

Object Orientation Programming

1st Given Edward Patch
Software Engineer Student (of HND Year 2)
Advanced Software Development
University of Wales Trinity St. Davids (of Gordan Dickers)
Swansea, Wales
Student ID: 1801492

CONTENTS

I	Introduction	1
II	Objective	1
III	Planning	1
III-A	Test Plan	1
III-B	Unit Testing	1
IV	Task One - Object Relationships	2
IV-A	Functions	2
IV-B	Vector	2
IV-C	Car Park	3
IV-D	Sign	3
IV-E	Card Reader	4
IV-F	Sensors	4
IV-G	Barrier	4
V	Task Two - Object Relationships	5
V-A	ID Reader	5
V-B	Barcode Sensor	6
V-C	Number Plate Sensor	6
VI	Structs	7
VI-A	Cards	7
VI-B	Barcodes	7
VI-C	Number Plates	7
VII	UML Class Diagram - Task One	8
VIII	UML Class Diagram - Task Two	9
IX	Conclusion	10
X	Appendix	10
X-A	Functions Library	10
X-B	Unit Testing - Source	10
X-C	Jupyter - Parking Error Codes	10
X-D	Parking Methods - Task 1 Source	10
X-E	Parking Methods - Task One Source	10
XI	References/Bibliography	10

Abstract—An object-orientated car park is requested. The objects within the car park must have appropriate attributes and methods. Unit tests are in the same Visual Studio solution package.

Index Terms—Car Park, C++, Unit Testing

I. INTRODUCTION

Video Demonstration: Parking Methods - <https://www.youtube.com/watch?v=1Eqz8b--Rew>

In this report, the Car Park object documented will demonstrate on how classes, sub-classes, methods and attributes relate and work together.

Relationships are explained and exemplified by UML Class Diagram's, which are found in sections VII - VIII. Page 8.

Features

- Parking Space number can be changed via the Car Park constructor.
- Parking Space IDs are customisable via the Car Park class.
- With the power of Object Orientation, the Car Park objects are designed for different technology and input methods.

II. OBJECTIVE

A local company requested that they wanted their staff Car Park redesigned with security barriers. The application should simulate the staff car park. Unit Testing and the console simulation, will be used to test theory and concepts. The classes can integrate in each object found in the physical car park.

III. PLANNING

The planning section contains the test plan and unit testing strategy.

A. Test Plan

To view the test plan table, refer to table I, Test Plan. Page 2.

B. Unit Testing

Existing classes and structs are used to create some Unit Tests. Custom functions are used to demonstrate certain theories and check if the logic works, before implementation. Unit testing creates a testing environment, to prove the classes and structs created are as versatile and flexible, as first intended.

Refer to table II, Unit Testing. Page 2, for the results of the Unit Testing of the Car Park features.

TABLE I
TEST PLAN

Test Name	What to Test?	Expected Result
Error Code: All Spaces Are Empty	If error code matches to logic created in Jupyter Notebook.	-15 as the return error code.
Error Code: General Spaces Are Full		-30 as the return error code.
Error Code: Disabled Spaces Are Full		-13 as the return error code.
Error Code: Child Spaces Are Full		-18 as the return error code.
Error Code: General and Disabled Spaces Are Full		-28 as the return error code.
Error Code: General and Child Spaces Are Full		-33 as the return error code.
Error Code: Disabled and Child Spaces Are Full		-16 as the return error code.
Error Code: All Spaces Are Full		-31 as the return error code.
Membership Card Struct	Create a vector of structs and store card details successfully.	Access the vector with the right card details.
Assignment of MembershipID	Check if membershipID starts at 0 and increments by 1.	MembershipID should pass with the test logic.
Check User Cards (Membership Status)	Check if membershipID is equal to query.	Check User Cards should pass with the test logic.
User Integrity Test	Check if the user card information has been populated properly.	User Integrity should pass the test.

IV. TASK ONE - OBJECT RELATIONSHIPS

Refer to figure 1, UML Class Diagram, Page 8 for a visual representation of the object relationships.

