Parking system design and implementation

Name:Yijun Kuang

Abstruct : This paper designs and implements a parking lot system with functional modules such as vehicle information recognition with bar code recognition and license plate number recognition, vehicle entry and exit management, and signal light management, demonstrating the effectiveness of object-oriented design principles and patterns in practical applications. The system provides a reliable foundation.

Table of Contents

1.Introduction.....1

2 System Design and Implementation.....2

2.1System Framework.....2

2.2 Functional Requirements.....2

2.3 Class design.....3

2.4 Application of OOP Concepts.....4

3.Performance and Memory Management Analysis.....5

3.1 Memory Usage Analysis 5

3.2 Performance Advantage Analysis.....6

4.System Implementation and Unit Testing.....6

5.User Guide.....10

6.Conclusion.....11

Reference

1.Introduction

Parking lot management is a critical aspect of urban life, especially in densely populated areas where the availability of parking spaces is limited. To address the challenges associated with parking lot management, an object-oriented approach can be employed to design and develop an efficient parking lot management system. This project aims to create a simulation system using C++ that mimics the operations of a real-world parking lot, focusing on vehicle entry and exit management, parking space allocation, and security measures.

Vehicle Entry and Exit Management:The parking lot simulation system will facilitate the seamless entry and exit of vehicles. Upon entry, the system will record the time and parking position of the vehicle. Similarly, when a vehicle exits the parking lot, the system will update the vehicle's exit time and release the occupied parking space for future use. This functionality ensures accurate tracking of vehicle movements within the parking lot.

Parking Space Management:Parking Space Management: Effective management of parking spaces is essential to optimize the utilization of available resources. The parking lot management system will dynamically allocate parking spaces based on availability. It will maintain a record of occupied and vacant parking spaces, allowing for efficient parking space allocation. Additionally, the system will implement strategies to handle situations such as full parking lots, prioritizing access for different types of vehicles based on predefined criteria.

Security and Surveillance:Security and Surveillance: Ensuring the security of the parking lot is paramount to prevent unauthorized access and mitigate potential risks. The parking lot simulation system will incorporate security features such as surveillance cameras and alarm systems. These measures will help monitor the parking lot environment in real-time, detect any suspicious activities, and alert authorities in case of security breaches. By enhancing security measures, the system aims to create a safe and secure parking environment for vehicle owners.

The project will utilize an object-oriented design methodology to model the parking lot system. Various classes will represent entities such as vehicles, parking spaces, and the parking lot itself. These classes will encapsulate data and behavior related to their respective entities, promoting modularity and code reusability. Additionally, interactions between classes will simulate the dynamic operations of the parking lot, allowing for a comprehensive representation of its functionalities.

In conclusion, the development of an object-oriented parking lot management system offers a practical solution to the complex challenges associated with urban parking. By simulating vehicle entry and exit processes, managing parking space allocation, and implementing security measures, the system aims to streamline parking lot operations and enhance overall efficiency. Through this project, valuable insights into object-oriented programming concepts will be gained, and innovative solutions for parking lot management will be explored, contributing to the improvement of urban infrastructure and services.

2.System Design and Implementation

2.1 System Framework

This paper utilizes C++ and Visual Studio 2022 to construct a parking lot system. The system primarily facilitates the entire process of vehicles entering and exiting the parking lot, which involves identifying vehicle identity information, managing vehicle entry and exit, and tracking the number of parking spaces in the parking lot. The flow for users entering and exiting the parking lot is as follows:

1.Check if the indicator light is red, indicating available parking spaces.
2.Vehicle approaches the barrier.
3.Vehicle information is recognized using an ID barcode reader or license plate input.
4.If the information is correct, the barrier opens; if incorrect, the barrier remains closed to prevent entry.
5.Upon correct information, the vehicle enters the parking lot and parks.
6.Vehicle approaches the barrier at the parking lot exit.
7.Barrier opens, allowing the vehicle to exit the parking lot.

