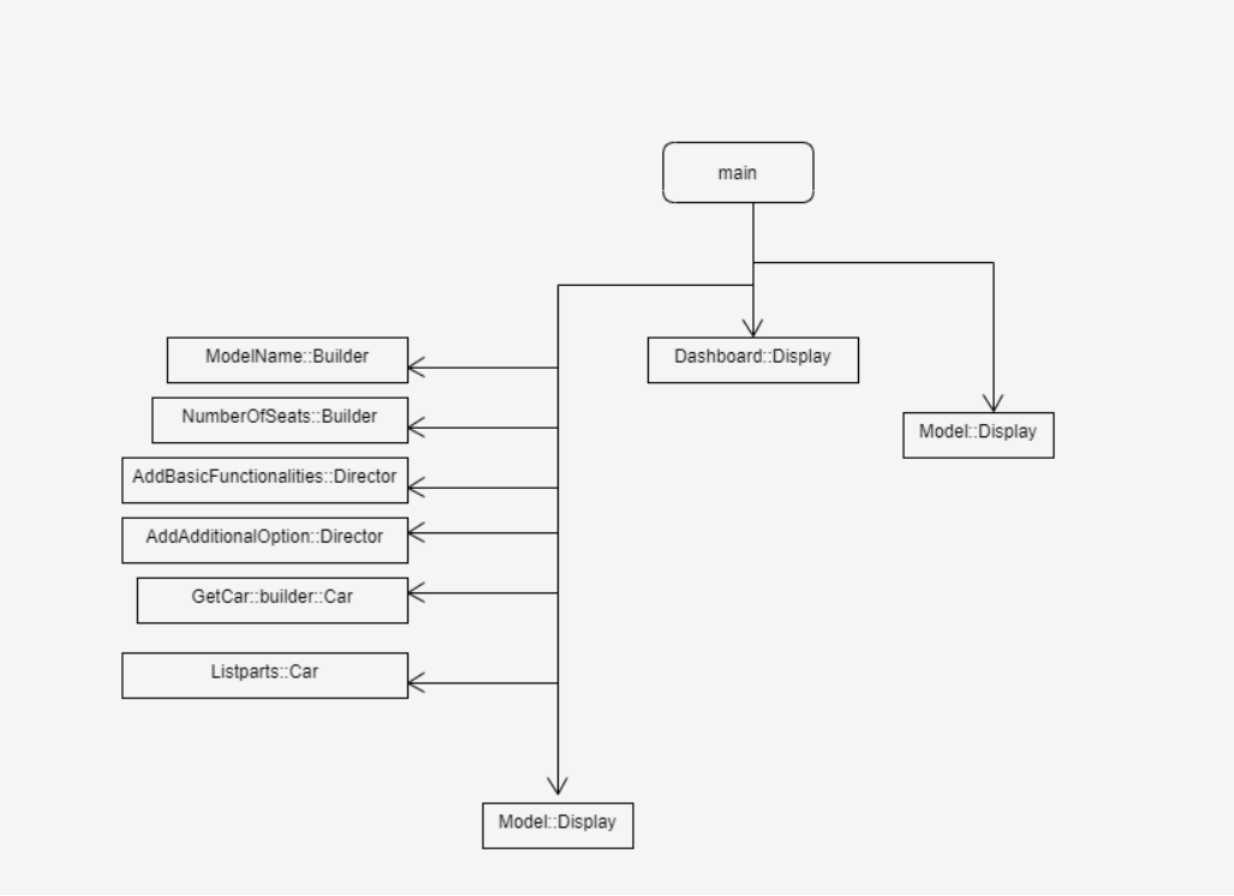
Report-3

Section 1 (Specifications):

