

assignment report

by Kewei Deng

Submission date: 09-Apr-2024 03:47PM (UTC+0100)

Submission ID: 228904494

File name: 50137_Kewei_Deng_assignment_report_570693_1512387899.docx (733.83K)

Word count: 6171

Character count: 34451

1. Introduction

With the rapid development of computing technology and the complexity of software systems, OOP has become an important method to improve the quality of software design and development efficiency. The carpark application presented in this report utilizes OOP principles to solve a range of problems. The application shows the OOP in the design and implementation of functions, and how to simplify the problem through abstraction and modular design.

The purpose of this report is to show the complete process of developing an application using OOP methods, and its practical value and best practices in real project development. It will discuss in detail application requirements, design methods, specific implementation, software testing, and a brief user guide. The demo video link is <https://youtu.be/vAkRg5fVx-g>.

2. Requirements Analysis

2.1 Functional requirements

Based on the real practice of parking lots, the main functional requirements of carpark application include car entry and exit management, parking fee calculation and payment, parking status tracking.

Car entry and exit management refers to the fact that cars have to interact with the parts of the parking lot so that the application can record the entry and exit times of each car. A free parking space should be allocated to the car when it enters the parking lot and released when it exits the parking lot.

Parking fee calculation and payment refers to the calculation of the parking fee based on the length of time the car stays in the parking lot.

Parking space status tracking refers to monitoring the status of each parking space in real time, showing whether it is occupied or not, and showing whether the parking lot is full or not. If the parking lot is full, cars that want to enter are prompted to leave.

2.2 Performance requirements

The application must maintain data consistency at all times to prevent problems such as duplicate allocation of parking spaces and duplicate admission of drivers.

A good exception handling mechanism can improve the stability of the application and avoid system crashes caused by unhandled exceptions.

Properly manage system resources such as memory release to avoid resource leakage when exceptions occur, which directly affects system performance.

3. Application design and implementation

3.1 Carpark class

The carpark class is the core of carpark application, which is responsible for managing various operations of the parking lot. The attributes of carpark include capacity and free spaces. Carpark, meanwhile, consists of various components used in parking lots. These include sensors, barriers and their subclasses, full sign, ID reader, and their subclasses. The carpark class is linked to other classes by means of aggregation and composition. Composition is a strong ownership relationship, indicating that one object is part of another object and that the life cycles of the two are closely related^[1]. In the carpark application, composition is used to represent the relationship between carpark class and the inherent components mentioned above. Composition gives carpark full control and responsibility over the objects it directly contains. If a carpark object is initialized, the component objects are initialized as well, and if a carpark object is destroyed, the component objects it owns should also be destroyed. This relationship guarantees the consistency and integrity of the application, as well as reflects the shared lifecycle, which is a fundamental characteristic of a composition^[2].

Aggregation permits component sharing and reuse since it is a weaker representation of ownership^[3]. Through aggregate relations, the carpark class is linked to many classes, including driver, car, and parking. However, the life cycles of these classes are not

entirely dictated by carpark. This relationship makes sense in the actual world, where the driver, car, and parking can all exist without reference to the carpark.

When used together, the combination of composition and aggregation offers a strong and adaptable foundation for simulating real-world carpark. A complicated system's maintainability can be enhanced via aggregation and composition, which divide it into smaller, more manageable components. It is less disruptive to the rest of the system to alter or expand a component without affecting the entire system^[4]. The code snippet is attached in Figure 1.

```
class Carpark {
private:
    int capacity;
    int freeSpaces;
    vector<Sensor*> sensors;
    EntranceBarrier entranceBarrier;
    ExitBarrier exitBarrier;
    FullSign fullSign;
    IDBarcodeReader* idBarcodeReader;
    NumberPlateReader* numberPlateReader;
    unordered_map<string, Parking*> parking; // Keyed by driverID and numberPlate
    unordered_map<string, Sensor*> plateToSensorMap; // Mapping of number plate to Sensor
    unordered_map<string, Driver*> drivers; // Store and find Driver objects based on driver ID

public:
    Carpark(int cap);
    ~Carpark();
    //void update();
    void incrementSpaces();
    void decrementSpaces();
    int spacesLeft() const;
    void addDriver(Driver* driver); // add new driver
    Driver* findDriver(const string& driverID); // find Driver by driverID
    Parking* findParkingRecord(const string& key) const;
    vector<Parking*> getParkingByDriver(const string& driverID) const;
    vector<Parking*> getParkingByCar(const string& carPlate) const;

    void update(); // polling

    bool carArrivesAtEntrance(const Car& car, const Driver& driver);
    void carEnters(const Car& car);
    void associateSensorWithPlate(const Car& car, Sensor* sensor); // associate sensor with number plate
    void carEntersParkingSpot(const Car& car);
    void carLeavesParkingSpot(const Car& car);
    void carArrivesAtExit(const Car& car, Driver& driver);
    void carExits(const Car& car, const Driver& driver);
    void carLeavesExit(const Car& car);
};
```

Figure 1 Code of carpark.h

3.2 Sensor class

The sensors in the parking lot that keep track of whether a spot is occupied or not are represented by the sensor class. It is a concrete class that has attributes such as sensorID and carPresent. The code snippet of the sensor is shown in Figure 2.

```

#pragma once
#include <string>
using namespace std;

class Sensor {
private:
    string sensorID;
    bool carPresent;

public:
    Sensor(const string& id);
    string getSensorID() const;
    void update(bool presence); // Update sensor status
    bool isCarPresent() const; // Check if the car is in the parking space
};

```

Figure 2 Code of sensor.h

3.3 FullSign class

An indicator of a full parking lot is represented by a sign of the full sign class. The class has a key attribute, `isFull`, which indicates whether the parking lot is full. It contains methods such as `switchOn()` and `switchOff()` to change the indicator state. Full sign is a concrete class that directly reflects the state of the carpark, closely related to `carpark` class. The code snippet of the full sign is shown in Figure 3.

```

1  #pragma once
2  #include <string>
3
4  class Carpark;
5
6  class FullSign {
7  private:
8      bool lightOn;
9      Carpark* carpark;
10
11  public:
12      FullSign(Carpark* cp);
13      void switchOn();
14      void switchOff();
15      void update(bool full); // set light on
16      bool getLightOn() const;
17  };
18

```

Figure 3 Code of fullSign.h

3.4 IDReader class and its subclasses

ID reader is an abstract base class that defines a generic interface, `read()`, for reading driver or car identifiers such as barcodes or plate numbers. Its subclasses ID barcode reader and number plate reader implement this interface, both of which are concrete implementations of ID reader for reading different types of identifier information.

