarpark carpark = new VehicleCarparkTest();

```

```

        carpark.capacity = capacity;
        carpark.vehicles = map;

        assertThrows(FailureToAdmitException.class, () -> carpark.admit(id, vehicle));
        verify(vehicle, times(0)).setAdmitted(true);
    }

    @Test
    void release_callsSetsAdmittedFalse() {
        ConcurrentHashMap<String, Vehicle> map = mock(ConcurrentHashMap.class);

        String id = "WV70 UFX";
        Vehicle vehicle = mock(Vehicle.class);
        when(map.remove(eq(id))).thenReturn(vehicle);

        VehicleCarpark carpark = new VehicleCarparkTest();
        carpark.vehicles = map;
        carpark.release(id);

        verify(vehicle).setAdmitted(false);
    }
}

```

B.14 config

```

package org.main.config;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;
import org.main.enums.IdentifierOption;

import java.io.File;
import java.io.FileWriter;

```

```

import java.io.IOException;
import java.util.List;

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

class ConfigLoaderTest {

    private static final String TEST_CONFIG = "testconfig.json";

    @AfterEach
    void cleanup() {
        File testFile = new File(TEST_CONFIG);
        if (testFile.exists()) {
            testFile.delete();
        }

        ConfigLoader.resetInstance();
    }

    @Test
    void getConfigs_returnsExpected_singleConfigFromFile() throws IOException {
        String singleConfigJson = "[{\"name\":\"Carpark A\", \"capacity\":50, \"numberOfEntrances\":2, \"numberOfExits\":1}]";
        createTestConfigFile(singleConfigJson);

        ConfigLoader loader = ConfigLoader.getInstance(TEST_CONFIG);
        List<Config> configs = loader.getConfigs();

        assertNotNull(configs);
        assertEquals(1, configs.size());
        assertEquals("Carpark A", configs.get(0).getName());
        assertEquals(50, configs.get(0).getCapacity());
        assertEquals(2, configs.get(0).getNumberOfEntrances());
        assertEquals(1, configs.get(0).getNumberOfExits());
    }
}

```

```

@Test
void getConfigs_returnsExpected_multipleConfigsFromFile() throws IOException {
    String multipleConfigJson =
        "[{\"name\":\"CarparkA\",\"capacity\":10,\"numberOfEntrances\":3,\"numberOfExits\":2,\"numberOfCars\":200,\"logInterval\":1000,\"carIdentifier\":\"NUMBERPLATE\",\"gateIdentifier\":\"NUMBERPLATE\"},\n" +
        "{\"name\":\"CarparkB\",\"capacity\":15,\"numberOfEntrances\":1,\"numberOfExits\":1,\"numberOfCars\":100,\"logInterval\":1000,\"carIdentifier\":\"BARCODE\",\"gateIdentifier\":\"NUMBERPLATE\"}]\n";
    createTestConfigFile(multipleConfigJson);

    ConfigLoader loader = ConfigLoader.getInstance(TEST_CONFIG);
    List<Config> configs = loader.getConfigs();

    assertNotNull(configs);
    assertEquals(2, configs.size());

    Config configA = configs.stream().filter(c ->
        c.getName().equals("CarparkA")).findFirst().get();
    assertNotNull(configA);
    assertEquals(10, configA.getCapacity());
    assertEquals(3, configA.getNumberOfEntrances());
    assertEquals(2, configA.getNumberOfExits());
    assertEquals(200, configA.getNumberOfCars());
    assertEquals(1000, configA.getLogInterval());
    assertEquals(IdentifierOption.NUMBERPLATE, configA.getCarIdentifier());
    assertEquals(IdentifierOption.NUMBERPLATE, configA.getGateIdentifier());