A. Functions

Functions class library, reference to subsection X-A, Functions Library. Page 10. The library offers variety of IOSTream tools such as-

- Console Control.
- String Conversion.
- Value Conversion.

TABLE II
UNIT TESTING

Reference Table 1		Unit Test Name	Test 1	Test 2	Test 3	Solution
T E S T N A M E	All Spaces Empty	UTAvailabilityErrorCodes EmptySpaces	Pass			N/A
	General Spaces Full	UTAvailabilityErrorCodes GeneralSpaces				
	Disabled Spaces Full	UTAvailabilityErrorCodes DisabledSpaces				
	Child Spaces Full	UTAvailabilityErrorCodes ChildSpaces				
	General Spaces and Disabled Spaces Full	UTAvailabilityErrorCodes GeneralSpacesAndDisabledSpaces				
	General Spaces and Child Spaces Full	UTAvailabilityErrorCodes GeneralSpacesAndChild Spaces				
	Disabled and Child Spaces Full	UTAvailabilityErrorCodes DisabilityAndChildSpaces				
	All Spaces Are Full	UTAvailabilityErrorCodes FullSpaces				
M	Membership Card Struct	UTCardStruct VectorMembershipAlgorithm	Fail	Pass		String Conversion Needed Work.
E	Membership Card Comparison Check	UTCardStruct ComparisonErrorCheck	Fail	Fail	Pass	Above error caused one fail. Test logic reworked.
S	Membership Card Integrity Check	UTCardStruct UserRecordIntegrity	Pass			N/A

The library is designed to work for both Linux and Windows operating systems. This means that the application is compatible for both Linux and Windows packages. The Functions library pack can be configured within the class itself.

B. Vector

Vector class offers location information, that any object that can be placed would benefit from. The vector class offers 3-dimensions, which are used to pin point object locations. The vector class will be related to the Car Park class object, reference to sub-section IV-C, Car Park. Page 3.

Features

- Scale Vector
- Add Two Vectors
- Subtract Two Vectors
- Dot Product of Two Vectors

This gives the derived classes of the **Car Park** class a method of storing and retrieving location details. Examples are shown below:-

Vector — Location Examples

```
// Car Park | Set Location | Argument:
// Vector Pointer
carPark->SetLocation(*new
    Vector(25.4f, 109.6f, -10.f));

// Returns: Vector Class
carPark->GetLocation();

// Card Reader Location
cardReader->SetLocation(*new
    Vector(20.5f, 100.f, -5.f));

// Returns: Vector Class
cardReader->GetLocation();
```

C. Car Park

Car Park is the base class of the following objects:-

- Sign Class.
- Card Reader Class.
- Sensors Class.
- Barrier Class.

This class is derived from the following two objects, **Functions** and **Vector**. It is possible to access these two classes from the sub-classes of Car Park. The base classes are set as a Private member, rather than a Protected member to access the base classes elsewhere.

As subobjects of the representation of a staff Car Park, is situated inside the Car Park, we can treat the Car Park class as the base class. In other words, to access the Car Park's functionality, sub-objects communicates with the base class.

Base classes offer tools, which are mainly provided outside of the class or to the derived sub-classes. The Car Park class contains a sub-object pointer of itself, to where the information is stored. This can enable data to live longer in the class, as each sub-class will recall the Car Park class, recalling the base class default constructor.

The terminology of this problem is called **Encapsulation**. Refer to the article, OOP Concept for Beginners: What is Encapsulation [1] for more information.

Virtual functions can be overridden or used with default settings via its derived classes. These are used on two occasions in this class. A virtual function was used on one occasion, to have a defaulted error message. Refer to the derived classes below for further details. The code is exemplified below:-

Virtuals — CarPark.h

```
// The message can be customised,
// depending on where the message is
// displayed.
virtual std::string
    GetParkStatusMessage(int parkingSpace
        = NULL, std::string parkingSpaceID =
        "")
{
    if(GetAvailabilityStatus() != -2)
        return "Success: Spaces Available";
    return "Error: Car Park is full.";
}

//Displays all spaces
virtual void DisplaySpaces()
{
    for (int i = 0; i <
        carPark->totalSpaces; i++)
    {
        std::cout << "Parking Space ID: " +
            carPark->parkingSpaceID[i] <<
            "\t\t";
        std::cout << "Parking Space Status: "
            + carPark->parkingSpaceStatus[i];
        std::cout << "\n";
    }
}
```

D. Sign

The Sign class has been introduced to represent the Car Park's information sign, to display whether the car park is Full or Empty.