Inheritance and polymorphism in OOP are the keys to build a flexible, extensible system concept. Inheritance is a technique of using the definition of an existing class as the basis to establish a subclass. The definition of a subclass can add new attributions or new functions, or use the functions of the parent class, but it cannot selectively inherit the parent class^[5]. These features have been fully applied and reflected in the carpark application, which not only reduces the duplication of the code but also makes the code clearer and easier to manage.

Polymorphism is used when there is a hierarchy between classes and the classes are related by inheritance. According to polymorphism, the execution of a member function on a different function depends on the type of object calling it. Programmers can utilize a pointer or reference to a parent class type to point to a subclass object and invoke the appropriate method at run time due to polymorphism^[6]. This makes it possible for programmers to develop more general and flexible code by using virtual functions and dynamic binding.

Virtual methods are an essential technique for enabling polymorphism in OOP. They enable dynamic binding of object behavior, allowing the decision of which class's method to call at runtime^[7]. In C++, a virtual method is declared using the `virtual` keyword. If a derived class provides a method implementation with the same name as the base class, then the call of that method on the derived class object overrides the base class implementation^[8]. Through the use of inheritance, virtual methods, and polymorphism, the application can define a set of flexible and extensible frameworks, allowing the flexibility to use all kinds of components in different contexts. This design not only reduces the code redundancy but also improves the overall flexibility and maintainability of the application.

The ID barcode reader class is specifically designed to simulate the reading of the driver's ID barcode through keyboard input, enabling the system to handle barcode-based authentication. The number plate reader reads plate numbers via keypad input, providing plate-based identification and recording. These two classes will not work if the full sign lights up when the car intends to enter the parking lot. Simultaneously, they enable the carpark class to select the suitable reader at runtime based on particular requirements, reading without concern for specifics, thanks to polymorphism. The code snippets are attached as follows.

```
#pragma once
#include <iostream>
#include <string>
using namespace std;

class Carpark;

class IDReader {
protected:
    string serialNumber;
    Carpark* carpark;
public:
    IDReader(Carpark* cp) : carpark(cp) {}
    virtual ~IDReader() {} // destructor
    virtual string read() = 0;
    virtual void update() = 0;
    //virtual string getSerialNumber() const = 0;
};
```

Figure 4 Code of IDReader.h

```
1  #pragma once
2  #include "pch.h"
3  #include "IDReader.h"
4
5
6  //Inherited from IDReader, specifically used to read barcodes (driver ID)
7  class IDBarcodeReader : public IDReader {
8  public:
9      IDBarcodeReader(Carpark* cp) : IDReader(cp) {}
10     string read() override;
11     void update();
12     //string getSerialNumber() const;
13 };
14
15
```

Figure 5 Code of IDBarcodeReader.h

```

1  #pragma once
2  #include "IDReader.h"
3
4  //Inherited from IDReader, specifically used to read number plate
5  class NumberPlateReader : public IDReader {
6  public:
7      NumberPlateReader(Carpark* cp) : IDReader(cp) {}
8      string read() override;
9      void update();
10     //string getSerialNumber() const;
11 };
12

```

Figure 6 Code of NumberPlateReader.h

3.5 Barrier class and its subclasses

The barrier class defines the basic methods of barriers in the carpark, such as raising and lowering. Since real parking lots could need different kinds of barriers, this is an abstract base class. The subclasses entrance barrier and exit barrier, which demonstrate the use of inheritance, virtual methods, and polymorphism, implement the specific behavior. Subclasses can provide concrete implementation details by overriding virtual methods in the base class.

The entrance barrier class and exit barrier class inherit from the barrier class and implement the specific functionality of the entry and exit barriers. The entrance barrier is a barrier at the entrance of the carpark, and the full status of the carpark and the authentication results determine whether a car is allowed to enter, that is, the barrier is raised. It will not work when the carpark is full. When the carpark is not full and the driver's ID and number plate have been verified, the entrance barrier will rise to allow a car to enter. The exit barrier is the barrier that controls the exit, and its behavior usually depends on the payment state. Only after the driver has completed the parking fee payment, the exit barrier will be raised to allow the car to leave. The two subclasses show how to implement a specific through inheritance, virtual methods, and polymorphism control logic of barriers, and at the same time maintain close collaboration with carpark class. The code snippets are attached as follows.

```

#pragma once
#include <string>
#include <iostream>
using namespace std;

class Carpark;

class Barrier {
protected:
    bool up;
    Carpark* carpark; // To facilitate the interaction between Barrier and Carpark
public:
    Barrier(Carpark* cp) : carpark(cp), up(false) {}
    virtual ~Barrier() {} // deconstructor

    virtual void raise() = 0;
    virtual void lower() = 0;
    virtual void update() = 0; // update barrier
    bool isUp() { return up; };
};

```

Figure 7 Code of barrier.h

```

1  #pragma once
2  #include "Barrier.h"
3
4  //Inherited from Barrier, the barrier at the entrance.
5  class EntranceBarrier : public Barrier {
6  public:
7      EntranceBarrier(Carpark* cp);
8
9      void raise() override;
10     void lower() override;
11     void update() override;
12
13 };
14
15

```

Figure 8 Code of entranceBarrier.h

```

1  #pragma once
2  #include "pch.h"
3  #include "Barrier.h"
4
5  //Inherited from Barrier, the barrier at the exit
6  class ExitBarrier : public Barrier {
7  public:
8      ExitBarrier(Carpark* cp);
9
10     void raise() override;
11     void lower() override;
12     void update() override;
13 };
14

```

Figure 9 Code of exitBarrier.h

3.6 Driver class

Driver is the main body that interacts with the application. The driver class has attributes such as driver ID and account balance, and methods such as recharging and paying parking fees. The code snippet of the driver is shown in Figure 10.

```
1  #pragma once
2  #include <string>
3  #include "Parking.h"
4  #include <iostream>
5
6  using namespace std;
7
8
9  class Driver {
10 private:
11     string driverID;
12     double balance; // account balance
13
14 public:
15     Driver(const string& id, double balance);
16     string getDriverID() const;
17     void payParkingFee(double fee);
18     void recharge(double amount); // Recharge method
19     double getBalance() const;
20     void setBalance(double newBalance);
21 };
22
```

Figure 10 Code of driver.h

3.7 Car class

One of the main components of the parking lot operation is the car class, which in the application symbolizes the cars. The design and implementation of the car class mainly focuses on the basic properties of the car such as the number plate, so that it has a clear identity and function in the application. The code snippet is shown in Figure 11.

```
1  #pragma once
2  #include <string>
3  using namespace std;
4
5  class Car {
6  private:
7      string numberPlate;
8
9  public:
10     Car(const string& plate);
11     string getNumberPlate() const;
12 };
13
```

