1 .**Class Design**

```
#include<bits/stdc++.h>
using namespace std;
class Vehicle
{
private:
string make;
string model;
int year;
public:
  Vehicle (string make, string model, int year) :make (make),
model(model),year(year)
  {}
      string getMake ()
  {
      return make;
  }

string getModel ()
  {
      return model;
  }

int getYear ()
  {
      return year;
  }
  void setMake (string make)
  {
      this->make = make;
  }
void setModel (string model)
  {
      this->model = model;
  }
void setYear (int year)
  {
      this->year = year;
  }
  virtual void displayInfo ()
  {
cout << "Make: " << make << endl;
cout << "Model: " << model << endl;
cout << "Year: " << year << endl;
}
};
```

```
 class Car : public Vehicle {
private:
    int numDoors;
public:
    Car(string make, string model, int year, int numDoors)
        : Vehicle(make, model, year), numDoors(numDoors) {}
       void displayInfo(){
        cout << "Car Information:" << endl;
        Vehicle::displayInfo();
        cout << "Number of Doors: " << numDoors << endl;
    }
};
class Motorcycle : public Vehicle {
private:
    string type;
public:
    Motorcycle(string make, string model, int year, string type)
        : Vehicle(make, model, year), type(type) {}
        void displayInfo(){
        cout << "Motorcycle Information:" << endl;
        Vehicle::displayInfo();
        cout << "Type: " << type << endl;
    }
};

int main ()
{
      Vehicle myVehicle ("Toyota", "Camry", 2020);
      myVehicle.displayInfo ();
      myVehicle.setMake ("Honda");
    myVehicle.setModel ("Accord");
    myVehicle.setYear (2019);
      cout << "\\nUpdated Vehicle Information:" << endl;
    myVehicle.displayInfo ();
    Car myCar("Toyota", "Camry", 2020, 4);
    myCar.displayInfo();
    cout << endl;
    Motorcycle myMotorcycle("Honda", "CBR500R", 2021, "Sport");
    myMotorcycle.displayInfo();

return 0;

}
```

2.Inheritance and Polymorphism

```
#include<bits/stdc++.h>
using namespace std;
class Shape{
    public:
    virtual double calculate_area()=0;
    virtual void displayShapeInfo()=0;

};
class Circle:public Shape{
    private:
    double radius;
    public:
    Circle(double radius):radius(radius){}
    double calculate_area(){
        return 3.14*radius*radius;
    }
    void displayShapeInfo(){
        cout<<"Information about Circle"<<endl;
        cout<<"Area of Circle is "<<calculate_area()<<endl;
    }
};
class Rectangle:public Shape{
    private:
    double length;
    double breadth;
    public:
    Rectangle(double length,double
breadth):length(length),breadth(breadth){}
    double calculate_area(){
        return length*breadth;
    }
    void displayShapeInfo(){
        cout<<"Information about Rectangle"<<endl;
        cout<<"Area of Circle is "<<calculate_area()<<endl;
    }
};
int main ()
{
Circle cr(5.0);
cr.displayShapeInfo();
Rectangle rc(4,6);
rc.displayShapeInfo();
return 0;;

}
```

3 Interface and abstraction

```
#include<bits/stdc++.h>
using namespace std;
class Drawable{
  public:
  virtual void draw()=0;
};
class Circle:public Drawable{
    private:
    double radius;
    public:
    Circle(double radius):radius(radius){}
    void draw(){
        cout<<"Drawing a Circle with radius "<<radius<<endl;
    }
};
class Rectangle:public Drawable{
    private:
    double length,breadth;
    public:
    Rectangle(double length,double
breadth):length(length),breadth(breadth){}
    void draw(){
        cout<<"Drawing a Rectangle with length "<<length<<" and
breadth "<<breadth<<endl;
    }
};
class DrawingBoard{
    private:
    vector<Drawable*>obj;
    public:
    void add(Drawable* object){
        obj.push_back(object);
    }
    void drawAll(){
        for(auto it:obj){
            it->draw();
        }
    }
};
int main ()
{
DrawingBoard drw;
drw.add(new Circle(7));
drw.add(new Rectangle(4,6));
drw.drawAll();
```

```
return 0;

}
```