    Config configB = configs.stream().filter(c ->
        c.getName().equals("CarparkB")).findFirst().get();
    assertNotNull(configB);
    assertEquals(15, configB.getCapacity());
    assertEquals(1, configB.getNumberOfEntrances());
    assertEquals(1, configB.getNumberOfExits());
    assertEquals(100, configB.getNumberOfCars());
    assertEquals(1000, configB.getLogInterval());
    assertEquals(IdentifierOption.BARCODE, configB.getCarIdentifier());
}

```

```

        assertEquals(IdentifierOption.BARCODE, configB.getGateIdentifier());
    }

    @Test
    void getConfigs_usesDefaultConfig_whenPathNull() throws IOException {
        ConfigLoader loader = ConfigLoader.getInstance(null);

        List<Config> configs = loader.getConfigs();

        assertNotNull(configs);
        assertEquals(1, configs.size());
    }

    @Test
    void getConfigs_usesDefaultConfig_whenPathEmpty() throws IOException {
        ConfigLoader loader = ConfigLoader.getInstance("");

        List<Config> configs = loader.getConfigs();

        assertNotNull(configs);
        assertEquals(1, configs.size());
    }

    @Test
    void getInstance_throws_whenInvalidJson() {
        String invalidJson = "[{\"name\":\"Carpark
            A\", \"capacity\":50, \"numberOfEntrances\":2, \"numberOfExits\":1\";
        createTestConfigFile(invalidJson);

        assertThrows(IOException.class, () -> ConfigLoader.getInstance(TEST_CONFIG));
    }

    @Test
    void ConfigLoader_isSingleton() throws IOException {
        String configJson = "[{\"name\":\"Carpark
            A\", \"capacity\":50, \"numberOfEntrances\":2, \"numberOfExits\":1}]]";

```

```

        createTestConfigFile(configJson);

        ConfigLoader loader1 = ConfigLoader.getInstance(TEST_CONFIG);
        ConfigLoader loader2 = ConfigLoader.getInstance(TEST_CONFIG);

        assertSame(loader1, loader2);
    }

    @Test
    void getConfigs_returnsExpected_defaultConfig() throws IOException {
        ConfigLoader loader = ConfigLoader.getInstance();
        List<Config> configs = loader.getConfigs();

        assertNotNull(configs);
        assertEquals(1, configs.size());
    }

    private void createTestConfigFile(String content) {
        createTestConfigFile(content, TEST_CONFIG);
    }

    private void createTestConfigFile(String content, String filePath) {
        try (FileWriter writer = new FileWriter(filePath)) {
            writer.write(content);
        } catch (IOException e) {
            throw new RuntimeException("Failed to create test config file", e);
        }
    }
}

```

B.15 factories

```
package org.main.factories;
```

```

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.main.Regex;
import org.main.enums.IdentifierOption;
import org.main.runnable.Car;
import org.main.runnable.CarParkOrchestrator;
import org.main.runnable.Entrance;
import org.main.runnable.Exit;
import org.mockito.Mock;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

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

class CarFactoryTest {
    List<Entrance> mockEntrances;
    List<Exit> mockExits;

    @Mock
    private CarParkOrchestrator carParkOrchestrator;
    private CarFactory carFactory;

    @BeforeEach
    void setUp() {
        carParkOrchestrator = mock(CarParkOrchestrator.class);

        mockEntrances = new ArrayList<>();
        mockExits = new ArrayList<>();
    }

```

```

    for(int i = 0; i < 4; i++) {
        mockExits.add(mock(Exit.class));
        mockEntrances.add(mock(Entrance.class));
    }

    when(carParkOrchestrator.getEntrances()).thenReturn(mockEntrances);
    when(carParkOrchestrator.getExits()).thenReturn(mockExits);
}

@Test
void createCar_createsCar_withEntranceAndExitFromList() {
    carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.ANY);

    Car car = carFactory.createCar();

    assertNotNull(car.getEntrance());
    assertNotNull(car.getExit());

    assertTrue(mockEntrances.contains(car.getEntrance()));
    assertTrue(mockExits.contains(car.getExit()));
}

