

OO report - iestyn green

by Iestyn Green

Submission date: 25-Apr-2025 01:23PM (UTC+0100)

Submission ID: 256608866

File name: 58847_Iestyn_Green_OO_report_-_iestyn_green_662765_1246873489.docx (149.78K)

Word count: 3589

Character count: 18801

Iestyn Green

2420522

Object Oriented Formal Report

Table of Contents

Introduction 3

Object-Oriented Design Principles 4

 Inheritance..... 4

 Interfaces 4

 Abstract classes 5

 Polymorphism 5

 Generics..... 6

Data Structures and Memory Management..... 7

Class Descriptions and Relationships 8

Testing..... 11

Conclusion..... 12

Bibliography 13

Introduction

The goal for this project was to develop a car park-based application that follows object-oriented programming principles. This car park application should include a barrier, sensor, ID reader and full sign which toggles depending on whether the car park is full to the set capacity. This application should be able to detect if there is a car present at the barrier and will scan the barcode or numberplate of the car at the barrier, if the barcode or numberplate is valid then depending on if there is space the barrier will rise and it will allow the vehicle to enter. Then it will do the same for a vehicle trying to exit the car park. The vehicle's ID should be stored and dynamically updated depending on if it is in the car park or has left, same with the available spaces. Should an invalid barcode, or numberplate or if there is no space for the vehicle then the barrier should not be raised, and the vehicle should not be allowed to enter.

The purpose of this report is to delve into the structure and implementation of this application, focusing on the individual classes, their responsibilities, and how they interact with one another. It also highlights the use of data structures and the object-oriented design patterns followed throughout the development.

This project was programmed using Java, with unit tests created using JUnit in order to test the application's methods.

Object-Oriented Design Principles

Inheritance

Inheritance is one of the fundamental Object-Oriented programming concepts that allows a new class to acquire the attributes and behaviours of an existing class, often referred to as the subclass and the superclass, or the child and parent class. This facilitates code reuse, reduces redundancy, and helps organise functionality into a logical hierarchy. Unique or additional features need to be explicitly defined in the subclass. The rest of the code is automatically included from the superclass and passed inherently to the subclasses. [1]

An example of inheritance being used in this program is the IDReader, BarcodeReader and NumberPlateReader classes. In this hierarchy, the IDReader abstract class serves as the parent class, while BarcodeReader and NumberPlateReader act as subclasses. IDReader defines the common interfaces and shared functionality, such as the IDReader() constructor and Read() method. Then both BarcodeReader and NumberPlateReader inherit from IDReader, reusing and extending their methods and properties while implementing their specialised behaviour on top. The idea of using inheritance here is particularly useful as it promotes code reuse and reduces duplication by allowing for shared functionality to be written once in the parent class. Additionally, it enhances the code's flexibility, maintainability and modularity by making it easier to introduce new types of readers in the future without modifying the existing code.

Interfaces

Interfaces in Object-Oriented Programming define a contract of operations that a class must support, without specifying how those operations should be implemented. The separation of this concern enhances code modularity and can improve clarity, supporting reusability in the process. Interfaces allow multiple classes to implement the same set of behaviours, promoting polymorphism and enabling a single function to operate across different object implementations. Interfaces also play a role in the design and specification process, acting as a contract and making systems easier to plan, design, extend and maintain. [2]

In the carpark system, interfaces are implemented through the Barrier interface. This barrier interface is created to specify the essential methods that all types of barriers must contain, such as raise(), lower() and a method to check whether the barrier is raised. Two separate classes – EntryBarrier and ExitBarrier - implement this interface and provide their own logic for each method. These classes are then called in the carpark class and are used if a new vehicle is entering or exiting the car park, and if so will be raised or lowered accordingly.

Abstract classes

Abstract classes are a key concept in Object-Oriented design, used to define a common structure for related subclasses while deferring some or all of the implementation details. They allow for shared methods, fields and constructors, encouraging code reuse and ensuring that all subclasses follow a consistent interface. Abstract classes also support polymorphism, enabling objects of different subclasses to be treated the same way, without needing to know what type each object is.