This class is fully customisable, based on if the hardware of the sign is smart or static. Which implies that the software, can either say Full or Empty or tell the Car Park user further information. The information can include what spaces are full or empty.

The base class, **Car Park**, helps supply the feature with its *GetParkStatusMessage()*; virtual function. This function can be defaulted to Full or Empty, or overridden.

The code snippets below show how the Sign class would display the status of the CarPark and how the class would override the display status messages from the base class:-

Display Status — Sign.cpp

```
void Sign::DisplayText()
{
    std::cout << "\t\t\tCar Park | Message
    Board\n\n\t";
    std::cout << GetParkStatusMessage() <<
        std::endl << std::endl << std::endl
        << std::endl;
}
```

Virtual Override — Sign.cpp

```
Sign::GetParkStatusMessage(int
    parkingSpace, std::string
    parkingSpaceID)
{
    switch (GetAvailabilityStatus())
    {
        case -15:
            return "Available Spaces";

        case -30:
            return "General Spaces are Full.";

        case -28:
            return "General Spaces and
            Disability Spaces are Full.";

        case -13:
            return "Disability Spaces are full.";

        case -18:
            return "Child Spaces are Full.";

        case -33:
            return "General Spaces and Child
            Spaces are Full.";

        case -16:
            return "Disability and Child Spaces
            are Full.";

        default:
            return "Error: Car Park is full.";
    }
}
```

E. Card Reader

Card Reader class represents a payment meter found in car parks. However, this object is designed to accept membership cards to access the staff Car Park. When a car pulls up, the user will see a “User Interaction Menu”. Instead of interacting with the IOSTream, the user would touch the button based on the options. This class can also work with another type of interfaces like a touch screen or voice-activated interface. Refer to the article, What is Object Oriented Programming? OOP Explained in Depth [2] for more information.

As the Card Reader object is most likely to have an LCD screen of some kind. The park status message tells the user where to park. The arguments such as *parkingSpace* and *parkingSpaceID* come into play.

Take a look at the code snippet to study how the GetPark-StatusMessage() virtual function was overridden within this class to widen the base class reusability and usefulness.

Virtual Override — CardReader.cpp

```
std::string
CardReader::GetParkStatusMessage(int
    parkingSpace, std::string
    parkingSpaceID)
{
    int availabilityStatus =
        GetAvailabilityStatus();

    // only continue if car park is not full
    if (availabilityStatus != -31)
    {
        // combines Proceed to "Parking Space
        ID"
        if (parkingSpace == 0 &&
            availabilityStatus != -30) return
            "Proceed to " + parkingSpaceID;
        if (parkingSpace == 1 &&
            availabilityStatus != -18) return
            "Proceed to " + parkingSpaceID;
        if (parkingSpace == 2 &&
            availabilityStatus != -13) return
            "Proceed to " + parkingSpaceID;
    }

    return "Please exit the car park."; //
    could send them back to the
    selection.

    DisplaySpaces();
}
```

F. Sensors

To trigger the Card Reader when a car arrives, the Sensor communicates to the Card Reader object. The Sensor is derived from the Car Park class. This object is declared and initialised in Card Reader, which enabling the communication between the Card Reader class object and the Sensor class object.

A weight minimum is required to trigger the Sensor object. This minimum weight can change based on the type of sensor used. Weight minimums are needed when calling the Sensor class default constructor.