@Test
void createCar_createsCarWithId_whenAnyIdentifierOption() {
    carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.ANY);
    Car car = carFactory.createCar();

    assertNotNull(car.getId());
    assertFalse(car.getId().isEmpty());
}

@Test
void createCar_createsCarWithUUID_whenBarcodeIdentifierOption() {

```

```

        carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.BARCODE);
        Car car = carFactory.createCar();

        assertDoesNotThrow(() -> UUID.fromString(car.getId()));
    }

    @Test
    void createCar_createsCarWithNumberplate_whenNumberplateIdentifierOption() {
        carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.NUMBERPLATE);
        Car car = carFactory.createCar();

        Pattern pattern = Pattern.compile(Regex.NUMBERPLATE);
        Matcher matcher = pattern.matcher(car.getId());
        assertTrue(matcher.matches());
    }

    @Test
    void createCar_throwsException_whenEntrancesIsEmpty() {
        when(carParkOrchestrator.getEntrances()).thenReturn(new ArrayList<>());

        carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.ANY);

        assertThrows(IllegalArgumentException.class, () -> carFactory.createCar());
    }

    @Test
    void createCar_throwsException_whenExitsIsEmpty() {
        when(carParkOrchestrator.getExits()).thenReturn(new ArrayList<>());

        carFactory = new CarFactory(carParkOrchestrator, IdentifierOption.ANY);

        assertThrows(IllegalArgumentException.class, () -> carFactory.createCar());
    }
}

```

```
package org.main.factories;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import org.main.runnable.Car;
import org.main.runnable.CarOrchestrator;
import org.mockito.Mock;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
import static org.mockito.MockitoAnnotations.initMocks;

class CarOrchestratorFactoryTest {

    @Mock
    CarFactory carFactory;

    CarOrchestratorFactory carOrchestratorFactory;

    @BeforeEach
    void setUp() {
        initMocks(this);

        carOrchestratorFactory = new CarOrchestratorFactory(carFactory);

        when(carFactory.createCar()).thenReturn(mock(Car.class));
    }
}
```

```

@ParameterizedTest
@ValueSource(ints = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
void create(int expected) {
    CarOrchestrator carOrchestrator = carOrchestratorFactory.create(expected);

    int actual = carOrchestrator.getCars().size();

    verify(carFactory, times(expected)).createCar();

    assertEquals(expected, actual);
}

@ParameterizedTest
@ValueSource(ints = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
void create_returnsCreatedCars(int count) {
    CarOrchestrator carOrchestrator = carOrchestratorFactory.create(count);

    List<Car> cars = carOrchestrator.getCars();

    assertAll(() -> cars.forEach(Assertions::assertNotNull));
}
}

package org.main.factories;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.main.abstracts.Vehicle;
import org.main.interfaces.Carpark;
import org.main.runnable.CarParkOrchestrator;

```

```

import org.main.runnable.Entrance;
import org.main.runnable.Exit;
import org.mockito.Mock;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.contains;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

class CarparkOrchestratorFactoryTest {

    @Mock
    EntranceFactory entranceFactory;

    @Mock
    ExitFactory exitFactory;

    @Mock
    Carpark<Vehicle> carpark;

    CarparkOrchestratorFactory carparkOrchestratorFactory;

    @BeforeEach
    void setUp(){
        entranceFactory = mock(EntranceFactory.class);
        exitFactory = mock(ExitFactory.class);
        carpark = mock(Carpark.class);

        carparkOrchestratorFactory = new CarparkOrchestratorFactory(carpark, entranceFactory,
            exitFactory);
    }
}

```