Figure 11 Code of car.h

3.8 Parking class

The parking class contains data about parking records, such as entry and exit time, number plate, driver ID, and other data. This concrete class is meant to monitor each parking space. The parking class has methods such as calculating parking time and fees, and mark as paid. The code snippet is shown in Figure 12.

```

1  #pragma once
2  #include <string>
3  #include <chrono>
4  using namespace std;
5  using namespace std::chrono;
6
7  class Parking {
8  private:
9      string parkingID;
10     string driverID;
11     string numberPlate;
12     string sensorID;
13     system_clock::time_point entryTime;
14     system_clock::time_point exitTime;
15     long duration;
16     double fee;
17     bool isPaid;
18
19 public:
20     Parking(const string& id, const string& driver, const string& plate);
21
22     void setEntryTime();
23     void setExitTime();
24     double calculateFee(); // only calculation result is returned
25     void markAsPaid();
26     bool getIsPaid() const;
27     long getDuration();
28
29     string getDriverID() const { return driverID; }
30     string getNumberPlate() const { return numberPlate; }
31     string getSensorID() const { return sensorID; }
32
33 };

```

Figure 12 Code of parking.h

The class diagram of carpark application is shown in Figure 13 Class Diagram.

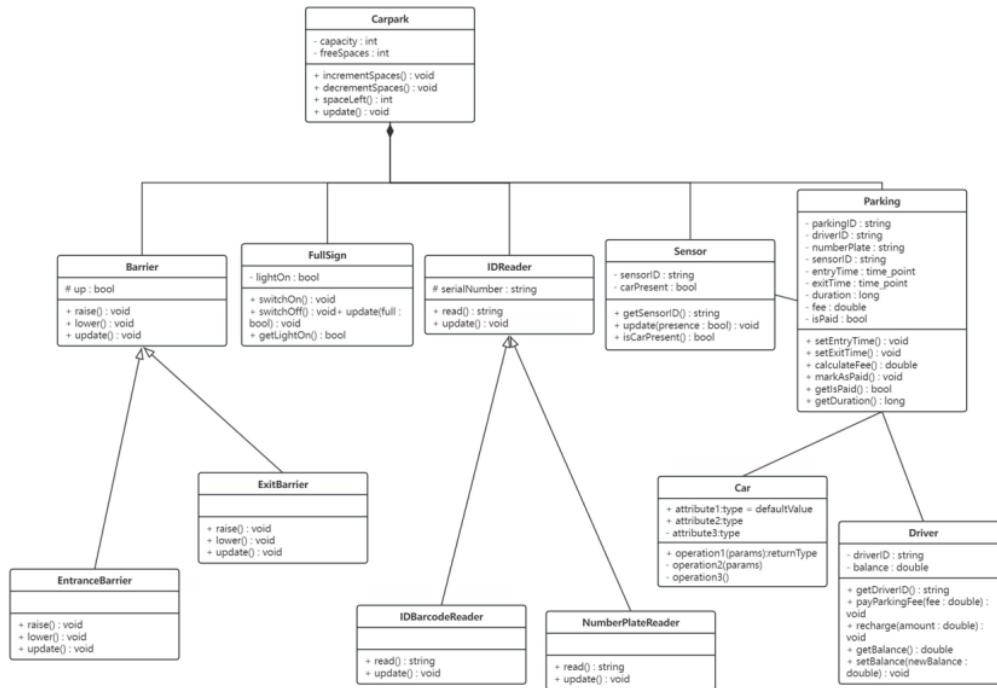


Figure 13 Class Diagram

3.9 Static memory allocation and dynamic memory allocation

Static memory is the fixed amount of memory allocated when a program is loaded^[9]. Programmers do not need to explicitly handle the allocation and release of static memory before a program begins because this is taken care of automatically by the operating system and compiler^[10]. The main advantage of static memory is easier to manage and less error-prone, but the flexibility is relatively low.

A fixed number of resources, such as a fixed number of sensors, barriers, ID readers, and full sign, can be managed using static memory allocation. In the carpark application, they usually just need to set amount of instances, which is why they are specified as member variables and use static memory allocation. This guarantees that these components are constantly available and streamlines memory management.

Dynamic memory is the memory space allocated by a program at runtime. The application can allocate any amount of memory space as needed because the

programmer controls its allocation and release^[11]. Dynamic memory allocation and release in C++ through the new and delete operators^[12]. The main advantage of dynamic memory is that it can dynamically allocate and release memory as needed, which is highly flexible, but it also requires programmers to manage memory themselves, and there is a risk of memory leaks and memory fragmentation^[13].

Objects whose number vary can be managed with dynamic memory allocation. Examples of such objects are dynamically added and removed drivers at different times, and parking records of cars entering and leaving the carpark. At the same time, STL containers such as vector and unordered_map can be used to conveniently implement dynamic memory management. In practical applications, to ensure the scalability of the carpark, the number of sensors may increase or decrease according to the actual demand, so the vector of dynamic memory allocation is selected to store the pointer of the sensor object, providing enough flexibility to dynamically manage the sensor. Mappings between arbitrary types are saved in the hash table unordered_map. The hash table support facilitates quick lookup, insert, and delete operations, making it perfect for maintaining vast volumes of dynamically changing data and offering a variety of rich manipulation capabilities for the data that is stored.

4. Test and Validate

The carpark application's unit test results are displayed in this chapter. Google Test was used as the testing framework.

To test the driver class, first confirm that the driver ID and balance are accurately set by the driver class constructor. After that, test the driver class's methods. SetBalance determines if the driver's balance may be appropriately set; Recharge verifies whether the driver's balance may be appropriately increased using the recharge technique; PayParkingFee checks to see if the driver's balance can accurately be deducted from the parking fee using the chosen payment method. The driver class test results are shown in Figure 14.

```

[ RUN      ] 4 tests from DriverTest
[ RUN      ] DriverTest.Properties
[ OK       ] DriverTest.Properties (0 ms)
[ RUN      ] DriverTest.SetBalance
[ OK       ] DriverTest.SetBalance (0 ms)
[ RUN      ] DriverTest.Recharge
Recharge successful. Your new balance is 100.
[ OK       ] DriverTest.Recharge (2 ms)
[ RUN      ] DriverTest.PayParkingFee
Payment successful. 30 has been deducted from your account. Your remaining balance is 70.
[ OK       ] DriverTest.PayParkingFee (0 ms)
[ RUN      ] 4 tests from DriverTest (18 ms total)

```

Figure 14 Unit tests of driver class

Carpark class testing starts by verifying that its constructor is initialized correctly. Then the function test is carried out, IncrementSpaces and DecrementSpaces test whether the number of free parking spaces is correctly increased or decreased when the car leaves or enters the carpark. The results are shown in Figure 15.

```

[ RUN      ] CarparkTest.ConstructorInitializesCorrectly
[ OK       ] CarparkTest.ConstructorInitializesCorrectly (1 ms)
[ RUN      ] CarparkTest.IncrementSpaces
Full Sign: Parking lot has space available.
Full Sign: Parking lot has space available.
[ OK       ] CarparkTest.IncrementSpaces (0 ms)
[ RUN      ] CarparkTest.DecrementSpaces
Full Sign: Parking lot has space available.
[ OK       ] CarparkTest.DecrementSpaces (0 ms)

