

Carparkreport

by Aaron David

Submission date: 22-Apr-2024 02:58PM (UTC+0100)

Submission ID: 230336077

File name: 49633_Aaron_David_Carparkreport_570693_936040235.docx (410.58K)

Word count: 2641

Character count: 14616

The Implementation of Object Oriented Programming Best Practices In a C++ Car Park Management Program

View running code here :

https://drive.google.com/file/d/1huhCKuKIDyB_7xQJnd3IZnVJESJUstIU/view?usp=sharing

Introduction

Object oriented programming (OOP) is a foundational concept in modern computing. In contrast to earlier programming paradigms such as functional programming, OOP focuses development on the use of distinct objects that represent the real-world entities involved in whatever function the program is aiming to carry out. [1]

OOP has been critical in facilitating modern software development. The modular nature, allowing for far greater extensibility and overall manageability than previous methods. Classes allow for high re-use of code, making development a far more efficient process. The easily re-usable nature of design using classes also contributing to a great increase in software scalability. [1]

There are a number of principles at the core of OOP that form a blueprint for developing effective and functional Object Oriented code. The four key principles being abstraction, inheritance, encapsulation, and polymorphism [1].

This report aims to outline these core concepts along with other key components of OOP in C++, the best practices in their implementation, and exemplify their application in the development of a C++ car park management program.

Classes

Within programming, we utilise many data types to carry out whatever operation that we are trying to achieve. These data types have pre-determined properties that ensure that they function in a certain way that we can count on to produce a particular result. Fundamentally, a class is a data type that is created and designed by a user to meet a set of specific requirements. A class generally consists of attributes, data members that set the initial state of a class, and member functions, that perform class specific operations. Once we have created a class we can then use it to create instances, these are the objects that we refer to in 'Object' oriented programming. They allow us to represent real-world items in our program.

Classes: Best practices

Encapsulation and Information Hiding

A fundamental principle within OOP, encapsulation is the process of building an object that contains its own relevant data that is protected via an interface that only reveals a necessary amount of access to the inner workings of the class to other elements of the program. Within C++, attributes and methods within a class fall under one of three use access modifiers; [2] [3] Private: in which data is only accessible to the class itself, Public in which data is accessible to the entire program, and Protected, which allows access only to a classes

subclasses. Private attributes should be made accessible only when necessary, by utilising getter and setter functions that ensure that the data is being accessed by any other class or function in a specific pre-defined way, preventing unwanted access and modification to the classes private data [4]. These functions can then include checks or triggers for specific operations to regulate data changes.

SOLID Principles in Class Design

In the design of classes in Object-Oriented Programming (OOP), a set of five principles known as SOLID are fundamental to creating functional, readable, and testable code. These principles guide the development of software architecture that is both robust and easy to maintain. [5]

Single Responsibility Principle (SRP)

The Single Responsibility Principle dictates that a class should have only one reason to change, meaning it should only have one job or responsibility. Following SRP can prevent classes from becoming overly complicated. Complicated structures in a class can hinder understanding and collaborative efforts. Sutter and Alexandrescu's 2005 advice, "Prefer minimal classes to monolithic classes"[1], reinforces SRP by suggesting it's easier and more effective to manage multiple simple classes than a single complex one. This approach reduces the risk of different responsibilities becoming intertwined and dependent on each other. By maintaining simpler classes that focus solely on necessary functions, we ensure better encapsulation. Overloaded classes with complex functions can expose or obscure the internal state of objects, which contradicts the principle of information hiding in OOP. [5]

Interface Segregation Principle (ISP)

The Interface Segregation Principle emphasizes the need for flexibility and extensibility in programming. ISP advises that interfaces should be separated to avoid unnecessary dependencies; no class should be forced to implement interfaces they do not use. This principle suggests that a base class should not include redundant logic just to fulfill the interface requirements. Like SRP, ISP supports the idea of simplicity and specialization in class design. Both principles advocate for reducing complexities in classes, thus enhancing maintainability and the clarity of the software design. [5]

Abstract classes

A class in its most basic form can be directly instantiated and utilised to create objects, however there are several other forms of classes that can allow us to produce, well-functioning, easy to read code, such as abstract classes. An abstract class is a form of class that acts as a blueprint or foundation for the creation of derived classes and can therefore not themselves be instantiated. Rather they encapsulate a common functionality and structure that allows code re-use and ensures that the design of derived classes remains consistent. They should only contain virtual methods that allow full extensibility in any derived classes and prevent any future modifications made to the abstract class from altering the functionality of any derived classes [2].