```

@Test
void createCarparkOrchestrator() {
    CarParkOrchestrator actual = carparkOrchestratorFactory.createCarparkOrchestrator(1,1);

    assertNotNull(actual);
}

@ParameterizedTest
@CsvSource({"0", "1", "2", "3", "10"})
void createCarparkOrchestrator_withCorrectNumEntrances(int expected) {
    CarParkOrchestrator res = carparkOrchestratorFactory.createCarparkOrchestrator( expected,1);

    int actual = res.getEntrances().size();
    assertEquals(expected, actual);
}

@ParameterizedTest
@CsvSource({"0", "1", "2", "3", "10"})
void createCarparkOrchestrator_withCorrectNumExits(int expected) {
    CarParkOrchestrator res = carparkOrchestratorFactory.createCarparkOrchestrator(1,expected);

    int actual = res.getExits().size();
    assertEquals(expected, actual);
}

@Test
void createCarparkOrchestrator_usesEntranceFactory() {
    Entrance entrance = mock(Entrance.class);
    when(entranceFactory.createEntrance(any(), any())).thenReturn(entrance);

    CarParkOrchestrator res = carparkOrchestratorFactory.createCarparkOrchestrator(1,1);

    List<Entrance> entrances = res.getEntrances();
    assertTrue(entrances.contains(entrance));
}

```

```

}

@Test
void createCarparkOrchestrator_usesExitFactory() {
    Exit exit = mock(Exit.class);
    when(exitFactory.createExit(any(), any())).thenReturn(exit);

    CarParkOrchestrator res = carparkOrchestratorFactory.createCarparkOrchestrator(1,1);

    List<Exit> exits = res.getExits();
    assertTrue(exits.contains(exit));
}

@Test
void createCarparkOrchestrator_passesCarpark() {
    carparkOrchestratorFactory.createCarparkOrchestrator(1,1);

    verify(entranceFactory).createEntrance(any(), eq(carpark));
    verify(exitFactory).createExit(any(), eq(carpark));
}

@Test
void createCarparkOrchestrator_generatesNameBasedOnCarparkName() {
    String expected = "TEST";
    when(carpark.getName()).thenReturn(expected);

    carparkOrchestratorFactory.createCarparkOrchestrator( 1,1);

    verify(entranceFactory).createEntrance(contains(expected), any());
    verify(exitFactory).createExit(contains(expected), any());
}
}

```

```

package org.main.factories;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;
import org.main.runnable.Entrance;

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

class EntranceFactoryTest {
    Carpark<Vehicle> carpark;
    ReaderFactory readerFactory;

    @BeforeEach
    void setUp() {
        carpark = mock(Carpark.class);
        readerFactory = mock(ReaderFactory.class);
        when(readerFactory.create()).thenReturn(mock(IDReader.class));
    }

    @Test
    void createEntrance() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);
    }
}

```

```

        assertNotNull(entrance);
    }

    @Test
    void createEntrance_hasName() {
        String expected = "TEST";

        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);

        Entrance entrance = entranceFactory.createEntrance(expected, carpark);
        String actual = entrance.getName();
        assertEquals(expected, actual);
    }

    @Test
    void createEntrance_hasIdentifier() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);

        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);
        IDReader<Identifiable> idReader = entrance.getIdReader();
        assertNotNull(idReader);
    }

    @Test
    void createEntrance_hasSensor() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);
        Sensor<Identifiable> sensor = entrance.getSensor();

        assertNotNull(sensor);
    }

    @Test
    void createEntrance_hasBarrier() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);

```

```

        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);
        Barrier barrier = entrance.getBarrier();

        assertNotNull(barrier);
    }

    @Test
    void createEntrance_hasCarpark() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);

        Carpark actual = entrance.getCarpark();
        assertEquals(carpark, actual);
    }

    @Test
    void createEntrance_hasEmptyGateQueue() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
        Entrance entrance = entranceFactory.createEntrance("TEST", carpark);

        CarparkGateQueue<Vehicle> queue = entrance.getCarparkGateQueue();

        assertNotNull(queue);
        assertTrue(queue.isEmpty());
    }

    @Test
    void createEntrance_callsReaderFactoryCreate() {
        EntranceFactory entranceFactory = new EntranceFactory(readerFactory);
        entranceFactory.createEntrance("TEST", carpark);

        verify(readerFactory).create();
    }
}

```

```
package org.main.factories;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Identifiable;
import org.main.interfaces.Sensor;
import org.main.runnable.Exit;