Sensor Constructor and Trigger — Sensor.cpp

```
Sensor::Sensor(CarPark& carParkObj,
    float aTriggerWeight) :
    CarPark(carParkObj)
{ triggerWeight = aTriggerWeight; }

// if carWeight >= triggerWeight then
// car is detected
void Sensor::ActivateTrigger(float
    carWeight)
{
    if (carWeight >= triggerWeight)
        bIsPressed = true;
    else bIsPressed = false;
}
```

G. Barrier

As the current staff Car Park has no Barriers, the Barrier class controls the entrance and exit Barriers' functionality. The Barrier class has a similar makeup of the Sensor class. The Barrier class offers functionality to open and close the Barrier.

The Card Reader class declares and initialises the Barrier class, and controls the Barrier. Entrance barrier opens if the membership card is valid. Otherwise, the exit barriers will open to direct the non-member outside of the car park.

Barrier Functionality — Barrier.cpp

```
Barrier::Barrier(CarPark& carParkObj) :
    CarPark(carParkObj) {
    SetState(false); }

void Barrier::SetState(bool state) {
    bBarrierStatus = state; }

// if barrier is open or close, change
// display
void Barrier::DisplayState()
{
    if (bBarrierStatus)
        std::cout << "Barrier is open!\n";
    else
        std::cout << "Barrier is closed!\n";
}
```

V. TASK TWO - OBJECT RELATIONSHIPS

A. ID Reader

ID Reader class is the base class of Barcode Sensor and Number Plate Sensor classes. Functions class is the base class of ID Reader. This class offers IOStream and other tools, for more information, refer to sub-section IV-A, Functions. Page 2.

This class offers tools to communicate to the membership and DVLA Car Registration database, refer to the UK Government's API Development documentation, What is Object Oriented Programming? OOP Explained in Depth [3]. The car number plate is recorded to a local database, cross-referenced to the DVLA Car Registration database. The method prevents non-staff or visitors from parking, after being instructed to leave the car park.

The IDReader class is not derived from the Car Park class, as it acts as an external object. The IDReader can be instantiated inside the Car Park class or as hand-tool objects, which staff-personnel can use, to check parked cars or membership cards to prevent fraudulent cards from being created.

Code snippets below demonstrate how Car Registrations and Barcodes are registered into the Struct's **Barcodes** and **NumberPlates**. The **Barcodes** struct uses data from the **Cards** vector struct, to assign a barcode to the membership cards.

Add and Shows Barcode — IDReader.cpp

```
void IDReaders::AddBarcode(std::string
    membershipID, std::vector<struct
    Card>* card)
{
    // transfers the card object from main to
    // update the object data
    this->cards = card;

    // Adds a new barcode
    barcodeDatabase.push_back(Barcodes());

    // if size of barcodeDatabase is 0, set
    // id to 'BC-0'
    if (barcodeDatabase.size() - 1 == 0)
        barcodeDatabase.at(0).id = "BC-0";

    // add one from last registered ID
    else barcodeDatabase.at(
        barcodeDatabase.size() - 1).id = "BC-" +
        std::to_string((int)TextToNumber(
            barcodeDatabase.at(
                barcodeDatabase.size() - 2).id) + 1);

    barcodeDatabase.at(
        barcodeDatabase.size() - 1).membershipID
        = membershipID;
}

void IDReaders::ShowBarcodes()
{
    for (int i = 0; i <
        barcodeDatabase.size(); i++)
    {
        std::cout << "ID: " <<
            barcodeDatabase.at(i).id << " |
            Membership ID: " <<
            barcodeDatabase.at(i).membershipID <<
            std::endl;

        for (int j = 0; j < cards->size(); j++)
            if (cards->at(j).membershipID ==
                barcodeDatabase.at(i).membershipID)
                std::cout << "Name: " <<
                    cards->at(j).memberName << " |
                    Membership Type: " <<
                    cards->at(j).membershipType <<
                    std::endl;
    }
    if (barcodeDatabase.size() != 0)
        PauseSystem();
}
```