This concept is especially useful when there is a variety of classes that must follow a common structure but may differ in specific behaviour, enabling reusable and extendable designs. [3]

In the project code, an abstract class is used for the IDReader. This was done as it includes shared implementations for all the subclasses (BarcodeReader, NumberPlateReader) to use. More specifically the constructor sets up a Scanner object, which is a common functionality required by all types of ID readers. In Java, interfaces cannot have constructors or instance fields, so using an interface in this instance would require each subclass to duplicate the scanner setup. By using an abstract class this shared logic can be written once and used by each subclass. Also, the `Read()` abstract method ensures that each subclass is required to define its own reading behaviour, which allows for more flexibility. As the program may expand in the future, the abstract class allows for a foundation to add more shared methods for ID readers to use, making the design maintainable and more scalable.

Polymorphism

Polymorphism is one of the most crucial aspects of Object-Oriented Programming. It enables objects and methods to take on multiple forms, adapting depending on their context. By using polymorphism, a single interface can represent different forms of objects, providing flexibility to a program design. Methods can be defined in a parent class, and then overridden in their subclasses, enhancing code readability and reusability. Polymorphism also improves program adaptability, allowing developers to write code that can handle a variety of object types through shared interfaces, simplifying system design and creating more modular programs. [4]

In this program, polymorphism is implemented not only in the IDReader class but also in the Barrier class. The Barrier interface gets implemented by both the EntryBarrier and ExitBarrier, allowing them to be treated as a Barrier type. This enables flexibility in handling different types of barriers in the system, where the same methods – such as `raise()`, `lower()`, and `isRaised()` – can be used in both entry and exit barriers. Similarly, for the IDReader class, BarcodeReader and NumberPlateReader inherit from the IDReader abstract superclass, inheriting and overriding the abstract `Read()` method to provide their own specific implementations. This allows for the code to be more flexible and extensible, allowing new types of barriers or readers to be added in the future without needing to change the core logic in the system.

Generics

The final Object-Oriented programming concept worth highlighting is generics. Generics are a core Java feature that allows you to write flexible, type-safe code where types such as integers, strings or user-defined classes are treated as parameters. You can define a class, interface or method to operate on objects of various types without needing to risk compile-time type checking. The use of generics allows for code reusability, improving the maintainability of systems. It also confronts challenges such as the expression problem, a known dilemma where you measure to which degree can a set of operations be extended without the need to modify existing code, without needing to repeat code or without run-time type errors. To which the use of generics can help combat this problem, enabling safe and flexible operations across different data types. [5]

In the Carpark program, generics are used in the Parked class to create a flexible and reusable container that can hold any type of object. The Parked class includes several methods that facilitate operations on the stored items: `add()` adds an item to the list if it's not already present, `remove()` removes an item from the list, `contains()` checks if an item is in the list and returns a Boolean value, `size()` returns the number of items in the list, and `getAll()` returns a new list containing all the items. In the Carpark class, an instance of the Parked class is created and named `carsInside`, which is used to track the cars currently parked in the carpark. The program adds the string ID of each car to the list when it enters and removes it when it exits. Before adding a car, it checks if the car is already in the list using the `contains()` method to avoid duplicates. The use of generics allows the Parked class to handle different types of objects while maintaining type safety, providing a more flexible and efficient way to manage the carpark.

Data Structures and Memory Management

In the application, various data structures are used throughout. The primary ones worth highlighting are generics and array lists. As mentioned in the previous segment, generics are used in the Parked class to pass generic type parameters to hold any type of object. These objects are stored in an ArrayList. ArrayLists are a dynamic array that handles resizing automatically as objects get added or removed, these being the primary reasons why an ArrayList was chosen to be used here. The operations that the parked class is capable of handling are `add()`, `remove()`, `contains()`, `size()` and `getAll()`, each of these interacting with the ArrayList in various forms. Other data structures used in the application include Strings, primitive types such as `int` and `Boolean`, and Scanners. String is used for storing the car ID, whether they be barcodes or number plates, `int` is used for values such as capacity and `freeSpace` in the Carpark class and `Boolean` is used for the Barrier and FullSign classes to manage their on or off or raised or lowered states.