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

class ExitFactoryTest {

    Carpark<Vehicle> carpark;

    ReaderFactory readerFactory;

    @BeforeEach
    void setUp() {
        carpark = mock(Carpark.class);
        readerFactory = mock(ReaderFactory.class);
        when(readerFactory.create()).thenReturn(mock(IDReader.class));
    }

    @Test
    void createExit() {
        ExitFactory exitFactory = new ExitFactory(readerFactory);
```

```

        Exit exit = exitFactory.createExit("TEST", carpark);

        assertNotNull(exit);
    }

    @Test
    void createExit_hasName() {
        String expected = "TEST";

        ExitFactory exitFactory = new ExitFactory(readerFactory);

        Exit exit = exitFactory.createExit(expected, carpark);
        String actual = exit.getName();
        assertEquals(expected, actual);
    }

    @Test
    void createExit_hasIdentifier() {
        ExitFactory exitFactory = new ExitFactory(readerFactory);

        Exit exit = exitFactory.createExit("TEST", carpark);
        IDReader<Identifiable> idReader = exit.getIdReader();
        assertNotNull(idReader);
    }

    @Test
    void createExit_hasSensor() {
        ExitFactory exitFactory = new ExitFactory(readerFactory);

        Exit exit = exitFactory.createExit("TEST", carpark);
        Sensor<Identifiable> sensor = exit.getSensor();

        assertNotNull(sensor);
    }

    @Test

```

```

void createExit_hasBarrier() {
    ExitFactory exitFactory = new ExitFactory(readerFactory);
    Exit exit = exitFactory.createExit("TEST", carpark);
    Barrier barrier = exit.getBarrier();

    assertNotNull(barrier);
}

@Test
void createExit_hasCarpark() {
    ExitFactory exitFactory = new ExitFactory(readerFactory);
    Exit exit = exitFactory.createExit("TEST", carpark);

    Carpark actual = exit.getCarpark();
    assertEquals(carpark, actual);
}

@Test
void createExit_hasEmptyGateQueue() {
    ExitFactory exitFactory = new ExitFactory(readerFactory);
    Exit exit = exitFactory.createExit("TEST", carpark);

    CarparkGateQueue<Vehicle> queue = exit.getCarparkGateQueue();

    assertNotNull(queue);
    assertTrue(queue.isEmpty());
}

@Test
void createExit_callsReaderFactoryCreate() {
    ExitFactory exitFactory = new ExitFactory(readerFactory);
    exitFactory.createExit("TEST", carpark);

    verify(readerFactory).create();
}

```

```
}  
}
```

```
package org.main.factories;  
  
import org.junit.jupiter.api.Test;  
import org.main.classes.BarcodeIDReader;  
import org.main.classes.IdentifiableIDReader;  
import org.main.classes.NumberplateIDReader;  
import org.main.enums.IdentifierOption;  
import org.main.interfaces.IDReader;  
import org.main.interfaces.Identifiable;  
  
import static org.junit.jupiter.api.Assertions.*;  
  
class ReaderFactoryTest {  
  
    @Test  
    void create_barcodeIDReader() {  
        ReaderFactory factory = new ReaderFactory(IdentifierOption.BARCODE);  
        IDReader<Identifiable> idReader = factory.create();  
  
        Class actual = idReader.getClass();  
        Class expected = BarcodeIDReader.class;  
  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void create_numberplateIDReader() {  
        ReaderFactory factory = new ReaderFactory(IdentifierOption.NUMBERPLATE);  
        IDReader<Identifiable> idReader = factory.create();
```

```

        Class actual = idReader.getClass();
        Class expected = NumberplateIDReader.class;

        assertEquals(expected, actual);
    }

    @Test
    void create_identifiableIDReader() {
        ReaderFactory factory = new ReaderFactory(IdentifierOption.ANY);
        IDReader<Identifiable> idReader = factory.create();