Evaluation of Car Park Management System design

In this section I shall proceed to describe the design of the Car Park management system, element by element, outlining the structure, operation, and relationships for each, finally assessing how well they adhere to the best practices that we have discussed.

Classes

Description

This is a simple abstract class that provides a blueprint for various ID reader classes that are needed in the implementation of the Car Park system. It contains just one attribute being **serialNumber** which stores the serial number that is read by an identification device. It also contains a virtual function **read()** representing the actual reading function of the ID reader.

```
#pragma once
#include <string>

class cIDReader {
protected:
    std::string serialNumber;
public:
    cIDReader(const std::string& serial);
    virtual std::string read() = 0; //virtual read function
};
```

Figure 1. cIDReader code

Derived Classes

There are two classes are derived from cIDReader; cBarcodeReader, and cNumberplateReader. cBarcode reader was designed in order to simulate the reading of a barcode on an individuals ID card, and cNumberplate reader to simulate the reading of a cars number plate.

Adherence to Best Practices

This class conforms to the single responsibility principle. It is a simple class that contains just one function that has one reason to be modified. There is no overcomplication or entanglement with other classes. It also complies with the Open Closed Principle, it is extensible and there is no overcomplication that would require changes in any derived classes in the case of changes to cIDReader.

The design of this class makes appropriate use of encapsulation. **serialNumber** is under the access modifier 'Protected' ensuring a level of information hiding while also allowing derived classes to access it when necessary.

cIDReader is an abstract class, and therefore utilises just a pure virtual function allowing proper polymorphic use in derived classes. The obligation to specialise this function in derived classes ensures that this class adheres to the Liskov substitution principle.

Overall this class conforms well to the best practices that we have covered. It should operate well in the programs current state, and facilitate any future modifications without any complication.

Each of cIDReaders derived classes class inherits the properties of the cIDReader class and simply implements a specialised version of the **read()** function. This is a simple and uncomplicated example of polymorphism. These classes both encapsulate their specific functionality simply and clearly and they should carry out their desired function well.

cBarrier

Description

This class acts as a base class for any barriers required in the car park. It has just one attribute, a Boolean value **'up'** which should show true when a barrier has opened to let a car through, an false when the barrier has closed. It has three virtual methods two; **'raise'** and **'lower'**, that modify the **'up'** attribute and a third; **'isup'** that operates as a get function for 'up' to be read by other classes

Derived classes

cBarrier has two derived classes; cEntranceBarrier, and cExitBarrier. Demonstrated clearly by their nomenclature cEntranceBarrier simulates the entrance of the car park, while cExitBarrier simulates the exit.

Adherence to best practices

The overall design of cBarrier adheres to the single responsibility principle well as it is solely concerned with the management of a barriers state. It is structured in a way that allows for extension of its functionality in derived classes without any modification of its own code, therefore fulfilling the Open/Closed principle. It only describes necessary woperations that are highly unlikely to be redundant to any derived classes, demonstrating Interface Segregation Principle.

This class utilises solely pure virtual methods to allow derived classes to properly specialise behaviour if needed, therefore following the required structure of an abstract class.

The class demonstrates proper encapsulation by keeping the **'up'** attribute private and only allowing its access/modification via the public **'raise'** and **'lower'** and **'isup'** methods. This ensures controlled access to and therefore proper security for the **'up'** attribute. Defining 'isup' as a **const** also ensures that its functionality can not be mistakenly modified in future.

cBarrier is a simple abstract class that should offer sufficient functionality and easy implementation of polymorphism for the required entrance and exit barriers within our car park.

The derived classes cEntranceBarrier and cExitBarrier both extend cBarrier and override the raise() and lower() virtual methods, specialising the operation of each to meet their differing needs. The use of inheritance here allows the creation of separate barrier types while maintaining a mutually defined interface.