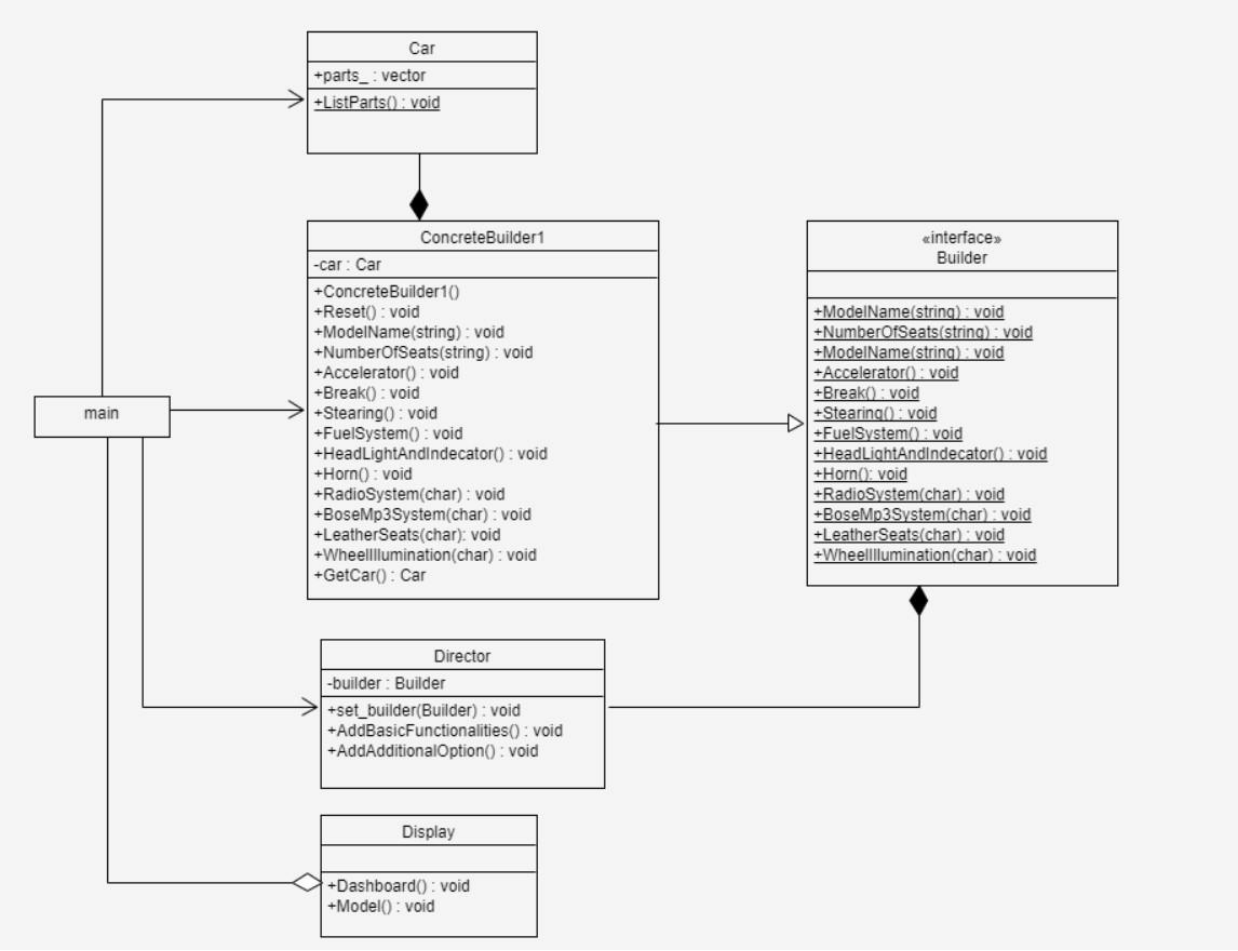
For the 3rd scenario, I have to design and implement an inventory system for this StudentCars company. This company produces two modelsof car. One is Amaze and another is Awe. Though these cars differ by the number of seats, we defined Amaze as a 7-seater car and Awe 4-seater car. Since, both this model offers the same basic functionalities of cars and some additional options which could be added by user choice. For this case I chosed builder design pattern. The way Ihave used this design pattern is, since there is an abstract class in this design pattern, Icreated an interface called Builder and 4 more classes named Car, ConcreteBuilder1, Director and Display. The Car class holds a vector where parts/functionalities of the car will be added. The ConcreteBuilder1 class extends the Builder interface. This class holds some functionalities of cars, when these functionalities/parts are called these functions will add these functions of cars. There is a class called Director. Which depends on the Builder class. This director class contrasts basic functionality and additional functionalities of cars and adds these functionalities through the object of ConcreteBuilder1 class. There is also an additional class called display. Which is used to display some info to the user.

Section 2 (Design):