        Class actual = idReader.getClass();
        Class expected = IdentifiableIDReader.class;

        assertEquals(expected, actual);
    }

    @Test
    void create_nullIdentifierOption_returnsIdentifiableIDReader() {
        ReaderFactory factory = new ReaderFactory(null);

        IDReader<Identifiable> idReader = factory.create();

        Class actual = idReader.getClass();
        Class expected = IdentifiableIDReader.class;

        assertEquals(expected, actual);
    }
}

```

B.16 runnable

```
package org.main.runnable;
```

```

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;

class CarOrchestratorTest {
    List<Car> cars;

    CarOrchestrator carOrchestrator;

    @BeforeEach
    void setUp() {
        cars = new ArrayList<>();
        for(int i = 0; i < 10; i++) {
            cars.add(mock(Car.class));
        }

        carOrchestrator = new CarOrchestrator(cars);
    }

    @Test
    void run() throws InterruptedException {
        carOrchestrator.run();

        Thread.sleep(100);

        for(Car car : cars) {
            verify(car).run();
        }
    }
}

```

```
}
```

```
package org.main.runnable;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.main.abstracts.RunnableGate;

import java.util.Arrays;
import java.util.List;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

class CarParkOrchestratorTest {

    private CarParkOrchestrator carParkGateOrchestrator;
    private List<Entrance> mockEntrances;
    private List<Exit> mockExits;

    @BeforeEach
    void setUp() {
        Entrance entrance1 = mock(Entrance.class);
        Entrance entrance2 = mock(Entrance.class);
        Exit exit1 = mock(Exit.class);
        Exit exit2 = mock(Exit.class);

        mockEntrances = Arrays.asList(entrance1, entrance2);
        mockExits = Arrays.asList(exit1, exit2);

        carParkGateOrchestrator = new CarParkOrchestrator(mockEntrances, mockExits);
    }
}
```

```

@Test
void run_ShouldRunAllEntrancesAndExits() throws InterruptedException {
    carParkGateOrchestrator.run();

    Thread.sleep(100);

    for (RunnableGate entrance : mockEntrances) {
        verify(entrance, times(1)).run();
    }

    for (RunnableGate exit : mockExits) {
        verify(exit, times(1)).run();
    }
}
}

```

```

package org.main.runnable;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

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

class CarTest {
    private Car car;
    private Entrance mockEntrance;
    private Exit mockExit;

```

```

@BeforeEach
void setUp() {
    mockEntrance = mock(Entrance.class);
    mockExit = mock(Exit.class);
    car = new Car("test-id", mockEntrance, mockExit);
}

@ParameterizedTest
@CsvSource({"WV70 UFX", "DF64 TLV", "123"})
void getId(String expected) {
    Car car = new Car(expected, mockEntrance, mockExit);
    String result = car.getId();
    assertEquals(expected, result);
}

@Test
void run_flowsAsExpected() {
    car.setAdmitted(true);
    car.setSleepRandomBound(1);

    car.run();

    verify(mockEntrance).arrive(car);
    verify(mockExit).arrive(car);
}

@Test
void admitted_shouldBeFalse_initially() {
    assertFalse(car.isAdmitted());
}

@Test
void exitArrive_notCalled_whenNotAdmitted() {

```

```

        car.setDO_NOT_POLL_IF_NOT_ADMITTED(true);
        car.setSleepRandomBound(1);

        car.run();

        verify(mockEntrance).arrive(car);
        verify(mockExit, times(0)).arrive(car);
    }

    @ParameterizedTest
    @CsvSource({"1", "2", "10", "500", "5000"})
    void setSleepRandomBound(int expected) {
        car.setSleepRandomBound(expected);

        int actual = car.getSLEEP_RANDOM_BOUND();
        assertEquals(expected, actual);
    }