```

Figure 15 Unit tests of carpark class

The tests for barrier and its subclasses are RaiseBarrier and LowerBarrier, which determines whether the barrier raising and lowering function is correct. The results are shown in Figure 16.

```

[-----] 4 tests from EntranceBarrierTest
[ RUN      ] EntranceBarrierTest.InitializesCorrectly
[ OK       ] EntranceBarrierTest.InitializesCorrectly (0 ms)
[ RUN      ] EntranceBarrierTest.RaiseBarrier

The entrance barrier has been raised.
[ OK       ] EntranceBarrierTest.RaiseBarrier (2 ms)
[ RUN      ] EntranceBarrierTest.LowerBarrier

The entrance barrier has been raised.

The entrance barrier has been lowered.
[ OK       ] EntranceBarrierTest.LowerBarrier (5 ms)
[ RUN      ] EntranceBarrierTest.UpdateDoesNotChangeState

The entrance barrier has been raised.
Entrance barrier updated.

The entrance barrier has been lowered.
Entrance barrier updated.
[ OK       ] EntranceBarrierTest.UpdateDoesNotChangeState (10 ms)
[-----] 4 tests from EntranceBarrierTest (26 ms total)

[-----] 4 tests from ExitBarrierTest
[ RUN      ] ExitBarrierTest.InitializesCorrectly
[ OK       ] ExitBarrierTest.InitializesCorrectly (0 ms)
[ RUN      ] ExitBarrierTest.RaiseBarrier

The exit barrier has been raised.
[ OK       ] ExitBarrierTest.RaiseBarrier (4 ms)
[ RUN      ] ExitBarrierTest.LowerBarrier

The exit barrier has been raised.

The exit barrier has been lowered.
[ OK       ] ExitBarrierTest.LowerBarrier (5 ms)
[ RUN      ] ExitBarrierTest.UpdateDoesNotChangeState

The exit barrier has been raised.
Exit barrier updated.

The exit barrier has been lowered.
Exit barrier updated.
[ OK       ] ExitBarrierTest.UpdateDoesNotChangeState (7 ms)
[-----] 4 tests from ExitBarrierTest (29 ms total)

```

Figure 16 Unit tests of subclasses of barrier

The test of Full Sign is mainly to test whether the full sign can be initialized correctly and whether the full sign of the parking lot can be updated correctly. The results are shown in Figure 17.

```

[-----] 5 tests from FullSignTest
[ RUN    ] FullSignTest.InitializesCorrectly
[ OK     ] FullSignTest.InitializesCorrectly (0 ms)
[ RUN    ] FullSignTest.UpdateToFull

Full Sign: Parking lot is full.
[ OK     ] FullSignTest.UpdateToFull (1 ms)
[ RUN    ] FullSignTest.UpdateToNotFull

Full Sign: Parking lot has space available.
[ OK     ] FullSignTest.UpdateToNotFull (4 ms)
[ RUN    ] FullSignTest.SwitchOn

Full Sign: Parking lot is full.
[ OK     ] FullSignTest.SwitchOn (2 ms)
[ RUN    ] FullSignTest.SwitchOff

Full Sign: Parking lot has space available.
[ OK     ] FullSignTest.SwitchOff (6 ms)
[-----] 5 tests from FullSignTest (26 ms total)

```

Figure 17 Unit tests of full sign class

Unit tests of IDReader and its subclasses are focused on whether it can be updated correctly. The results are shown in Figure 18.

```

[-----] 1 test from IDBarcodeReaderTest
[ RUN    ] IDBarcodeReaderTest.UpdateDoesNotCrash
ID barcode reader updated.
[ OK     ] IDBarcodeReaderTest.UpdateDoesNotCrash (0 ms)
[-----] 1 test from IDBarcodeReaderTest (4 ms total)

[-----] 1 test from NumberPlateReaderTest
[ RUN    ] NumberPlateReaderTest.UpdateDoesNotCrash
Number plate reader updated.
[ OK     ] NumberPlateReaderTest.UpdateDoesNotCrash (4 ms)
[-----] 1 test from NumberPlateReaderTest (5 ms total)

```

Figure 18 Unit tests of subclasses of ID reader

The test of the sensor class first verifies whether the sensor initialization is correct, and then tests whether the sensor correctly updates the status of parking space. The results are shown in Figure 19.

```

[-----] 3 tests from SensorTest
[ RUN    ] SensorTest.InitializesCorrectly
[ OK     ] SensorTest.InitializesCorrectly (0 ms)
[ RUN    ] SensorTest.UpdatePresence
[ OK     ] SensorTest.UpdatePresence (0 ms)
[ RUN    ] SensorTest.ReportsCorrectID
[ OK     ] SensorTest.ReportsCorrectID (0 ms)
[-----] 3 tests from SensorTest (7 ms total)

```

Figure 19 Unit tests of sensor class

The test of the parking class is to verify that it can correctly calculate the parking duration and update the payment status. The results are shown in Figure 20.

```
[-----] 2 tests from ParkingTest
[ RUN      ] ParkingTest.Duration
[          ] ParkingTest.Duration (1 ms)
[ RUN      ] ParkingTest.PayStatus
[          ] ParkingTest.PayStatus (0 ms)
[-----] 2 tests from ParkingTest (6 ms total)
```

Figure 20 Unit tests of parking class

The test of the car class is to verify that it can get the plate number correctly. The result is shown in Figure 21.

```
[-----] 1 test from CarTest
[ RUN      ] CarTest.NumberPlate
[          ] CarTest.NumberPlate (0 ms)
[-----] 1 test from CarTest (2 ms total)
```

Figure 21 Unit test of car class

As a result, all tests were passed and no functional abnormalities or defects were found.

5. User Guide

5.1 Operation guide

Find the executable file in carpark/carparkSimulator/x64/Debug/carparkSimulator.exe.

After starting the application, initialize the carpark and enter the capacity of the carpark.

In the main menu, enter 1 to enter the create instance menu, enter 2 to enter the simulated parking lot menu, and enter 3 to exit the application.

In the create instance menu, the user can enter 1 and 2 respectively to create the driver and car instances, that is, enter the driver ID, car number plate, and other information as prompted. Enter 3 to return to the main menu.

In the simulate carpark menu, enter 1 and 2 respectively to simulate entering and leaving the carpark, and enter 3 to return to the main menu.