In the parking lot system, the flow for parking management is outlined as follows:

1.Detection of vehicle approaching the barrier.
2.Check if the indicator light is green, indicating available parking spaces. If available, continue; otherwise, close the barrier.
3.Verify input vehicle information against the database.
4.If vehicle data is correct, open the barrier; otherwise, keep the barrier closed.
5.Once the vehicle passes through the barrier, close it.
6.Update parking lot occupancy and update the indicator light.
7.Detection of vehicle approaching the exit.
8.Open the barrier.
9.Verify vehicle has left the parking lot.
10.Close the barrier.
11.Update parking lot occupancy and update the indicator light.

2.2 Functional Requirements

Vehicle Information Recognition and Management: Prior recognition of vehicle information is necessary for vehicles entering the parking lot. The management system should feature license plate recognition and ID barcode reading capabilities. This functionality enhances the efficiency and security of vehicle entry and exit, effectively preventing unauthorized access.

Signal Light Management: Signal light management is a crucial aspect of parking lot management systems. It indicates the availability of parking spaces, providing convenient parking information to drivers. By implementing signal light management, the parking lot management system becomes more intelligent and user-friendly, offering a better parking experience for drivers and providing more effective tools for parking lot management.

2.3 Class design

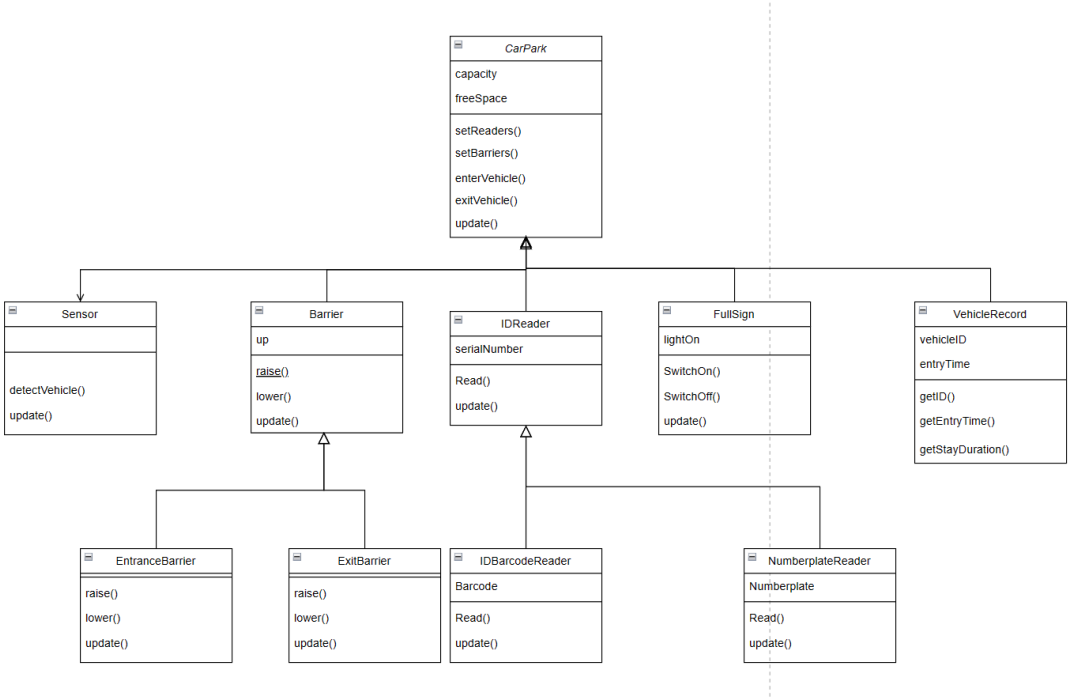

Fig.1 Class design

Barrier: This class manages barriers in the parking lot system, primarily controlling the opening and closing of barriers at entrances and exits to allow vehicle passage. It has three methods: raise() to raise the barrier, lower() to lower the barrier, and update() to update the status of the barrier.

CarPark: The core class of the parking lot system, responsible for managing the entire operation of the parking lot. It handles vehicle entry and exit, management of available parking spaces, control of barriers, and updating of the full sign. It has three methods: enterVehicle() to determine if a vehicle can enter the parking lot, exitVehicle() to process a vehicle leaving, and update() to update the status of the parking lot.

Sensor: Used to detect whether vehicles have arrived at the entrance or exit. The method detectVehicle() checks if a vehicle is detected and returns a boolean indicating the result.