Add Number Plate — IDReader.cpp

```
void IDReaders::AddNumberPlate()
{
    // Adds a NumberPlate to vector object
    numberPlateDatabase.push_back(
        NumberPlates());

    // Car Registrations are already
    unique.
    numberPlateDatabase.at(
        numberPlateDatabase.size() -
        1).carRegistration =
        TextInput("Input Car
        Registration: ");

    //Asks user for make/model, colour and
    milage
    numberPlateDatabase.at(
        numberPlateDatabase.size() -
        1).carMakeModel =
        TextInput("Input Car Make/Model:
        ");
    numberPlateDatabase.at(
        numberPlateDatabase.size() -
        1).carColour = TextInput("Input
        Car Colour: ");
    numberPlateDatabase.at(
        numberPlateDatabase.size() -
        1).carMilage = TextInput("Input
        Car Milage: ");
}
```

B. Barcode Sensor

Membership cards or keycards could have QR or Barcode readers integrated into the Barcode Sensor class. Either QR or Barcodes references the correct primary index within the Barcodes database. **Barcodes** struct holds the Barcode/QR ID and *membershipID* variable.

The code snippet below shows the code that is used to trigger the barcode. The Card Reader object could activate the trigger.

Barcode Trigger — BarcodeSensor.cpp

```
void BarcodeSensor::OperateScanner(
    std::vector<struct Barcodes>*
    barcodeDatabase,
    std::vector<struct Card>*
    userCardDatabase)
{
    // Update Vectors before using them
    UpdateVectors(barcodeDatabase,
        userCardDatabase);

    // This will be the query for the
    Barcode Scanner
    std::string qry = TextInput("Enter
    Barcode Scanner: ");
    SubmitQuery(qry);
}
```

C. Number Plate Sensor

A Number Plate Sensor is requested for the Staff car park, which increases the security of the Staff Car Park parameter. The functionality of the Number Plate Sensor as listed:-

- **Registration Plate Input -**
A function of Input is needed to gather the registration plate. If there is no input method, then the number plate sensor won't recognise existing number plates.
- **Query the Struct Vector of Number Plates,** refer to section VI-C, Number Plates. Page 7. -
When querying the registration plate, the string information is compared with the string information of the database. If there is a match, then the Number Plate is recognised. However, if otherwise, then the Number Plate will not be able to identify the registration plate.

Extra Functionality

It's possible to cross-reference the struct of Number Plates against the DVLA's database to ensure the vehicle is registered and to gather vehicle information. The functionality enables this information, and with the help of DVLA's API availability, integration of this feature would seem modular. Again, with the help of Object Orientation, creates the possibility of joining other objects together.

OperatingScanner — NumberPlateSensor.cpp

```
void NumberPlateSensor::OperateScanner(
    std::vector<struct NumberPlates>*
    numberPlateDatabase)
{
    // Update Vectors before using them
    UpdateVectors(numberPlateDatabase);

    // This will be the query for the Car
    Registration
    std::string qry = TextInput("Enter Car
    Registration: ");
    SubmitQuery(qry);
}
```

VI. STRUCTS

In this particular solution, structs demonstrate database data. Structs are defaulted to a public scope and have the same functionality as classes. These structs are less focused on displaying the idea that Car Park would connect to an external database.

A. Cards

The code snippet below shows the Cards struct. This struct is designed to hold membership card information.

Card Struct — CardReader.h

```
// Card could link to the database (via
// the membershipID)
struct Card
{
    // Database
    std::string membershipID; // Unique
    // IDs to link with database
    std::string memberName; // Card Reader
    // could output member's name
    std::string membershipType; //
    // membership type
    std::string department; // if
    // membershipType is Staff, the
    // department will become required.

    // Card Details
    std::string typeOfCard; // Visitor
    // Card or Staff Card
    std::string expiryDate; // If expired,
    // sign will tell user to enter the
    // carpark, and report to office to
    // renew their membership or drive
    // around carpark and approach exit
    // barrier.
};
```

B. Barcodes

The code snippet below shows the Barcodes struct. This struct is designed to hold barcode information.

Barcodes — IDReaders.h

```
struct Barcodes
{
    std::string id;
    std::string membershipID;
};
```

C. Number Plates

The code snippet below shows the Number Plate struct. This struct is designed to hold Number Plate information.