After the correct input of the driver ID and number plate, the user can simulate the operation of the carpark. During simulating admission, if the full sign of the carpark

lights up, it means that the parking lot is full and cars cannot enter the parking lot at this time. When the parking lot is not full, the ID barcode reader and number plate reader are simulated, requiring the user to enter the corresponding driver ID and number plate on the keyboard to verify identity. Authentication can only be successful if the driver ID and number plate read match the driver ID and number plate information entered earlier. Subsequently, the car interacts with the entrance barrier of the parking lot. Then, the car finds the parking space and interacts with the sensor, indicating that the parking space is occupied. At this point, the parking is successful.

When simulating leaving the parking lot, the user first needs to enter the driver ID and number plate correctly and select the driver and car that need to be operated. When the car leaves the parking space, it interacts with the sensor, indicating that the parking space is released. When the car is driven to the exit, the ID barcode reader and number plate reader will authenticate. Then, the parking record will calculate the parking fee according to the parking duration. If the balance in the driver's account is insufficient, the driver will be asked to recharge until the balance is sufficient to pay the parking fee. After successful payment, the car interacts with the exit barrier of the parking lot, and when the exit barrier lowers, the car successfully leaves the parking lot.

5.2 Precautions

1. When creating a driver or car instance, re-entering existing information will cause the creation to fail.
2. When simulating entering and leaving the parking lot, the user must first enter the ID and number plate of the driver which the user wants to operate. If the user enters a non-existent driver ID or number plate, the carpark application will indicate that the input is invalid and will not carry out subsequent operations. When simulating entry, if the entered driver ID or number plate can be queried in the parking record; or when simulating exit, if the entered driver ID and number plate do not match those in the parking record, the carpark application will prompt an error, and no subsequent operations will be performed, thus maintaining data consistency.

3. When simulating the ID barcode reader and number plate reader, the user needs to enter the driver ID and number plate being operated on the keyboard. If the input is incorrect, the carpark application will prompt that the driving information does not match, causing the identity verification to fail.

6. Conclusion

This report provides a detailed analysis of the best practices of OOP core concepts in parking application development.

It first introduces the requirements of the parking application and identifies its main functions. Then it delves into key aspects of system design according to OOP concepts. Static and dynamic memory allocation are introduced in the report and applied in carpark applications.

The report ensures the stability and reliability of the application by presenting the results of the unit tests. It also provides a user guide to help users quickly get started with the system.

In summary, this report not only details the design and implementation process of the parking lot application but also deeply discusses the practice and theoretical practice of the core concepts of OOP.

7. References

- [1] El Hayat, Soumiya Ain, Fouad Toufik, and Mohamed Bahaj. "UML/OCL based design and the transition towards temporal object-relational database with bitemporal data." *Journal of King Saud University-Computer and Information Sciences* 32.4 (2020): 398-407.
- [2] Shah, Kruti P., and Emanuel S. Grant. "Towards Verification of UML Class Models using Formal Specification Methods: A Review." *Global Journal of Computer Science and Technology* 23.H1 (2023): 1-7.

- [3] Arunakumari, B. N., and Shivanand M. Handigund. "Automatic Design of Aggregation, Generalization and Specialization of Object-Oriented Paradigm Embedded in SRS." *Advances in Electrical and Computer Technologies: Select Proceedings of ICAECT 2019*. Springer Singapore, 2020.
- [4] Parsons, David, and David Parsons. "Objects Working Together: Association, Aggregation and Composition." *Foundational Java: Key Elements and Practical Programming* (2020): 141-175.
- [5] Mwangi, Waweru. "An object-oriented programming solution for managing inventory of used cars in Kenya using inheritance relationships." *Journal of Agriculture, Science and Technology* 22.3 (2023): 1-5.
- [6] Brusca, Victor G. "Encapsulation, Inheritance, and Polymorphism." *Introduction to Java Through Game Development: Learn Java Programming Skills by Working with Video Games*. Berkeley, CA: Apress, 2022. 189-222.
- [7] Sarcar, Vaskaran. "Inheritance." *Test Your Skills in C# Programming: Review and Analyze Important Features of C#*. Berkeley, CA: Apress, 2022. 111-146.
- [8] Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley Professional, 2022.
- [9] Zehra, Farzeen, et al. "Comparative analysis of C++ and python in terms of memory and time." (2020).
- [10] Chang, J. Morris, Woo Hyong Lee, and Witawas Srisa-an. "A study of the allocation behavior of C++ programs." *Journal of Systems and Software* 57.2 (2001): 107-118.
- [11] Kholmatov, Abrorjon, and Omonova Nazokatxon. "WORKING WITH POINTERS AND DYNAMIC MEMORY IN THE C++ PROGRAMMING LANGUAGE." *Ta'limning zamonaviy transformatsiyasi* 1.1 (2023): 746-755.
- [12] Gregoire, Marc. *Professional C++*. John Wiley & Sons, 2021.
- [13] Andrist, Bjorn, Viktor Sehr, and Ben Garney. *C++ high performance: master the art of optimizing the functioning of your C++ code*. Packt Publishing Ltd, 2020.

8. Appendix

// Since all of the class header files are displayed in Chapter 3, the appendix just provides source file codes and resource file codes.

```
//car.cpp
#include "pch.h"
#include "Car.h"

Car::Car(const string& plate) : numberPlate(plate) {}

string Car::getNumberPlate() const {
    return numberPlate;
}

//carpark.cpp
#include "pch.h"
#include "Carpark.h"
#include <iostream>

Carpark::Carpark(int cap)
    : capacity(cap),
    freeSpaces(cap),
    entranceBarrier(this),
    exitBarrier(this),
    fullSign(this) {
    idBarcodeReader = new IDBarcodeReader(this);
    numberPlateReader = new NumberPlateReader(this);

    // Initialize sensors, the number is consistent with capacity
    for (int i = 0; i < capacity; ++i) {
        sensors.push_back(new Sensor("Sensor_" + to_string(i + 1)));
    }
}

Carpark::~Carpark() {
    for (auto sensor : sensors) {
        delete sensor;
    }
    for (auto& pair : parking) {
        delete pair.second;
    }

    delete idBarcodeReader;
    delete numberPlateReader;
```

```

        for (auto& pair : drivers) {
            delete pair.second;
        }
    }

    void Carpark::update() {
        cout << "\nPolling carpark components" << endl;
        // Poll all sensors
        for (auto& sensor : sensors) {
            if (sensor->isCarPresent()) {
                cout << "Car detected at sensor: " << sensor->getSensorID() << endl;
            }
            else {
                cout << "No car at sensor ID: " << sensor->getSensorID() << endl;
            }
        }

        entranceBarrier.update();
        exitBarrier.update();
        idBarcodeReader->update();
        numberPlateReader->update();

        // Check if the parking lot is full, update FullSign status
        if (freeSpaces == 0) {
            fullSign.update(true);
        }
        else {
            fullSign.update(false);
        }
        cout << "" << endl;
    }

    void Carpark::incrementSpaces() {
        // Only increase the number of free parking spaces when the available parking
        spaces are less than the parking lot capacity
        if (freeSpaces < capacity) {
            freeSpaces++;
            // If the increased number of free parking spaces is greater than 0, ensure
            that fullSign is turned off.
            fullSign.update(false); // fullSign.update(false) means there are vacancies
        }
    }
}