FullSign: Manages the display of the parking lot full sign. It has three methods: switchOn() to turn on the full sign, switchOff() to turn off the full sign, and update() to return whether the full sign is currently displayed.

VehicleRecord: Records detailed information about each vehicle, such as the license plate number and entry time. It has three main methods: getID() to return the vehicle's identifier, getEntryTime() to return the vehicle's entry time, and getStayDuration() to calculate and return the vehicle's duration of stay.

EntranceBarrier: A concrete implementation of Barrier, used to control the entrance barrier.

It inherits the raise() and lower() methods to raise and lower the entrance barrier, respectively.

ExitBarrier: A concrete implementation of Barrier, used to control the exit barrier. It inherits the raise() and lower() methods to raise and lower the exit barrier, respectively.

IDReader: An abstract class defining the interface for reading identifiers such as barcodes or license plate numbers. It has a method readID() to read and return the identifier.

IDBarcodeReader: A concrete implementation of IDReader, used to read barcodes. It inherits the readID() method to read and return barcodes.

NumberplateReader: A concrete implementation of IDReader, used to read license plate numbers. It inherits the readID() method to read and return license plate numbers.

2.4 Application of OOP Concepts

In the design of this parking system, the core concepts of object-oriented programming—encapsulation, inheritance, polymorphism, and abstraction—are well implemented.

1.Encapsulation: Encapsulation is a fundamental concept in object-oriented programming that binds data (attributes) and the code that manipulates the data (methods) together into classes. The VehicleRecord class encapsulates vehicle identification information and entry time. It provides methods to access this data such as getID() and getEntryTime(), and it also includes a method to calculate the stay duration, getStayDuration(). This encapsulation protects vehicle data, allowing it to be accessed and manipulated only through public interfaces. The CarPark class encapsulates all the details of parking operations, such as managing vehicles, controlling entrance and exit barriers, and the full-capacity signal, hiding the internal implementation details and exposing only the necessary interfaces for external use.

2.Inheritance: Inheritance allows new classes to inherit attributes and methods from existing classes. This helps in code reuse and extendibility. For example, the Barrier class is an abstract base class that defines the methods raise() and lower() that all barrier classes must implement. Its existence allows for the extension of various specific types of barriers. EntranceBarrier and ExitBarrier are concrete classes derived from the Barrier class. They inherit methods from the base class and can provide specific implementations as needed, reflecting code reuse and standardized behavior.

3.Polymorphism: Polymorphism allows the use of a base class pointer or reference to call methods of derived classes, enabling the same interface to have multiple implementations. The raise() and lower() methods in the Barrier class have specific implementations in the EntranceBarrier and ExitBarrier. This allows the CarPark class to uniformly call these

methods through a reference to the Barrier class, but the actual methods executed are from the derived classes. The IDReader class and its derived classes, IDBarcodeReader and NumberplateReader, demonstrate the use of polymorphism. The CarPark class can use the interface of the IDReader class to call readID(), while the actual execution is carried out by the specific derived class.

4.Abstraction: Abstraction involves using interfaces and abstract classes to define a template for operations without specifying the details of the implementation. Both Barrier and IDReader are abstract base classes that define a set of interfaces (raise, lower, readID) without providing specific implementations. This forces the derived classes to provide specific implementations of these methods, ensuring design consistency and system flexibility.

3.Performance and Memory Management Analysis

3.1 Memory Usage Analysis

Memory allocation methods can be broadly categorized into static and dynamic memory allocation, each with distinct purposes and characteristics. Static memory allocation refers to memory that is allocated at compile time, and its lifespan typically spans the entire runtime of the program. In C++, static memory allocation is common for global variables, static variables, and local variables on the stack. The advantages include fast access speeds, as memory addresses are determined at compile time, and simple memory management, as there is no need for programmers to manually manage the creation and destruction of memory. The disadvantages are the lack of flexibility, as the size and lifecycle of the memory are fixed at compile time, which can lead to inefficient memory usage, especially when the usage duration of variables is significantly shorter than the runtime of the program.