2.1 Call Graph:



2.2 UML Diagram:



2.3 Implementation:

```
// The Builder interface specifies methods for creating the different parts of the cars objects
class Builder
{
public:
    // basic functionalities
    virtual void ModelName(string m) const = 0;
    virtual void NumberOfSeats(string s) const = 0;
    virtual void Accelerator() const = 0;
    virtual void Break() const = 0;
    virtual void Stearing() const = 0;
    virtual void FuelSystem() const = 0;
    virtual void HeadLightAndIndecator() const = 0;
    virtual void Horn() const = 0;
    // Additional Option
    virtual void RadioSystem(char yn) const = 0;
    virtual void BoseMp3System(char yn) const = 0;
    virtual void LeatherSeats(char yn) const = 0;
    virtual void WheelIllumination(char yn) const = 0;
};
```

```cpp
// Class Car
class Car
{
public:
    std::vector<std::string> parts_; // creating vector to add cars functionalities

    void ListParts() const
    {
        cout << "\nYour Car is Ready.";
        cout << "\nBasic functionalities: \n";
        cout << "-----------------------\n\n";
        for (size_t i = 0; i < parts_.size(); i++)
        {
            if (parts_[i] == parts_.back())
            {
                std::cout << parts_[i];
            }
            else
            {
                std::cout << parts_[i] << "\n";
            }
            if (i == 7)
            {
                cout << "\n\nAdditional functionalities: \n";
                cout << "------------------------------\n\n";
            }
        }
        std::cout << "\n\n";
    }
};
```

```cpp
/**
 * The Concrete Builder classes follow the Builder interface and provide
 * specific implementations of the building steps. So that variation of
 * Builders, implemented differently.
 */
class ConcreteBuilder1 : public Builder
{
private:
    Car *car;

    // A fresh builder instance should contain a blank product object, which is used in further assembly.
public:
    ConcreteBuilder1() // constructor
    {
        this->Reset();
    }

    void Reset()
    {
        this->car = new Car();
    }

    // All functionalities for cars
    // fucntions for Basic functionalities of cars

    void ModelName(string m) const override
    {
        this->car->parts_.push_back("Model Name               : " + m);
    }

    void NumberOfSeats(string s) const override
    {
        this->car->parts_.push_back("Number of Seats          : " + s); // adding parts to vector
    }

    void Accelerator() const override
    {
```

```
void Accelerator() const override
{
    this->car->parts_.push_back("Accelerator              :true"); // adding parts to vector
}

void Break() const override
{
    this->car->parts_.push_back("Break                    :true"); // adding parts to vector
}

void Stearing() const override
{
    this->car->parts_.push_back("Stearing                 :true");
}

void FuelSystem() const override
{
    this->car->parts_.push_back("Fuel System              :true");
}

void HeadLightAndIndecator() const override
{
    this->car->parts_.push_back("HeadLight & Indicator    :true");
}

void Horn() const override
{
    this->car->parts_.push_back("Horn                     :true");
}

// funcitons for Additional functionalities with user choice
void RadioSystem(char yn) const override
{
    string selection;
    if (yn == 'Y')
        selection = "true";
    else
```

```
        selection = "false";

        this->car->parts_.push_back("Radio System             :" + selection);
    }

    void BoseMp3System(char yn) const override
    {
        string selection;
        if (yn == 'Y')
            selection = "true";
        else
            selection = "false";

        this->car->parts_.push_back("Bose Mp3 System          :" + selection);
    }

    void LeatherSeats(char yn) const override
    {
        string selection;
        if (yn == 'Y')
            selection = "true";
        else
            selection = "false";

        this->car->parts_.push_back("Leather Seats            :" + selection);
    }

    void WheelIllumination(char yn) const override
    {
        string selection;
        if (yn == 'Y')
            selection = "true";
        else
            selection = "false";

        this->car->parts_.push_back("Wheel Illumination       :" + selection);
    }
```

```
    void WheelIllumination(char yn) const override
    {
        string selection;
        if (yn == 'Y')
            selection = "true";
        else
            selection = "false";

        this->car->parts_.push_back("Wheel Illumination       :" + selection);
    }

    // get result object of Car class
    Car *GetCar()
    {
        Car *result = this->car;
        this->Reset();
        return result;
    }
};
```

```
/**
 * The Director is only responsible for executing the building steps in a
 * particular sequence. It is helpful when producing products according to a
 * specific order or configuration. Strictly speaking, the Director class is
 * optional, since the client can control builders directly.
 */
class Director
{

private:
    Builder *builder;
    /**
     * The Director works with any builder instance that the client code passes
     * to it. This way, the client code may alter the final type of the newly
     * assembled product.
     */

public:
    void set_builder(Builder *builder)
    {
        this->builder = builder;
    }

    /**
     * The Director can construct several product variations using the same building steps.
     */

    // AddBasicFunctionalities()  to add basic functions of car
    void AddBasicFunctionalities()
    {
        this->builder->Accelerator();
        this->builder->Break();
        this->builder->Stearing();
        this->builder->FuelSystem();
        this->builder->HeadLightAndIndecator();
        this->builder->Horn();
```

```
    // AddBasicFunctionalities()  to add basic functions of car
    void AddBasicFunctionalities()