The application uses several memory management concepts, including heap memory, stack memory, and generics or type erasure. Heap memory is used for objects such as Carpark, Barrier, and FullSign which are created using the 'new' keyword. The objects are allocated memory on the heap that the in-built Java garbage collector handles when these objects are no longer referenced automatically. Stack memory is used when local variables such as String ID are called, which stores them in a stack. This memory is freed up automatically once the methods are finished being executed. Lastly, generics in Java are implemented by type erasure, meaning that the generic type information is removed during runtime. However, it does ensure type safety at compile time, reducing the need for casting and preventing ClassCastException errors. These exceptions typically occur when raw types are used without using carry type information, meaning that the compiler cannot ensure type safety, resulting in types of any kind being added, but if that element is later cast to the wrong type, then it will return that exception. [6]

Java mainly uses dynamic memory allocation for objects or collections by storing them on the heap at runtime, as mentioned earlier. This allows for more flexibility, especially for structures such as ArrayLists as they can grow as needed during runtime. Primitive types such as integers or Booleans are usually stored in the stack, where memory is managed more predictably with a fixed size, being a more static approach.

As mentioned previously the chosen data structure for storing car IDs inside the carpark is an ArrayList. This structure was selected due to its simplicity, dynamic sizing and ability to maintain the order of elements, making it helpful for such a small-scale application. ArrayLists provide a fast enough lookup and insertion when dealing with a relatively small number of entries, making it very practical in this situation. However, for a larger-scale application, it would be worth considering switching to an alternative such as a HashSet or HashMap, particularly when faster lookup times are important. These other structures offer constant-time performances for operations like `contains` and `remove` due to hashing, and while ArrayList is good for the current limited scope, moving to a HashSet in the future could improve efficiency and scalability, particularly as parked vehicle capacity grows.

Class Descriptions and Relationships

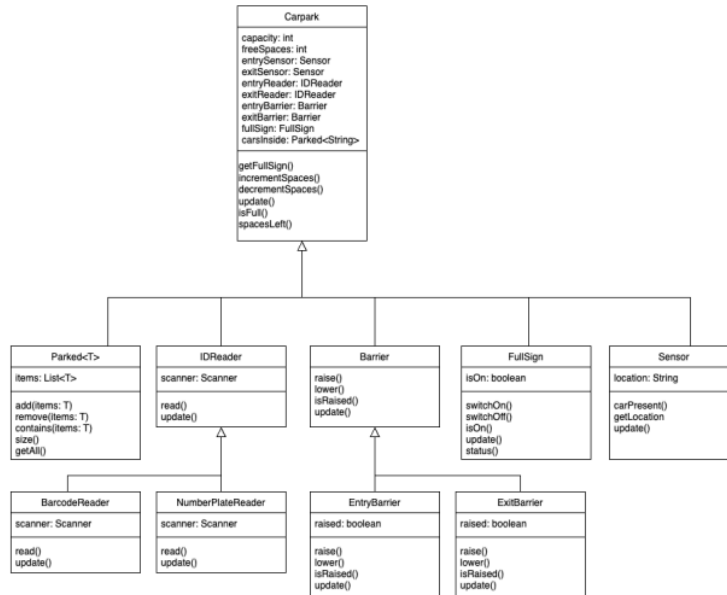


Diagram 1: Demonstration of the application's classes and relationships

In this section, each class will be described along with their contained attributes, methods and relationships with other classes. To begin the Carpark class will be looked at, this is because it is the central class that contains the most interactions with other classes. The attributes contained within this class are capacity, entry and exit sensors, barriers, IDReaders, fullSign and a Parsed list to track vehicles currently in the carpark. These attributes are called within the class methods. Among its methods, getFullSign(), simply returns the fullSign, while incrementSpaces() and decrementSpaces() manage the number of available spaces – ensuring the count stays within a valid range based on the carpark's defined capacity.