    @ParameterizedTest
    @CsvSource({"0", "-1", "-2", "-10", "-100"})
    void setSleepRandomBound_setsOne_ifLessThanOnw(int value) {
        int expected = 1;
        car.setSleepRandomBound(value);

        int actual = car.getSLEEP_RANDOM_BOUND();
        assertEquals(expected, actual);
    }
}

package org.main.runnable;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

```

```

import org.junit.jupiter.api.extension.ExtendWith;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.exceptions.FailureToAdmitException;
import org.main.interfaces.Identifiable;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Sensor;

import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class EntranceTest {

    @Mock
    private IDReader<Identifiable> idReader;

    @Mock
    private Sensor<Identifiable> sensor;

    @Mock
    private Barrier barrier;

    @Mock
    private Carpark<Vehicle> carpark;

    @Mock
    private CarparkGateQueue<Vehicle> carparkGateQueue;

    @InjectMocks

```

```

private Entrance entrance;

@BeforeEach
public void setUp() {
    entrance = new Entrance("Test Gate", idReader, sensor, barrier, carpark, carparkGateQueue);
    entrance.setKEEP_RUNNING(false);
    entrance.setPOLL_INTERVAL(1);
}

@Test
public void run_admitsOneCar_whenSpaceAvailable() throws InterruptedException,
    FailureToAdmitException {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(true, false);
    when(carpark.spacesLeft()).thenReturn(1);
    when(idReader.read(carparkGateQueue)).thenReturn("car-id");
    when(carparkGateQueue.peek()).thenReturn(mock(Vehicle.class));

    Thread testThread = new Thread(entrance);
    testThread.start();
    testThread.join();

    verify(barrier).raise();
    verify(carpark).admit(eq("car-id"), any());
    verify(carparkGateQueue).dequeue();
    verify(barrier).lower();
}

@Test
public void run_doesNotAdmitCar_whenNoSpaceAvailable() throws InterruptedException,
    FailureToAdmitException {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(true);
    when(carpark.spacesLeft()).thenReturn(0);

    Thread testThread = new Thread(() -> {

```

```

        entrance.run();
    });
    testThread.start();
    testThread.join();

    verify(barrier, never()).raise();
    verify(carpark, never()).admit(anyString(), any());
    verify(carparkGateQueue, never()).dequeue();
    verify(barrier, never()).lower();
}

@Test
public void run_doesNotCompleteFlow_whenNoCarPresent() throws InterruptedException,
    FailureToAdmitException {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(false);

    Thread testThread = new Thread(() -> {
        entrance.run();
    });
    testThread.start();
    testThread.join();

    verify(barrier, never()).raise();
    verify(carpark, never()).admit(anyString(), any());
    verify(carpark, never()).spacesLeft();
    verify(carparkGateQueue, never()).dequeue();
    verify(barrier, never()).lower();
}

@Test
public void run_doesNotCallDequeue_whenCarparkAdmitFailsWithFailureToAdmitException() throws
    Exception {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(true);
    when(carpark.spacesLeft()).thenReturn(1);

```

```

        when(idReader.read(carparkGateQueue)).thenReturn("car-id");
        when(carparkGateQueue.peek()).thenReturn(mock(Vehicle.class));

        doThrow(new FailureToAdmitException("")).when(carpark).admit(any(String.class), any());

        Thread testThread = new Thread(entrance);
        testThread.start();
        testThread.join();

        verify(carparkGateQueue, never()).dequeue();
    }

    @Test
    public void run_callsDequeue_whenCarparkAdmitFailsWithIllegalArgumentException() throws Exception {
        when(sensor.carPresent(carparkGateQueue)).thenReturn(true, false);
        when(carpark.spacesLeft()).thenReturn(1);
        when(idReader.read(carparkGateQueue)).thenReturn("car-id");
        when(carparkGateQueue.peek()).thenReturn(mock(Vehicle.class));
        doThrow(new IllegalArgumentException("")).when(carpark).admit(any(String.class), any());