Number Plates — NumberPlates

```
struct NumberPlates
{
    std::string carRegistration;
    std::string carMakeModel;
    std::string carColour;
    std::string carMilage;
};
```

VII. UML CLASS DIAGRAM - TASK ONE

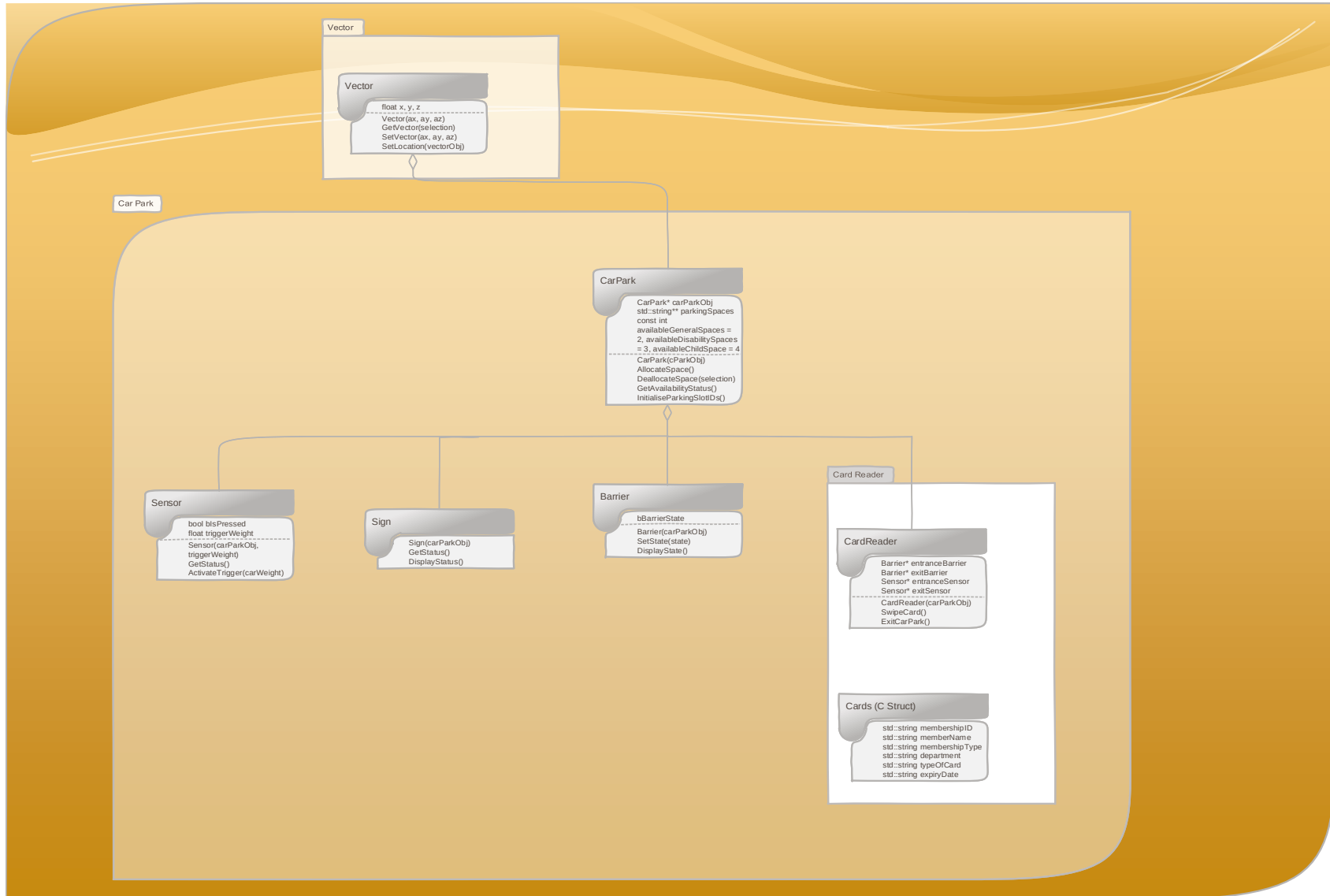


Fig. 1. UML Class Diagram - Car Park Object