```

```

void Carpark::decrementSpaces() {
    // Reduce the number of free parking spaces only when there are free parking
spaces
    if (freeSpaces > 0) {
        freeSpaces--;
        // If there are no free parking spaces after the reduction, fullSign should be
marked as true.
        fullSign.update(freeSpaces == 0);
    }
}

int Carpark::spacesLeft() const {
    return freeSpaces;
}

void Carpark::addDriver(Driver* driver) {
    if (driver != nullptr) {
        drivers[driver->getDriverID()] = driver;
    }
}

Driver* Carpark::findDriver(const string& driverID) {
    auto it = drivers.find(driverID);
    if (it != drivers.end()) {
        return it->second; // Found the corresponding Driver object
    }
    return nullptr; // No corresponding Driver object found
}

Parking* Carpark::findParkingRecord(const std::string& key) const {
    auto it = parking.find(key);
    if (it != parking.end()) {
        return it->second;
    }
    else {
        return nullptr;
    }
}

vector<Parking*> Carpark::getParkingByDriver(const string& driverID) const {
    vector<Parking*> driverParkings;
    for (const auto& kv : parking) {
        if (kv.first.find(driverID) != string::npos) { // key contains driverID

```

```

        driverParkings.push_back(kv.second);
    }
}
return driverParkings;
}

vector<Parking*> Carpark::getParkingByCar(const string& carPlate) const {
    vector<Parking*> carParkings;
    for (const auto& kv : parking) {
        if (kv.first.find(carPlate) != string::npos) { // key contains number plate
            carParkings.push_back(kv.second);
        }
    }
    return carParkings;
}

bool Carpark::carArrivesAtEntrance(const Car& car, const Driver& driver) {
    if (fullSign.getLightOn()) {
        cout << "Parking lot is full. " << car.getNumberPlate() << " cannot enter."
        << endl;
        return false; // The parking lot is full and the car cannot enter
    }
    else {
        // Directly use Driver object and Car object for verification
        string driverID = driver.getDriverID();
        string plate = car.getNumberPlate();

        // Use IDReader to read information for secondary verification.
        string readDriverID = idBarcodeReader->read();
        string readPlate = numberPlateReader->read();

        if (driverID == readDriverID && plate == readPlate) {
            string key = driverID + "_" + plate;
            if (parking.find(key) == parking.end()) { // Make sure there are no
duplicate records
                Parking* newParking = new Parking(key, driver.getDriverID(),
car.getNumberPlate());
                newParking->setEntryTime();
                parking[key] = newParking;
                decrementSpaces();
                entranceBarrier.raise();
                cout << "\n" << car.getNumberPlate() << " enters the parking
lot." << endl;

```

```

        return true; // car enters the parking lot successfully
    }
    else {
        cout << "Error: A parking record for " << car.getNumberPlate()
<< " already exists." << endl;
        return false;
    }
}
else {
    cout << "Error: The entered information does not match the car and the
driver attempting to enter." << endl;
    return false;
}
}
}

void Carpark::carEnters(const Car& car) {
    entranceBarrier.lower();
    cout << "\n" << car.getNumberPlate() << " has passed the entrance barrier." <<
endl;
}

void Carpark::associateSensorWithPlate(const Car& car, Sensor* sensor) {
    string numberPlate = car.getNumberPlate();
    plateToSensorMap[numberPlate] = sensor;
}

void Carpark::carEntersParkingSpot(const Car& car) {
    string numberPlate = car.getNumberPlate();
    // Find the first empty parking space where the sensor carPresent is false
    for (auto& sensor : sensors) {
        if (!sensor->isCarPresent()) {
            sensor->update(true); // Update the sensor status to indicate that the
parking space is occupied
            plateToSensorMap[numberPlate] = sensor; // Associate license plate
number with sensor
            cout << "\n" << numberPlate << " has parked in spot " <<
sensor->getSensorID() << "." << endl;
            return; // Parking successful, exit method
        }
    }
}
}

```

```

void Carpark::carLeavesParkingSpot(const Car& car) {
    string numberPlate = car.getNumberPlate();
    auto sensorIt = plateToSensorMap.find(numberPlate);
    if (sensorIt != plateToSensorMap.end()) {
        sensorIt->second->update(false); // update carPresent as false
        cout << "\n" << numberPlate << " leaves the parking spot." << endl;
    }
}

void Carpark::carArrivesAtExit(const Car& car, Driver& driver) {
    // Directly use Driver object and Car object for verification
    string driverID = driver.getDriverID();
    string plate = car.getNumberPlate();

    // Use IDReader to read information for secondary verification.
    string readDriverID = idBarcodeReader->read(); // read id barcode
    string readPlate = numberPlateReader->read(); // read number plate

    if (driverID == readDriverID && plate == readPlate) {
        string key = driverID + "_" + plate;
        auto parkingIt = parking.find(key);

        // The corresponding parking record is found
        Parking* parkingRecord = parkingIt->second;
        parkingRecord->setExitTime();
        double fee = parkingRecord->calculateFee();
        long duration = parkingRecord->getDuration();
        cout << "\nParking duration is " << duration << " seconds" << endl;
        cout << "\nParking fee for " << plate << " is " << fee << " pounds. Please
pay." << endl;

        // The driver is required to pay for parking.
        driver.payParkingFee(fee);

        // Update the isPaid status
        parkingRecord->markAsPaid();

        // Payments are always processed successfully
        cout << "\nPayment processed for " << plate << "." << endl;
    }
    else {
        cout << "\nError: The entered information does not match the car and the
driver attempting to leave." << endl;
    }
}

```

```

        carEntersParkingSpot(car);
    }

}

void Carpark::carExits(const Car& car, const Driver& driver) {
    string key = driver.getDriverID() + "_" + car.getNumberPlate();
    auto it = parking.find(key);
    if (it != parking.end() && it->second->getIsPaid()) {
        exitBarrier.raise();
        cout << "\n" << car.getNumberPlate() << " exits the parking lot." << endl;
        incrementSpaces();
        delete it->second; // Clean up
        parking.erase(it);
    }
}

void Carpark::carLeavesExit(const Car& car) {
    cout << "\n" << car.getNumberPlate() << " has passed the exit barrier." << endl;
    exitBarrier.lower();
}

//driver.cpp
#include "pch.h"
#include "Driver.h"

Driver::Driver(const string& id, double balance) : driverID(id), balance(balance) {}

string Driver::getDriverID() const {
    return driverID;
}

double Driver::getBalance() const {
    return balance;
}

void Driver::setBalance(double newBalance) {
    balance = newBalance;
}