The most significant method is the update() method, which handles both entrance and exit logic. On the entrance side, the method checks for a vehicle using the entrance sensor and then retrieves the vehicle's ID using the entryReader. It then checks if the carpark is full and if the taken ID is already contained within the carpark, if not then the entranceBarrier will be raised and the ID will be added to the ArrayList of cars inside the carpark, then the number of free spaces will decrease by calling the decrementSpaces() method, after which the entranceBarrier will be lowered again after the car has entered the carpark. The same update() method also checks using the exit sensor to see if a vehicle is attempting to exit the carpark, it follows the same flow as the entrance sensor however it instead removes the car ID from the ArrayList if it is valid and already present in the carpark, if that ID is not in the ArrayList then it will not allow the car to exit as it is an unknown ID, however, if it is valid then the exit barrier will raise, the car will be removed, the number of free

spaces in the carpark will increase by calling the `incrementSpaces()` method and finally the exit barrier will lower.

After these checks, the `FullSign` class will check if it needs updating using the `updateFullSign()` method. If the car park is full then it will call the `FullSign` class and signal for it to switch on, otherwise, it will be switched off. The final two methods worth highlighting are only small but are still important, one is a Boolean method that is called to check if the available spaces left are equal to 0, if so then `isFull()` will equal true, then the other one is an integer method that returns the number of free spaces when `spacesLeft()` is called.

The `IDReader` class is an abstract class designed for reading vehicle IDs in the carpark system, with `BarcodeReader` and `NumberPlateReader` as its subclasses. Inside the `IDReader` class, there is a scanner attribute used for reading inputs and two abstract methods: `Read()`, which reads the vehicle's ID, and `update()` which doesn't currently get used for anything. These methods are then implemented by the subclasses to provide different logic for the barcodes or number plates. The `IDReader` class is important to the `Carpark` class as it allows the system to read vehicle IDs at the entrance and exit, making sure that cars are correctly tracked.

The `Sensor` class is responsible for detecting the presence of a vehicle at specific locations, such as the entrance or the exit of the carpark. Therefore, it contains a location attribute to specify where the sensor is placed. The `carPresent()` method prompts the user to input whether a car is present, returning a Boolean value based on the response. This class is used by the `Carpark` class to check if a car is present at the entrance or exit, allowing the system to respond appropriately updating the carpark capacity and managing the barriers.

The `Barrier` class is an interface which defines methods for raising and lowering barriers in the car park's entrance and exit. This class contains three methods: `raise()`, `lower()`, and `isRaised()`. The `EntryBarrier` and `ExitBarrier` classes implement this interface, allowing them to control their barrier's state. These classes interact with the `Carpark` class, raising and lowering the barriers as cars try to enter or exit the carpark.

The `FullSign` class is responsible for managing the display that indicates whether the car park is at capacity. It provides methods such as `SwitchOn()` and `SwitchOff()` to control whether the sign is on or off, depending on whether there are available spaces. The `Carpark` class interacts with `FullSign` through its `update()` method, which checks the car park's status and updates the full sign accordingly. If the car park is full, the `FullSign` is turned on, otherwise it remains off.

The final class is the `Parked` class. This class manages the list of vehicles currently inside the car park. It uses a generic `List<T>` to store vehicles, with `T` being the type of the object, which in the case of this application is a `String` to represent the vehicle IDs. The main responsibility of the class is to ensure no duplicates are added to the list of parked cars and to control the process of adding, removing and checking if a car is already parked. This class methods include `add()` which adds a vehicle to the list if it's not already inside, `remove()` which removes a vehicle from the list, `contains()` which checks if a vehicle is in the array list, `size()` which returns the number of vehicles in the list, used to see if the carpark is at capacity or not, and

`getAll()` which returns a list of all vehicles currently in the car park. `Parked` is used by the carpark class to track vehicles that have entered the carpark. When a car enters, the Carpark class adds the vehicle ID to the `Parked` list and when a car exits then it removes the ID from the list. Ensuring that each vehicle is only parked once, and removing only vehicles that are in the list, managing the cars in the carpark.

Testing

In this project, several unit tests have been created to test the functionality of the system and to catch various edge cases. These unit tests were created using JUnit. 28 unit tests were created across multiple classes, with the main focus of the tests being on the Carpark, EntryBarrier, ExitBarrier, FullSign, IDReader, Parked and Sensor classes. For the carpark class, whether a car enters or exits successfully was tested and whether the number of available spaces increments or decrements successfully. Additional tests checked whether the car park correctly identifies when it is full and prevents further entries, along with validating that the FullSign accurately reflects the car park's status. Edge cases were also covered here such as preventing vehicles from exiting when the carpark is empty and ensuring that the space count does not exceed the defined capacity, returning a message when full.