Dynamic memory allocation involves requesting and releasing memory during program execution. In C++, this is typically achieved using the new and delete operators, or through smart pointers such as std::unique_ptr and std::shared_ptr for automatic memory management. The advantage of this method is its high flexibility, allowing memory size to be determined at runtime based on actual needs. The disadvantage is that access speeds are relatively slower, as it involves pointer usage. In the parking system, the use of memory is crucial, especially when dealing with real-time data processing and system responses. The code employs smart pointers, specifically std::shared_ptr, to manage IDReader and Barrier objects. The primary advantage of these smart pointers is their automatic memory management, which prevents memory leaks and simplifies resource sharing. They enable automatic memory release, prevent memory leaks, and allow multiple pointers to point to the same resource, facilitating easy resource sharing. Additionally, the system uses std::vector to store VehicleRecord instances, leveraging the convenience of dynamic arrays for easy expansion and contraction. This dynamic memory management capability allows for adjustments in capacity as needed and the continuous memory layout optimizes access speeds.

3.2 Performance Advantage Analysis

Within the system, methods like CarPark::enterVehicle and CarPark::exitVehicle involve multiple conditional checks and frequent calls to virtual functions. The method calls are designed to be easily expandable and modifiable, adhering to the open/closed principle. It is important to avoid excessive use of virtual function calls in functions that are called frequently, especially in performance-critical sections. Additionally, data access and processing involve traversing a std::vector to locate specific VehicleRecord instances, which is performed efficiently. The current system design already has a solid infrastructure and memory management strategy, but there is still room to enhance performance and optimize memory usage. By adjusting data structures and reducing the complexity of resource management, the system's response speed and efficiency can be further improved.

4.System Implementation and Unit Testing

This section details the implementation of the entire operation process of the parking system, along with unit testing of its main modules. The test results and simulated operation outcomes are as follows:

```cpp
// TestVehicleRecord.cpp
#include "VehicleRecord.h"
#include <cassert>
#include <iostream>

void testVehicleRecord() {
    VehicleRecord vehicle("TestID123");
    assert(vehicle.getID() == "TestID123");
    std::cout << "VehicleRecord creation and ID retrieval test passed.\n";
}
```

Fig.2 TestVehicleRecord.cpp

As shown in Figure 2, the code for TestVehicleRecord.cpp tests the instantiation and data retrieval capabilities of the VehicleRecord class. It creates a VehicleRecord object and verifies if it can correctly return the vehicle ID assigned at initialization.

Figure 3 illustrates that the test passed successfully; the VehicleRecord class can correctly store and retrieve vehicle IDs, indicating that its constructor and getID method work as expected.

```
Microsoft Visual Studio 调试控

Starting all unit tests...
VehicleRecord creation and ID retrieval test passed.
All unit tests completed successfully.

C:\Users\kyj\Desktop\Carpark\x64\Debug\Carpark.exe (进程 27964)已退出，代码为 0。
按任意键关闭此窗口. . .
```

Fig.3 The test results for TestVehicleRecord

```cpp
// TestBarrier.cpp
#include "EntranceBarrier.h"
#include "ExitBarrier.h"
#include <iostream>
#include <cassert>

void testEntranceBarrier() {
    EntranceBarrier entrance;
    entrance.raise();
    std::cout << "Entrance barrier raise test passed.\n";
}

void testExitBarrier() {
    ExitBarrier exit;
    exit.raise();
    std::cout << "Exit barrier raise test passed.\n";
}
```

Fig.4 testEntranceBarrier and testExitBarrier.cpp

Figure 4 contains the code for testEntranceBarrier and testExitBarrier, which test the raise method of the EntranceBarrier and ExitBarrier classes, respectively. The tests only invoke the raise method and assume it functions as expected.

Figure 5 shows that the test passed successfully; the raise method of both EntranceBarrier and ExitBarrier can be successfully called, suggesting that the derived classes correctly implement the method of the abstract base class Barrier.

```
Starting all unit tests...
Entrance barrier raised.
Entrance barrier raise test passed.
Exit barrier raised.
Exit barrier raise test passed.
All unit tests completed successfully.

C:\Users\kyj\Desktop\Carpark\x64\Debug\Carpark.exe (进程 34328)已退出，代码为 0。
按任意键关闭此窗口. . .
```

Fig.5 The test results for testEntranceBarrier and testExitBarrier

```cpp
// TestFullSign.cpp
#include "FullSign.h"
#include <cassert>
#include <iostream>

void testFullSign() {
    FullSign sign;
    sign.switchOn();
    assert(sign.isOn() == true);
    std::cout << "Full sign switch on test passed.\n";

    sign.switchOff();
    assert(sign.isOn() == false);
    std::cout << "Full sign switch off test passed.\n";
}
```

Fig.6 testFullSign.cpp

Figure 6 displays the code for testFullSign, which tests the switchOn and switchOff methods of the FullSign class, as well as the state check of the isOn method. Assertions are used to verify whether the light status switches correctly.