    {
        this->builder->Accelerator();
        this->builder->Break();
        this->builder->Stearing();
        this->builder->FuelSystem();
        this->builder->HeadLightAndIndecator();
        this->builder->Horn();
    }

    // AddAdditionalOption()  to add additional functions of car with users choice
    void AddAdditionalOption()
    {
        cout << "\nBasic Functionalities of car already added, now choose your additional option.\n";
        char radioSys, boseMpSys, leatherSeat, wheelIlmn;
        cout << "\nDo you want to add Radio System in your car? (Y/N)\n";
        cin >> radioSys;
        cout << "\nDo you want to add Bose Mp3 System in your car? (Y/N)\n";
        cin >> boseMpSys;
        cout << "\nDo you want to add Leather Seats in your car? (Y/N)\n";
        cin >> leatherSeat;
        cout << "\nDo you want to add Wheel Illumination in your car? (Y/N)\n";
        cin >> wheelIlmn;

        radioSys = toupper(radioSys);
        boseMpSys = toupper(boseMpSys);
        leatherSeat = toupper(leatherSeat);
        wheelIlmn = toupper(wheelIlmn);

        // calling functions of ConcreteBuilder1 class to add additional functionlities of cars
        this->builder->RadioSystem(radioSys);
        this->builder->BoseMp3System(boseMpSys);
        this->builder->LeatherSeats(leatherSeat);
        this->builder->WheelIllumination(wheelIlmn);
    }
};
```

```
// Display class to print some info
class Display
{

public:
    // Dashboard function
    void Dashboard()
    {
        system("CLS");
        cout << "-------------------------StudentCars-------------------------\n\n";
        cout << "1. View Model.\n";
        cout << "2. Build one.\n";
        cout << "3. Exit.\n";
        cout << "\nPlease Enter Your Choice: ";
    }

    // Model function
    void Model()
    {

        cout << "\n\n-------------------------StudentCars-------------------------\n";
        cout << "Cars model:\n";
        cout << "1. Amaze  -> (7 seater)\n";
        cout << "2. Awe    -> (4 seater)\n\n";

        // Since these two models are differd by the number of seats. So, we defined    model Amaze: 7 seater   &    model Awe: 4 seater
    }
};
```

```cpp
// main function
int main()
{

    // creating object of classes
    Display *display = new Display();
    Director *director = new Director();
    ConcreteBuilder1 *builder = new ConcreteBuilder1();
    director->set_builder(builder); // callling set_builder() function of Director class through pointer

    while (true)
    {
        display->Dashboard(); // calling Dashboard() function of Display class to print some info
        int optn;
        cin >> optn;

        if (optn == 1)
        {
            display->Model(); // calling Model() function of Display class to print some info
            system("pause");
        }
        //-------------------------------------
        else if (optn == 2)
        {
            display->Model(); // calling Model() function of Display class to print some info
            cout << "\nChoose your model: ";
            int model;
            cin >> model;

            if (model == 1)
            {
                // Adding some basic functionalities and additional funcitonalities for Amaze model
                builder->ModelName("Amaze");
                builder->NumberOfSeats("07");
                director->AddBasicFunctionalities();
                director->AddAdditionalOption();
                director->AddAdditionalOption();
                Car *p = builder->GetCar();
                p->ListParts();
                delete p;
                system("pause");
            }
            else if (model == 2)
            {
                // Adding some basic functionalities and additional funcitonalities for Awe model
                builder->ModelName("Awe");
                builder->NumberOfSeats("04");
                director->AddBasicFunctionalities();
                director->AddAdditionalOption();
                Car *p = builder->GetCar();
                p->ListParts();
                delete p;
                system("pause");
            }
            else
            {
                cout << "\nPlease enter correct number.\n";
            }
        }
        //-------------------------------------
        else if (optn == 3)
            return 0;
        else
        {
            cout << "\nPlease enter correct number.\n";
            continue;
        }
    }

    delete builder;
    delete director;
    return 0;
}
```

Section 3 (Discussion):

The chosen design pattern is the builder design pattern. This builder design pattern is best to design the StudentCars inventory system. Builder design patterns let you construct complex objects step by step. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. A Builder class builds the final object step by step. This builder is independent of other objects. The pattern allows you to produce different types and representations of an object using the same construction code. This builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builder. The pattern organizes object construction into a set of steps. To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object. For example, (AddBasicFucntionalities() -> which is used in this program code). Some of the construction steps might require different implementations when you need to build various representations of the product. For example, (AddAdditionalOptions() -> which is used in this program code). In this case you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the car building process.

The solid principles of OOP are Single-responsibility-Principle, Open-closed-principle, Liskov-Substitution-principle, Interface-segregation-principle and Dependency-Inversion-Principle. Though the builder pattern used in this case does not meet with the Dependency-Inversion-Principle but It fulfills the other solid principle of OOP.