```

```

};

void Driver::payParkingFee(double fee) {
    while (balance < fee) {
        cout << "Insufficient balance. Your current balance is " << balance << "."
<< endl;
        // If the balance is insufficient, guide the driver to recharge.
        double amountToRecharge = 0.0;
        cout << "Please enter the amount to recharge: ";
        cin >> amountToRecharge;
        recharge(amountToRecharge);

        // Check whether the balance is sufficient
        if (balance >= fee) {
            break;
        }
    }

    // The balance must be sufficient at this time
    balance -= fee;
    cout << "Payment successful. " << fee << " has been deducted from your
account. Your remaining balance is " << balance << "." << endl;
}

void Driver::recharge(double amount) {
    if (amount > 0) {
        balance += amount;
        setBalance(balance);
        cout << "Recharge successful. Your new balance is " << balance << "." <<
endl;
    }
    else {
        cout << "Invalid recharge amount." << endl;
    }
}

// entranceBarrier.cpp
#include "pch.h"
#include "EntranceBarrier.h"

EntranceBarrier::EntranceBarrier(Carpark* cp) : Barrier(cp) {}

```

```

void EntranceBarrier::raise() {
    if (!up) {
        up = true;
        cout << "\nThe entrance barrier has been raised." << endl;
    }
}

void EntranceBarrier::lower() {
    if (up) {
        up = false;
        cout << "\nThe entrance barrier has been lowered." << endl;
    }
}

void EntranceBarrier::update() {
    cout << "Entrance barrier updated." << endl;
}

// exitBarrier.cpp
#include "pch.h"
#include "ExitBarrier.h"

ExitBarrier::ExitBarrier(Carpark* cp) : Barrier(cp) {}

void ExitBarrier::raise() {
    if (!up) {
        up = true;
        cout << "\nThe exit barrier has been raised." << endl;
    }
}

void ExitBarrier::lower() {
    if (up) {
        up = false;
        cout << "\nThe exit barrier has been lowered." << endl;
    }
}

void ExitBarrier::update() {
    cout << "Exit barrier updated." << endl;
}

```

```

// fullSign.cpp
#include "pch.h"
#include "FullSign.h"
#include "Carpark.h"
#include <iostream>

using namespace std;

FullSign::FullSign(Carpark* cp) : carpark(cp), lightOn(false) {}

void FullSign::switchOn() {
    cout << "\nFull Sign: Parking lot is full." << endl;
}

void FullSign::switchOff() {
    cout << "\nFull Sign: Parking lot has space available." << endl;
}

void FullSign::update(bool full) {
    lightOn = full;
    if (lightOn) {
        switchOn();
    }
    else {
        switchOff();
    }
}

bool FullSign::getLightOn() const {
    return lightOn;
}

//idBarcodeReader.cpp
#include "pch.h"
#include "IDBarcodeReader.h"

string IDBarcodeReader::read() {
    cout << "To simulate the ID barcode reader's read action, please input the ID
barcode: ";
    string barcode;
    cin >> barcode;
    return barcode;
}

```

```

void IDBarcodeReader::update() {
    cout << "ID barcode reader updated." << endl;
}

//numberPlateReader.cpp
#include "pch.h"
#include "NumberPlateReader.h"

string NumberPlateReader::read() {
    cout << "To simulate the number plate reader's read action, please input the
number plate: ";
    string numberPlate;
    cin >> numberPlate;
    return numberPlate;
}

void NumberPlateReader::update() {
    cout << "Number plate reader updated." << endl;
}

//parking.cpp
#include "pch.h"
#include "Parking.h"

Parking::Parking(const string& id, const string& driver, const string& plate)
    : parkingID(id), driverID(driver), numberPlate(plate), fee(0.0), isPaid(false) {
    setEntryTime();
}

void Parking::setEntryTime() {
    entryTime = system_clock::now();
}

void Parking::setExitTime() {
    exitTime = system_clock::now();
    duration = duration_cast<seconds>(exitTime - entryTime).count();
}

long Parking::getDuration() {
    return duration;
}

```

```

double Parking::calculateFee() {
    long parkedDuration = getDuration();
    double minutes = parkedDuration / 60.0;
    fee = minutes * 5.0; // 5 pounds per minute
    return fee;
}

void Parking::markAsPaid() {
    isPaid = true;
}

bool Parking::getIsPaid() const {
    return isPaid;
}

//sensor.cpp
#include "pch.h"
#include "Sensor.h"

Sensor::Sensor(const string& id) : sensorID(id), carPresent(false) {}

string Sensor::getSensorID() const{
    return sensorID;
}

void Sensor::update(bool presence) {
    carPresent = presence;
}

bool Sensor::isCarPresent() const {
    return carPresent;
}

//main.cpp
#include <iostream>
#include <unordered_map>
#include "Carpark.h"
#include "Driver.h"
#include "Car.h"

```

```

unordered_map<string, Driver*> drivers;
unordered_map<string, Car*> cars;
Carpark* carpark = nullptr;

void createInstanceMenu();
void simulateCarparkMenu();
void entranceSubMenu();
void exitSubMenu();

int main() {
    int capacity;
    cout << "Enter the capacity of the car park: ";
    cin >> capacity;
    carpark = new Carpark(capacity);

    drivers["ava"] = new Driver("ava", 0);
    drivers["zoe"] = new Driver("zoe", 0);
    drivers["sam"] = new Driver("sam", 0);

    cars["car1"] = new Car("car1");
    cars["car2"] = new Car("car2");
    cars["car3"] = new Car("car3");

    cout << "\nThree default initial Driver and Car instances have been created
    respectively." << endl;
    cout << "\nDriver:\nID: ava, balance: 0\nID: zoe, balance: 0\nID: sam, balance:
    0" << endl;
    cout << "\nCar:\nplate: car1\nplate: car2\nplate: car3" << endl;

    int command = 0;
    while (command != 3) {
        cout << "\nMain Menu:\n";
        cout << "1. Create Instances\n";
        cout << "2. Simulate Carpark\n";
        cout << "3. Quit\n";
        cout << "Enter your choice: ";
        cin >> command;

        if (cin.fail()) {
            cin.clear(); // clear error status
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
            remaining input on the current line
            cout << "Invalid choice. Please try again.\n";
        }
    }
}