VIII. UML CLASS DIAGRAM - TASK TWO

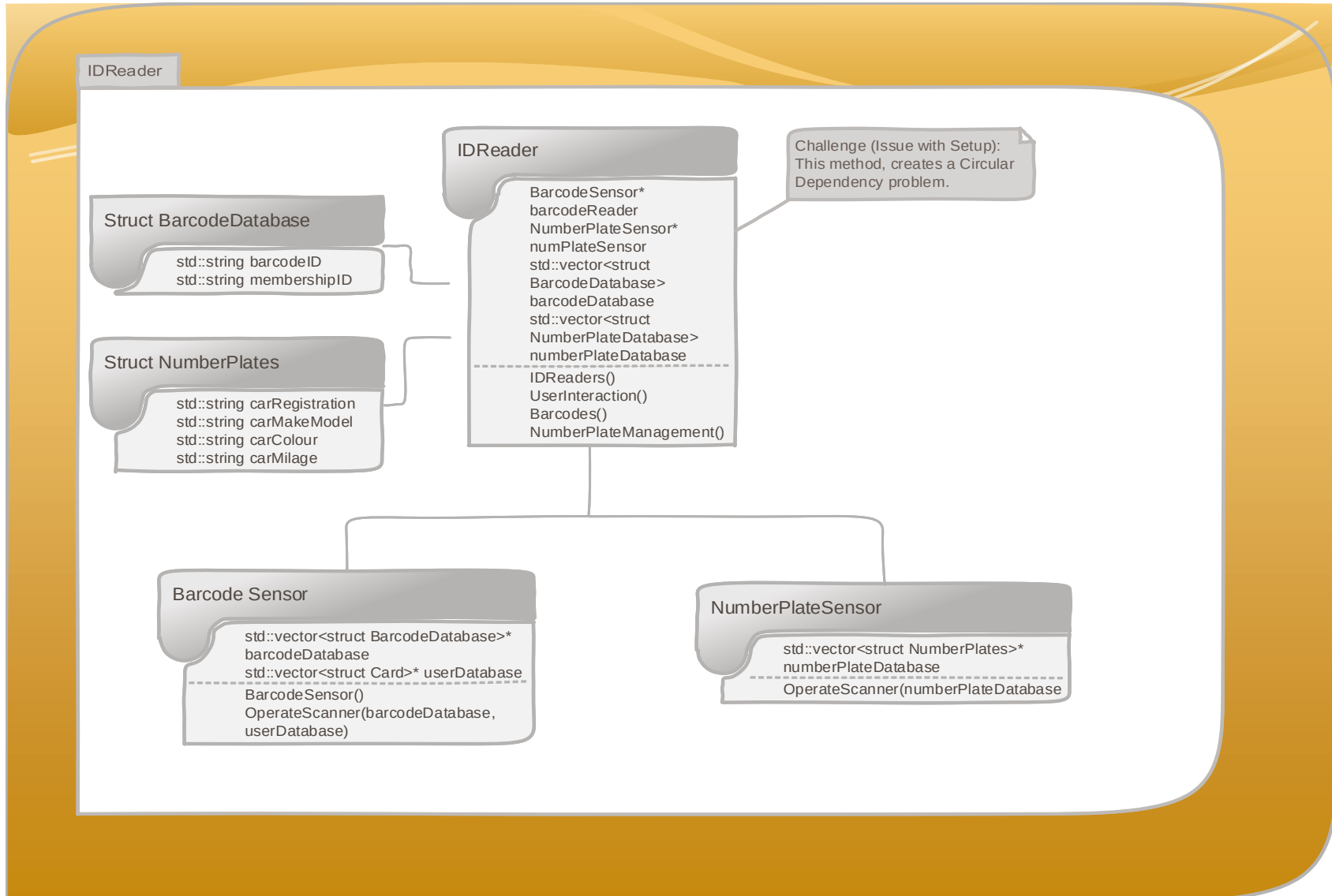


Fig. 2. UML Class Diagram - IDReader Object

IX. CONCLUSION

Staff Car Park can add security measures and improve the service of the staff car park. The methods have proven to be effective via Unit Testing and final implementations to the Car Park project.