Figure 7 shows that the test passed; the state control methods of the FullSign class function properly, correctly altering the internal state and accurately reporting the current state via the isOn method.

```
Starting all unit tests...
Full sign switched on.
Full sign switch on test passed.
Full sign switched off.
Full sign switch off test passed.
All unit tests completed successfully.

C:\Users\kyj\Desktop\Carpark\x64\Debug\Carpark.exe (进程 42652)已退出，代码为 0。
按任意键关闭此窗口. . .
```

Fig.7 The test results for testFullSign

```cpp
// TestIDReaders.cpp
#include "IDBarcodeReader.h"
#include "NumberplateReader.h"
#include <cassert>
#include <iostream>

void testIDBarcodeReader() {
    IDBarcodeReader barcodeReader("SampleSerial");
    assert(barcodeReader.readID() == "SampleID");
    std::cout << "IDBarcodeReader test passed.\n";
}

void testNumberplateReader() {
    NumberplateReader plateReader("SampleSerial");
    assert(plateReader.readID() == "Plate123");
    std::cout << "NumberplateReader test passed.\n";
}
```

Fig.8 testIDBarcodeReader and testNumberplateReader.cpp

Figure 8 presents the test code for testIDBarcodeReader and testNumberplateReader, testing the readID method of the IDBarcodeReader and NumberplateReader classes. These tests assume that the readID method returns a fixed string in the test environment, verifying the method's invocation.

Figure 9 indicates that inputting test codes or license plates from the test files resulted in the test passing; the IDBarcodeReader and NumberplateReader can return the expected string, demonstrating that their readID methods perform correctly and inherit the correct structure from the IDReader.

```
Microsoft Visual Studio 调试控

Starting all unit tests...
Scan barcode: SampleID
IDBarcodeReader test passed.
Enter the number plate: Plate123
NumberplateReader test passed.
All unit tests completed successfully.

C:\Users\kyj\Desktop\Carpark\x64\Debug\Carpark.exe (进程 40384)已退出，代码为 0。
按任意键关闭此窗口. . .
```

Fig.9 The test results for testIDBarcodeReader and testNumberplateReader

```cpp
// TestCarPark.cpp
#include "CarPark.h"
#include "IDBarcodeReader.h"
#include "NumberplateReader.h"
#include "EntranceBarrier.h"
#include "ExitBarrier.h"
#include <cassert>
#include <iostream>

void testCarPark() {
    CarPark park(5);
    IDBarcodeReader barcodeReader("123");
    NumberplateReader plateReader("ABC");
    park.setReaders(&barcodeReader, &plateReader);

    EntranceBarrier entrance;
    ExitBarrier exit;
    park.setBarriers(&entrance, &exit);

    assert(park.enterVehicle("123") == true);
    std::cout << "CarPark enter vehicle test passed.\n";

    park.exitVehicle("123");
    std::cout << "CarPark exit vehicle test passed.\n";
}
```

Fig.10 CarPark.cpp

Figure 10 features the test code for testing key functions of the CarPark class, including setting up readers and barriers, and vehicle entry and exit. This test verifies whether vehicles can enter and exit the parking lot based on set conditions (vehicle detection, space checking, ID verification).