```

```

        continue; // Skip the remainder of the current loop and start the loop
again
    }

    switch (command) {
    case 1:
        createInstanceMenu();
        break;
    case 2:
        simulateCarparkMenu();
        break;
    case 3:
        cout << "Exiting program...\n";
        break;
    default:
        cout << "Invalid choice. Please try again.\n";
    }
}

// Cleanup dynamically allocated resources
delete carpark;
for (auto& pair : drivers) delete pair.second;
for (auto& pair : cars) delete pair.second;

return 0;
}

void createInstanceMenu() {
    int command = 0;
    while (command != 3) {
        cout << "\nCreate Instance Menu:\n";
        cout << "1. Create Driver\n";
        cout << "2. Create Car\n";
        cout << "3. Back to Main Menu\n";
        cout << "Enter your choice: ";
        cin >> command;

        if (cin.fail()) {
            cin.clear(); // clear error status
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
remaining input on the current line
            cout << "Invalid choice. Please try again.\n";
            continue; // Skip the remainder of the current loop and start the loop
again
        }
    }
}

```

```

    }

    string id;
    double balance;
    switch (command) {
    case 1:
        cout << "Enter driver ID: ";
        cin >> id;
        if (drivers.find(id) != drivers.end()) {
            // if find existing driver UD
            cout << "Driver ID already exists. Please try a different ID.\n";
        }
        else {
            cout << "Enter initial balance: ";
            cin >> balance;
            drivers[id] = new Driver(id, balance);
            cout << "Driver created successfully.\n";
        }
        break;
    case 2:
        cout << "Enter car number plate: ";
        cin >> id;
        if (cars.find(id) != cars.end()) {
            // if find existing number plate
            cout << "Car number plate already exists. Please try a different
number plate.\n";
        }
        else {
            cars[id] = new Car(id);
            cout << "Car created successfully.\n";
        }
        break;
    case 3:
        return;
    default:
        cout << "Invalid choice. Please try again.\n";
    }
}
}

```

```

void simulateCarparkMenu() {
    // Ensure there's at least one driver and one car
    if (drivers.empty() || cars.empty()) {

```

```

        cout << "\nYou need at least one driver and one car to simulate the
carpark.\n";
        return;
    }

    int command = 0;
    while (command != 3) {
        cout << "\nSimulate Carpark Menu:\n";
        cout << "1. Entrance\n";
        cout << "2. Exit\n";
        cout << "3. Back to Main Menu\n";
        cout << "Enter your choice: ";
        cin >> command;

        if (cin.fail()) {
            cin.clear(); // clear error status
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Ignore
remaining input on the current line
            cout << "Invalid choice. Please try again.\n";
            continue; // Skip
        }

        switch (command) {
            case 1:
                entranceSubMenu();
                break;
            case 2:
                exitSubMenu();
                break;
            case 3:
                return;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }
}

void entranceSubMenu() {
    string driverID, plate;
    cout << "Enter driver ID: ";
    cin >> driverID;
    cout << "Enter number plate: ";
    cin >> plate;
}

```

```

    if (drivers.find(driverID) == drivers.end() || cars.find(plate) == cars.end()) {
        cout << "Invalid driver ID or number plate. Returning to previous menu.\n";
        return;
    }

    // Get all parking records related to driver ID
    auto driverParkings = carpark->getParkingByDriver(driverID);
    if (!driverParkings.empty()) {
        cout << "A parking record for driver " << driverID << " already exists.
Returning to previous menu.\n";
        return; // If there is a driver's parking record, return to the previous menu
    }

    // Get all parking records related to number plate
    auto carParkings = carpark->getParkingByCar(plate);
    if (!carParkings.empty()) {
        cout << "A parking record for car " << plate << " already exists. Returning
to previous menu.\n";
        return; // If there is a car's parking record, return to the previous menu
    }

    Driver* driver = drivers[driverID];
    Car* car = cars[plate];

    // Simulate entrance process
    cout << "\nSimulating entrance for " << driverID << " with car " << plate <<
".\n";

    // This indicates whether the car can successfully enter
    bool carEnteredSuccessfully = false;

    // Simulate the car arriving at the entrance
    // update carEnteredSuccessfully based on the operation results
    carEnteredSuccessfully = carpark->carArrivesAtEntrance(*car, *driver);

    if (carEnteredSuccessfully) {
        carpark->carEnters(*car);
        carpark->carEntersParkingSpot(*car);
    }
    else {
        cout << "Car entry failed." << endl;
    }

    carpark->update();

```

```

}

void exitSubMenu() {
    string driverID, plate;
    cout << "Enter driver ID: ";
    cin >> driverID;
    cout << "Enter number plate: ";
    cin >> plate;

    Driver* driver = drivers[driverID];
    Car* car = cars[plate];

    if (drivers.find(driverID) == drivers.end() || cars.find(plate) == cars.end()) {
        cout << "Invalid driver ID or number plate. Returning to previous menu.\n";
        return;
    }

    // Construct the key of the parking record
    string key = driverID + "_" + plate;

    // Check whether there is a corresponding parking record
    Parking* parkingRecord = carpark->findParkingRecord(key);
    if (parkingRecord == nullptr) {
        cout << "No parking record found for " << driverID << " with car " << plate
        << ". Returning to previous menu.\n";
        return; // If the parking record is not found, return to the previous menu
    }

    cout << "\nSimulating exit for " << driverID << " with car " << plate << ".\n";

    carpark->carLeavesParkingSpot(*car);
    carpark->carArrivesAtExit(*car, *driver);
    carpark->carExits(*car, *driver);
    carpark->carLeavesExit(*car);
    carpark->update();
}

```

assignment report

GRADEMARK REPORT

FINAL GRADE

75 / 100

GENERAL COMMENTS

In general, a well written report. The structure is a little unusual but overall it reads quite well. References are provided and are relevant. They are used to support arguments/support assertions appropriately in the body of the report.

The class diagram shows how the basic structure of the car park has been extended to included new and useful components. Good.

The discussion and analysis of OO concepts and techniques used forms part of the design discussion and makes clear arguments for the use of OO concepts. Perhaps a little more discussion of use of the vector class and related dynamic container objects would help here.

A discussion of the composition relationship between the carpark and its component classes, especially with regard to the use of a pointer to the carpark object it self would help. That is, explain why you need to instantiate a pointer to a carpark in a component class which belongs to the carpark. There are easier ways. Explain this!

Use of inheritance, virtual methods, etc is otherwise appropriate. Separation of classes into .hpp and .cpp files is consistently done.

Some further discussion of constructors/destructors in the report could help.

Testing is, in general, well done.

Use of screen shots is helpful and works well with the video demo.

Overall a good effort.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

GRADING FORM: SMU GENERIC_GJ

KEWEI DENG

0

RESEARCH-INFORMED LITERATURE

KNOWLEDGE AND UNDERSTANDING

ANALYSIS

PRACTICAL APPLICATION AND DEP.

SKILLS FOR PROFESSIONAL PRAC.