The implementation of the cBarrier and its derived classes successfully fulfils the need for multiple types of barrier in our car park system and generally conforms to best practices.

cCarPark

This class is more complicated than those covered so far. The `cCarPark` class has been designed to represent the carpark as a whole. It essentially serves as the operational centre of our car park, orchestrating the overall function of each individual component. It contains two private int variables '**capacity**' that represents the maximum number of cars able to park in the car park, and '**freeSpaces**' representing the number of parking spaces available in the carpark. four methods; '**update()**', '**decrementSpaces()**', '**incrementSpaces()**', and '**spacesLeft()**'.

'**update()**' contains the update logic for each of the major events in the car park. It relies on a switch that takes an input, which is taken in `main()`, from '**1**' to '**4**'; '**1**' triggering the simulation of a car entering, '**2**' simulating a car exiting, '**3**' displaying the details of all cars parked in the car park, and '**4**' exiting the program.

```
void cCarPark::update(char input) {
    switch (input) {
        case '1': // Car entering
            //Sensor detects car
            entranceSensor.detectCar();
            // If sensor
            if (entra
                // on
                manager.printAllParkedCars();
            //Has
            entrance
            userguide.printUserGuide();
            //Has
            std::
            break;
        case '3': // display details of all cars currently parked
            manager.printAllParkedCars();
            //Has
            entrance
            userguide.printUserGuide();
            //Has
            std::
            break;
        case '4': // Exit program
            std::cout << "Exiting system." << std::endl;
            return;
        //Has
        decre
    }
    entranceSensor.clearDetection();
    //Print statement to notify that car has entered
    std::cout << "Car has entered the car park\n" << std::endl;
    //update fullsign
    fullsign.update(freeSpaces);
    //Show current status of fullsign
    fullsign.display();
    userguide.printUserGuide();
}
else {
    std::cout << "Apologies, there is no space in the car park";
}
break;
}
```

```
case '2':
    //Sensor detects car at exit barrier
    exitSensor.detectCar();
    //If statement checking if car is detected
    if (exitSensor.isCarDetected()) {
        // car manager exiting method
        manager.carExiting();
        //raise the exit barrier
        exitBarrier.raise();
        // keep raised for 5 seconds for car to leave
        std::this_thread::sleep_for(std::chrono::seconds(5));
        // lower barrier
        exitBarrier.lower();
        //increase freespaces accordingly
        incrementSpaces();
        //reset sensor
        exitSensor.clearDetection();
        // Print statement to thank parker
        std::cout << "Thank you for parking with us." << std::endl;
        //update fullsign
        fullsign.update(freeSpaces);
        //print status of fullsign
        fullsign.display();
        userguide.printUserGuide();
    }
    break;
}
```

'**decrementSpaces()**' acts to reduce the value of '**freeSpaces**' by one upon a car entering the car park. '**incrementSpaces**' acts to increment the value of '**freeSpaces**' ++1 upon a car leaving the car park.

'**spacesLeft()**' acts as a get function to call the value of '**freeSpaces**'.

Adherence to best practices

This class attempts to encompass a wide range of functions that aren't necessarily closely related, meaning that it violates the Single responsibility principle. A refactoring of this code and possible decomposition into smaller classes may prove favourable to future development and extensibility of the class. The extensive nature of the functionalities entangled in the class also lead to possible violation of the Interface Segregation principle.

When designing the class it seemed necessary to include all of this functionality in the central car park class, however upon reflection it should be simplified and decomposed.

cCarManager

This class was created out of a need to manage the data of the cars moving in and out of our car park. It contains three attributes, '**numberPlateReader**' which points to an instance of '**cNumberPlateReader**', '**barcodeReader**', which points to an instance of '**cBarcode Reader**' and '**cars**' an unordered map of the information of each car parked in the carpark. To store car information, I utilised a struct called `CarInfo` to link to the unordered map.

time that a particular car has been parked.

[illegible]

Adherence to best practices

In regards to SOLID principles; this class has a good focus on the management of car information, therefore adhering to Single Responsibility Principle. The access modifiers used and structure of the class ensure that it fits with Open/Closed principle.