Objects available in this project are dynamic and adaptable to a different type of technology, whether manual input or automated input.

Security measures the Car Park software offers:-

- Membership Card Information.
- Card Readers.
- Sensors and Barriers.
- IDReaders (Number Plate Sensor and Barcode Readers).

X. APPENDIX

A. Functions Library

Author: Edward Patch (1801492)

Disclaimer: The Functions library started development since 2019. This library is developed throughout the author's studies.

Reference: Functions Library - **Hyperlink:**

<https://github.com/ShinkuKira21/Virtual-CarPark/blob/master/Library/Functions.h>

The Function library offers streaming tools, such as:-

- TextInput(std::string msg) - Allows user text input.
- NumberInput(std::string msg) - Allows user number input.
- TextToNumber(std::string input) - Converts text to integer.

After completing the parking methods project, a Color-Text(std::string input, int FG, int BG, int opt = 1) function was added, to add coloured text to console applications.

This method was used in the IDReader objects.

B. Unit Testing - Source

Reference: Unit Testing Car Park - **Hyperlink:**

<https://github.com/ShinkuKira21/Virtual-CarPark/blob/master/C%2B%2B/UT-CarPark/UT-CarPark.cpp>

C. Jupyter - Parking Error Codes

Reference: Jupyter Planning - **Hyperlink:**

<https://github.com/ShinkuKira21/Virtual-CarPark/blob/master/Jupyter/Sign%20Operators.ipynb>

The Jupyter Notebook enables users to enter code in either Python, R, C++ or languages. Python is a great tool when it comes to planning with Python's statistic capabilities. Jupyter Notebook: <http://www.jupyter.org/try>

D. Parking Methods - Task 1 Source

Reference: Parking Methods - **Hyperlink:**

<https://github.com/ShinkuKira21/Virtual-CarPark/tree/master/C%2B%2B/Parking%20Methods>

- Vector Object - [Vector.h](#) - [Vector.cpp](#)
- Car Park Object - [CarPark.h](#) - [CarPark.cpp](#)
- Sign Object - [Sign.h](#) - [Sign.cpp](#)
- Card Reader and Card Objects - [CardReader.h](#) - [CardReader.cpp](#)
- Barrier Object - [Barrier.h](#) - [Barrier.cpp](#)
- Sensors Object - [Sensors.h](#) - [Sensors.cpp](#)

E. Parking Methods - Task One Source

Reference: Parking Methods - **Hyperlink:**

<https://github.com/ShinkuKira21/Virtual-CarPark/tree/master/C%2B%2B/Parking%20Methods>

- IDReaders, Barcodes, NumberPlates Object - [IDReaders.h](#) - [IDReaders.cpp](#)
- BarcodeSensor Object - [BarcodeSensor.h](#) - [BarcodeSensor.cpp](#)
- NumberPlateSensor Object - [NumberPlateSensor.h](#) - [NumberPlateSensor.cpp](#)

XI. REFERENCES/BIBLIOGRAPHY

- [1] T. Janssen, "OOP Concept for Beginners: What is Encapsulation," Nov. 2017. [Online]. Available: <https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/>
- [2] E. Doherty, "What is Object Oriented Programming? OOP Explained in Depth," Apr. 2020. [Online]. Available: <https://www.educative.io/blog/object-oriented-programming>
- [3] GOV.UK, "DVLA Vehicle Enquiry Service API." [Online]. Available: <https://localhost/apis/vehicle-enquiry-service/vehicle-enquiry-service-description.html>
- [4] "Project Jupyter." [Online]. Available: <https://www.jupyter.org>
- [5] M. Weststrate, "How to fix nasty circular dependency issues once and for all in JavaScript & TypeScript," Jul. 2018. [Online]. Available: shorturl.at/zAJ68
- [6] cassou, "Subobject-oriented programming: diamonds are forever," Jun. 2015. [Online]. Available: <https://project.inria.fr/se-seminars/subobject-oriented-programming-diamonds-are-forever/>
- [7] "std::vector - cppreference.com." [Online]. Available: <https://en.cppreference.com/w/cpp/container/vector>