4.Exception Handling

```
#include<bits/stdc++.h>
using namespace std;
class InvalidYearException : public exception {
public:
    char* what() const noexcept override {
        return "Invalid year specified";
    }
};
class Vehicle {
private:
    string make;
    string model;
    int year;
public:
    Vehicle(string make, string model, int year)
        : make(make), model(model) {
        if (year < 1900 || year > 2022) {
            throw InvalidYearException();
        }
        this->year = year;
    }
    void displayInfo(){
        cout << "Make: " << make << endl;
        cout << "Model: " << model << endl;
        cout << "Year: " << year << endl;
    }
};

int main() {
    try {
        Vehicle myVehicle("Toyota", "Camry", 1800);
    } catch (InvalidYearException& e) {
        cerr << "Error: " << e.what() << endl;
    }
    try {
        Vehicle myVehicle("Honda", "Accord", 2010);
        myVehicle.displayInfo();
    } catch (const InvalidYearException& e) {
        cerr << "Error: " << e.what() << endl;
    }
```

```
    return 0;
}

5. Design Pattern
int main() {
 // Singleton Logger instance
 Logger* loggerInstance1 = Logger::getInstance();
 Logger* loggerInstance2 = Logger::getInstance();
 std::cout << "Are loggerInstance1 and loggerInstance2 the same
instance? " <<
(loggerInstance1 == loggerInstance2) << std::endl;
 // Observer design pattern for notifying subscribers
 VehicleCollection vehicleCollection;
 // Subscriber 1
 ConcreteVehicleSubscriber subscriber1;
 vehicleCollection.addSubscriber(&subscriber1);
 // Subscriber 2
 ConcreteVehicleSubscriber subscriber2;
 vehicleCollection.addSubscriber(&subscriber2);
 // Notify subscribers when a new vehicle is added
 vehicleCollection.notifySubscribers("Car1");
 vehicleCollection.notifySubscribers("Truck1");
 // Log some messages using the Singleton Logger
 loggerInstance1->log("This is a log message.");
 loggerInstance2->log("Another log message.");
 return 0;
}
```

1. **File Handling and Serialization**

```
#include<bits/stdc++.h>
using namespace std;
class Vehicle {
public:
    string make;
    string model;
    int year;
    Vehicle(string make, string model, int year)
        : make(make), model(model), year(year) {}
    void displayInfo() const {
        cout << "Make: " << make << ", Model: " << model << ",
Year: " << year << endl;
    }
};
class Inventory {
private:
```

```
    vector<Vehicle*> vehicles;

public:
    void addVehicle(Vehicle* vehicle) {
        vehicles.push_back(vehicle);
    }
    void removeVehicle(int index) {
        if (index >= 0 && index < vehicles.size()) {
            delete vehicles[index];
            vehicles.erase(vehicles.begin() + index);
        }
    }
    void displayAllVehicles() const {
        for (const auto& vehicle : vehicles) {
            vehicle->displayInfo();
        }
    }
    void serializeToFile(const string& filename) const {
        ofstream file(filename);
        if (file.is_open()) {
            for (const auto& vehicle : vehicles) {
                file << vehicle->make << "," << vehicle->model
<< "," << vehicle->year << endl;
            }
            file.close();
        } else {
            cerr << "Error: Unable to open file for writing: "
<< filename << endl;
        }
    }
    void deserializeFromFile(const string& filename) {
        ifstream file(filename);
        if (file.is_open()) {
            vehicles.clear();
            string make, model;
            int year;
            while (file >> make >> model >> year) {
                vehicles.push_back(new Vehicle(make, model,
year));
            }
            file.close();
        } else {
            cerr << "Error: Unable to open file for reading: "
<< filename << endl;
        }
    }
    ~Inventory() {
```

```
        for (auto vehicle : vehicles) {
            delete vehicle;
        }
    }
};

int main() {
    Vehicle* car1 = new Vehicle("Toyota", "Camry", 2015);
    Vehicle* car2 = new Vehicle("Honda", "Accord", 2018);
    Vehicle* car3 = new Vehicle("Ford", "Mustang", 2020);
    Inventory inventory;
    inventory.addVehicle(car1);
    inventory.addVehicle(car2);
    inventory.addVehicle(car3);
    cout << "Vehicles in Inventory:" << endl;
    inventory.displayAllVehicles();
    inventory.serializeToFile("inventory.txt");
    Inventory newInventory;
    newInventory.deserializeFromFile("inventory.txt");
    cout << "\\nVehicles in New Inventory (after
deserialization):" << endl;
    newInventory.displayAllVehicles();

    return 0;
}
```

7.Unit Testing

Unit testing means writing code that verifies individual parts, or **units,** of an application or library. A **unit** is the smallest testable part of an application. Unit tests assess code in **isolation.**

In C++ this means writing tests for methods or functions. Tests only examine code within a single object. They don't rely on external resources such as databases, web servers, or message brokers.