        Thread testThread = new Thread(entrance);
        testThread.start();
        testThread.join();

        verify(carparkGateQueue).dequeue();
    }

    @Test
    public void run_callsDequeueAndNotAdmit_whenIDReaderThrowsIllegalArgumentException() throws
        Exception {
        when(sensor.carPresent(carparkGateQueue)).thenReturn(true, false);
        when(carpark.spacesLeft()).thenReturn(1);
        when(idReader.read(carparkGateQueue)).thenThrow(new IllegalArgumentException(""));
    }

```

```

        Thread testThread = new Thread(entrance);
        testThread.start();
        testThread.join();

        verify(carparkGateQueue, times(1)).dequeue();
        verify(carpark, never()).admit(anyString(), any());
    }
}

```

```

package org.main.runnable;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.main.abstracts.Vehicle;
import org.main.classes.Barrier;
import org.main.classes.CarparkGateQueue;
import org.main.exceptions.FailureToAdmitException;
import org.main.interfaces.Identifiable;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import org.main.interfaces.Carpark;
import org.main.interfaces.IDReader;
import org.main.interfaces.Sensor;

import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class ExitTest {

    @Mock
    private IDReader<Identifiable> idReader;

```

```

@Mock
private Sensor<Identifiable> sensor;

@Mock
private Barrier barrier;

@Mock
private Carpark<Vehicle> carpark;

@Mock
private CarparkGateQueue<Vehicle> carparkGateQueue;

@InjectMocks
private Exit exit;

@BeforeEach
public void setUp() {
    exit = new Exit("Test Gate", idReader, sensor, barrier, carpark, carparkGateQueue);
    exit.setKEEP_RUNNING(false);
    exit.setPOLL_INTERVAL(1);
}

@Test
public void run_releasesCarWhenPresent() throws InterruptedException {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(true);
    when(idReader.read(carparkGateQueue)).thenReturn("car-id");

    Thread testThread = new Thread(exit);
    testThread.start();
    testThread.join();

    verify(barrier, atLeast(1)).raise();
    verify(carpark, atLeast(1)).release(eq("car-id"));
}

```

```

        verify(carParkGateQueue, atLeast(1)).dequeue();
        verify(barrier, atLeast(1)).lower();
    }

    @Test
    public void run_releasesOneCar() throws InterruptedException {
        when(sensor.carPresent(carParkGateQueue)).thenReturn(true, false);
        when(idReader.read(carParkGateQueue)).thenReturn("car-id");

        Thread testThread = new Thread(exit);
        testThread.start();
        testThread.join();

        verify(barrier).raise();
        verify(carPark).release(eq("car-id"));
        verify(carParkGateQueue).dequeue();
        verify(barrier).lower();
    }

```

```

    @Test
    public void run_noCarPresent() throws InterruptedException, FailureToAdmitException {
        when(sensor.carPresent(carParkGateQueue)).thenReturn(false);

        Thread testThread = new Thread(exit);
        testThread.start();
        testThread.join();

        verify(barrier, never()).raise();
        verify(carPark, never()).admit(anyString(), any());
        verify(carPark, never()).spacesLeft();
        verify(carParkGateQueue, never()).dequeue();
        verify(barrier, never()).lower();
    }

```

```

@Test
public void run_callsDequeueAndNotRelease_whenIDReaderThrowsIllegalArgumentException() throws
    InterruptedException {
    when(sensor.carPresent(carparkGateQueue)).thenReturn(true, false);
    when(idReader.read(carparkGateQueue)).thenThrow(new IllegalArgumentException(""));

    Thread testThread = new Thread(exit);
    testThread.start();
    testThread.join();

    verify(barrier, never()).raise();
    verify(carparkGateQueue, times(1)).dequeue();
    verify(barrier, times(1)).lower();
}
}

```

B.17 stubs

```

package stubs;

import org.main.interfaces.Identifiable;

public class TestIdentifiable implements Identifiable {
    public String Id = "TEST";

    @Override
    public String getId() {
        return Id;
    }
}

```