The barrier classes were tested to see if they correctly raise and lower as expected, including scenarios where if they are already raised if requested to raise again will remain raised and if they are already lowered will remain lowered when asked to be lowered again.

The sensor class was tested to see if it correctly identifies if a car is present using the input yes or no, as well as what will happen if a non-valid input is entered. When a non-valid input is entered then it does not recognise it as being a car present.

For the parked class, tests focused on preventing duplicate entries, verifying the correct addition and removal of vehicles, and handling attempts to remove vehicles that are not present correctly.

Lastly, the IDReader class and its subclasses (BarcodeReader and NumberPlateReader) were tested. These tests focused on verifying that both barcode and number plate IDs were able to read correctly. Additionally, the input trimming functionality was tested to ensure that any leading or trailing spaces in the input were properly removed before processing.

Conclusion

In conclusion, this project successfully met its goal of creating a functional carpark management application using key object-oriented concepts in the process. Each class played a strong role in making the system easy to maintain, test and grow, assisting the goal of making the system more modular in the process.

Unit testing with JUnit further strengthened the system by ensuring reliability and catching edge cases during development. These tests covered a range of different situations, from testing basic functionality to error handling and validation.

There are possible areas for improvement, however, such as switching to a HashSet for faster vehicle ID lookup, and optimising efficiency, particularly for a larger-scale system. For the current scope and usage, the existing implementation remains effective.

Bibliography

- [1] A. TAIVALSAARI, "On the Notion of Inheritance," Nokia Research Center, 1996.
- [2] P. S. Canning, W. R. Cook, W. L. Hill and W. G. Olthoff, "Interfaces for Strongly-Typed," Hewlett-Packard Laboratories, 1989.
- [3] B. Woolf, "The Abstract Class Pattern," Knowledge Systems Corp, 1997.
- [4] H. M. Ali, M. Y. Hamza and T. A. Rashid, "Exploring Polymorphism: Flexibility and Code Reusability in ObjectOriented Programming," ICRIE, Duhok, 2023.
- [5] M. Torgersen, "The Expression Problem Revisited," Computer Science Department University of Aarhus, Aarhus, 2004.
- [6] J. Niño, "The Cost of Erasure in Java Generics Type System," Consortium for Computing Sciences in Colleges, Evansville, IN, 2007.

FINAL GRADE

75/100

GENERAL COMMENTS

Engagement with Literature (5 marks): 3/5

Range & credibility of sources 1/2

Correct in-text citations & IEEE formatting 1/2

Evidence of critical reading 1/1

Knowledge & Understanding (20 marks): 14/20

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

Static vs dynamic memory explanation 5/7

Data-structure justification 5/7

Cognitive & Intellectual Skills (20 marks): 16/20

Critical evaluation of designs & literature 7/10

Justification of hypotheses/decisions 4/5

Synthesis of complex issues 5/5

Practical Application Skills (50 marks): 38/50

Class design & file organisation 7/10

Implementation of core simulation 6 / 10

Inheritance & IDReader subclasses 5/5

Dynamic data-structure coding 3/5

Use of OO concepts (abstract classes,

polymorphism,

generics,

inheritance,

interfaces) 8/ 10

Memory-based storage structures 4/5

Unit tests & harness 5/5

~~~~~

### **Transferable Skills (5 marks):4/5**

Report structure & clarity 1/2

Academic writing & organisation 1/1

User guide clarity 1/1

Video presentation quality 1/1

Comment:

Great work Lestyn

PAGE 2

---

PAGE 3

---

PAGE 4

---



### Comment 1

a minor thing, but put the reference before the fullstop

PAGE 5

---

PAGE 6

---

PAGE 7

---



### Comment 2

Nice

PAGE 8

---

PAGE 9

---

PAGE 10

---

PAGE 11

---

PAGE 12

---

PAGE 13

---



### Comment 3

Bibliography is not the same as a reference list,

References are works directly cited, but a bibliography can be resources read that influenced your report but not directly cited