Figure 11 illustrates that, with input test barcodes and license plates, the CarPark class's methods and logic are integrated correctly, capable of handling vehicle entries and exits and managing vehicles based on their IDs. Moreover, interactions with the Barrier and IDReader classes perform normally, able to call the raise method of barriers under the correct conditions.

```
Microsoft Visual Studio 调试控

Starting all unit tests...
Scan barcode: 123
Enter the number plate: ABC
Entrance barrier raised.
Vehicle entered with ID: 123
CarPark enter vehicle test passed.
Vehicle with ID 123 exited after 0 seconds.
Exit barrier raised.
CarPark exit vehicle test passed.
All unit tests completed successfully.

C:\Users\kyj\Desktop\Carpark\x64\Debug\Carpark.exe (进程 31584)已退出，代码为 0。
按任意键关闭此窗口. . .
```

Fig.11 The test results for CarPark

Depicts the simulated process of vehicles entering and exiting the parking system. The simulation test inputs valid barcodes and license plates early, testing the entry of 5 vehicles into the parking lot, with one vehicle exiting as the third vehicle enters. The results are shown in Figure 12， A total of five vehicles attempt to enter a parking lot that has a maximum capacity of three cars. Initially, the first vehicle enters by scanning a valid barcode, causing the barrier to lift and allowing the car to pass through normally. The second and third vehicles enter by providing valid license plate numbers, and the barrier lifts for each, allowing them to pass through normally as well. At this point, one vehicle exits the parking lot, and it is reported that it stayed in the parking lot for 16 seconds. The fourth vehicle enters by scanning a valid barcode; the barrier lifts, and the vehicle enters the parking lot normally. By then, the parking lot is filled to its capacity with three cars. As a result, the indicator light changes, and further vehicles are prevented from entering. The last vehicle attempts to enter with an invalid code, and the parking lot denies entry to this vehicle.

```
Enter plate number: BCR123
Scan barcode: BCR123
Enter the number plate: BCR123
Entrance barrier raised.
Vehicle entered with ID: BCR123
Vehicle entered with ID: BCR123
Full sign switched off.
Full sign switched off.
Attempting to enter vehicle 2:
Enter barcode: NPR458
Enter plate number: NPR458
Scan barcode: NPR458
Enter the number plate: NPR458
Entrance barrier raised.
Vehicle entered with ID: NPR458
Vehicle entered with ID: NPR458
Full sign switched off.
Full sign switched off.
Attempting to enter vehicle 3:
Enter barcode: BCR126
Enter plate number: BCR126
Scan barcode: BCR126
Enter the number plate: BCR126
Entrance barrier raised.
Vehicle entered with ID: BCR126
Vehicle entered with ID: BCR126
Vehicle with ID BCR123 exited after 16 seconds.
Exit barrier raised.
Full sign switched off.
Full sign switched off.
Attempting to enter vehicle 4:
Enter barcode: BCR123
Enter plate number: BCR123
Scan barcode: BCR123
Enter the number plate: BCR123
Entrance barrier raised.
Vehicle entered with ID: BCR123
Vehicle entered with ID: BCR123
Full sign switched on.
Full sign switched on.
Attempting to enter vehicle 5:
Enter barcode: 123
Enter plate number: 123
Invalid barcode or plate number. Access denied.
Full sign switched on.
Full sign switched on.
```

Fig.10 Carpark Simulation Results

5.User Guide

System Overview：This parking system is a simulation program designed to mimic the entry and exit of vehicles in a parking lot with a limited number of spaces. The system uses barcodes and license plate recognition to manage vehicle movements.

System Requirements:
Operating System: Windows, macOS, Linux
Compiler Environment: Supports C++17, such as GCC, Clang, or Microsoft Visual Studio
Required Libraries: No additional library dependencies

Instructions for Use:
Start the Program: After launching, the program automatically enters vehicle simulation mode.
Input Barcode and License Plate Number: The program prompts the user to enter a barcode and license plate number. Follow the prompts to enter a predefined valid barcode or license plate number, then press Enter to confirm.
Observe Results: The program responds according to the input barcode and license plate number, as well as the parking lot's status (whether it is full), such as allowing entry or denying entry.
Parking Lot Status Update: The program displays the current status of the parking spaces and updates it as vehicles leave.
End Simulation: Once the simulation is complete, the program will output the final status and end automatically.

Frequently Asked Questions:
Barcode or License Plate Number Input Errors: If the input information is not on the valid list, the system will deny entry to the vehicle.
Parking Lot Full: When the parking lot is full, no vehicles can enter, and the system will display a "Parking Lot Full" message.