The use of raw pointers for numberplate and barcode values does seem to contradict many recommendations for modern c++ programming, which often suggest that smart pointers should be used in their place for a safer program [4].

cFullSign

This class is responsible for displaying the number of spaces left in the car park to potential users. It has one private attribute, **'freeSpaces'**, which represents the number of free spaces left in the car park. It has two methods, **'update'** which takes an int as an argument and sets **'freeSpaces'** to the value of its input, and **'display'** which prints the status of the car park.

Adherence to best practices

This is a simple class that performs a basic function, adhering to the single responsibility principle.

cSensor

This class represents the a sensor that detects the presence of a car at the barriers of the car park. It contains a private Boolean attribute that shows true when a car is detected, and false when not. It has three methods, detectcar() that sets carDetected to true, clearDetection() that sets carDetected to false, and isCarDetected(), which acts as a get function for carDetected.

Adherence to best practice

While the class itself mostly adhered to relevant SOLID principles. It should have been refactored into an abstract class to allow for proper polymorphism, [3] as there should be individual sensors at each barrier. However I neglected to do this as I had simply set multiple instances of cSensor instead.

Conclusion

Overall I believe the implementation of my program to be generally compliant with common best practices in c++ OOP, however there are a few areas for improvement that have been discussed in this report.

Report References

- [1] E. Balagurusamy, Object oriented programming with C++, (2021)..
- [2] B. B. L. Stroustrup, The C++ programming language: reference manual (No. BELL-CSTR-108), (1984)..
- [3] H. & A. A. Sutter, C++ coding standards: 101 rules, guidelines, and best practices., Pearson Education., (2004).
- [4] R. Grimm, C++ Core Guidelines Explained: Best Practices for Modern C++, Addison-Wesley Professional., (2022). .
- [5] V. K. V. T. V. S. N. E. T. M. M. A. A. N. A. & B. A. Madasu, "Solid principles in software architecture and introduction to resm," *concept in oop. Studies*, , pp. 35(38), 8., (2015)..
- [6] J. J. & N. L. R. Barton, Scientific and Engineering C++ an Introduction with Advanced Techniques and Examples., Addison-Wesley Longman Publishing Co., Inc., (1994)..
- [7] S. Prata, C++ primer plus., Sams Publishing., (2002)..

References used in development

<https://en.cppreference.com>

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

<https://stackoverflow.com/>

B. B. L. Stroustrup, The C++ programming language: reference manual (No. BELL-CSTR-108), (1984)..

R. Grimm, C++ Core Guidelines Explained: Best Practices for Modern C++, Addison-Wesley Professional., (2022). .

Annex

```
#include "cCarpark.h"  
#include "cUserGuide.h"  
#include <iostream>  
#include <conio.h>  
  
using namespace std;
```

```
int main(void) {
    cCarpark* NCP = new cCarpark("NCP Abertawe", 3); // Create NCP with
    capacity of 3
    NCP->userguide.printUserGuide();

    while (true) {
        char input = _getch(); //accepting user input

        NCP-> update(input); // Pass the input to the car park update method

        cout << "Polling carpark components and reacting to state" << endl;
    }

    delete NCP; // Clean up allocated memory
    return 0;
}
```

FINAL GRADE

85 / 100

GENERAL COMMENTS

Perhaps you could have used some more virtual functions and overriding when using some derived class method calls. This would simplify the design.

Good use of C++ standard container objects and related classes - well done. Well chosen dynamic, built in C++ containers..

Separation of interface from implementation well done.

Comments helpful in general.

Code executes and matches demo. Essential functionality implemented with some additions-good.

Use of composition appropriate.

Your report is well written and your analysis is well contextualised within established good OO practice/principles. You have researched these well and your references are well used and appropriate. An excellent attempt overall, with limitations identified for future work, perhaps.

Well done. It is late so will be capped at 50 if no EC awarded, but a mark of 85 if submitted on time.

GRADING FORM: SMU GENERIC_GJ

AARON DAVID

0

RESEARCH-INFORMED LITERATURE

KNOWLEDGE AND UNDERSTANDING

ANALYSIS

PRACTICAL APPLICATION AND DEP.

SKILLS FOR PROFESSIONAL PRAC.