6.Conclusion

This project has successfully implemented a simulated parking lot system based on barcode and license plate recognition. The system allows the simulation of vehicle entries and exits, managing the status of parking spaces through interactions with simulated entrance and exit barriers, as well as a full-capacity indicator. The system accepts user inputs via a simple command-line interface and can respond based on the availability of parking spaces and vehicle identification information. The system design adheres to the principles of object-oriented programming, ensuring the modularity and maintainability of the code, utilizing concepts like encapsulation, inheritance, polymorphism, and abstraction.
Although the system meets basic requirements, there is potential for future improvements. For example, the current system uses a command-line interface, which may not be very intuitive or user-friendly. Future developments could include creating a graphical user interface to enhance user interaction experiences. Adding networking capabilities would allow for remote monitoring and management of the system, support centralized management of multiple parking facilities, and enhance data input validation to ensure the system effectively prevents invalid or malicious inputs.

References

[1] Chu S, Weitz K, Cheung A and Suciu D. (2017). HoTTSQL: proving query rewrites with univalent SQL semantics. ACM SIGPLAN Notices. 52:6. (510-524). Online publication date: 14-Sep-2017. https://doi.org/10.1145/3140587.3062348.

[2] D. E. Denning et al., "Views for Multilevel Database Security," in IEEE Transactions on Software Engineering, vol. SE-13, no. 2, pp. 129-140, Feb. 1987, doi: 10.1109/TSE.1987.232889.

[3] Kitchenham B, Brereton P, Budgen D, Turner M, Bailey J, Linkman S. Systematic literature reviews in software engineering—A systematic literature review. Information and Software Technology 2009; 51(1): 7–15

[4] Gabriel Dos Reis and Bjarne Stroustrup. 2006. Specifying C++ concepts. SIGPLAN Not. 41, 1 (January 2006), 295–308. https://doi.org/10.1145/1111320.1111064

[5] B. Stroustrup, "What is object-oriented programming?," in IEEE Software, vol. 5, no. 3, pp. 10-20, May 1988, doi: 10.1109/52.2020.

[6] R. Klump, "Understanding object-oriented programming concepts," 2001 Power Engineering Society Summer Meeting. Conference Proceedings (Cat. No.01CH37262), Vancouver, BC, Canada, 2001, pp. 1070-1074 vol.2, doi: 10.1109/PESS.2001.970207.

[7] S. Amann, S. Proksch, S. Nadi and M. Mezini, "A Study of Visual Studio Usage in Practice," 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 2016, pp. 124-134, doi: 10.1109/SANER.2016.39.

[8] Hovorushchenko T, Pavlova O, Kostiuk M. Method of Increasing the Security of Smart Parking System. JCSANDM [Internet]. 2023 May 18 [cited 2024 Apr. 15];12(03):297-314.

[9] Prasant Misra, Arunchandar Vasan, Balasubramaniam Krishnan, Venkatachari Raghavan, and Anand Sivasubramaniam. 2019. The Future of Smart Parking Systems with Parking 4.0. GetMobile: Mobile Comp. and Comm. 23, 1 (March 2019), 10–15. https://doi.org/10.1145/3351422.3351426

Appendix 1:Pseudo-code for simulated parking system

```
Program Start
    Initialize CarPark object with capacity of 3
    Create BarcodeReader and PlateReader objects
    Create EntranceBarrier and ExitBarrier objects
    Configure CarPark with reader and barrier objects

    Define lists of valid barcodes and license plate numbers
    Start Simulation:
        Loop 5 times
            Print "Attempting to enter vehicle {vehicle number}:"
            Accept user input for barcode and license plate number
            Check if input is in the valid lists
            If input is valid:
                Attempt to let the vehicle enter the car park using the barcode
                If the vehicle successfully enters:
                    Print "Vehicle has entered, ID: {barcode}"
                Else:
                    Print "Car park is full"
            Else:
                Print "Invalid barcode or license plate number, access denied"

            If it is the third iteration and the valid barcode list is not empty:
                Let the first vehicle exit (using the first barcode in the list)
                Print "Vehicle has exited, ID: {barcode}"

            Update the car park status
        End Loop

    Print "Simulation ended"
Program End
```

Video link： https://youtu.be/tMiJX5ZRb1M