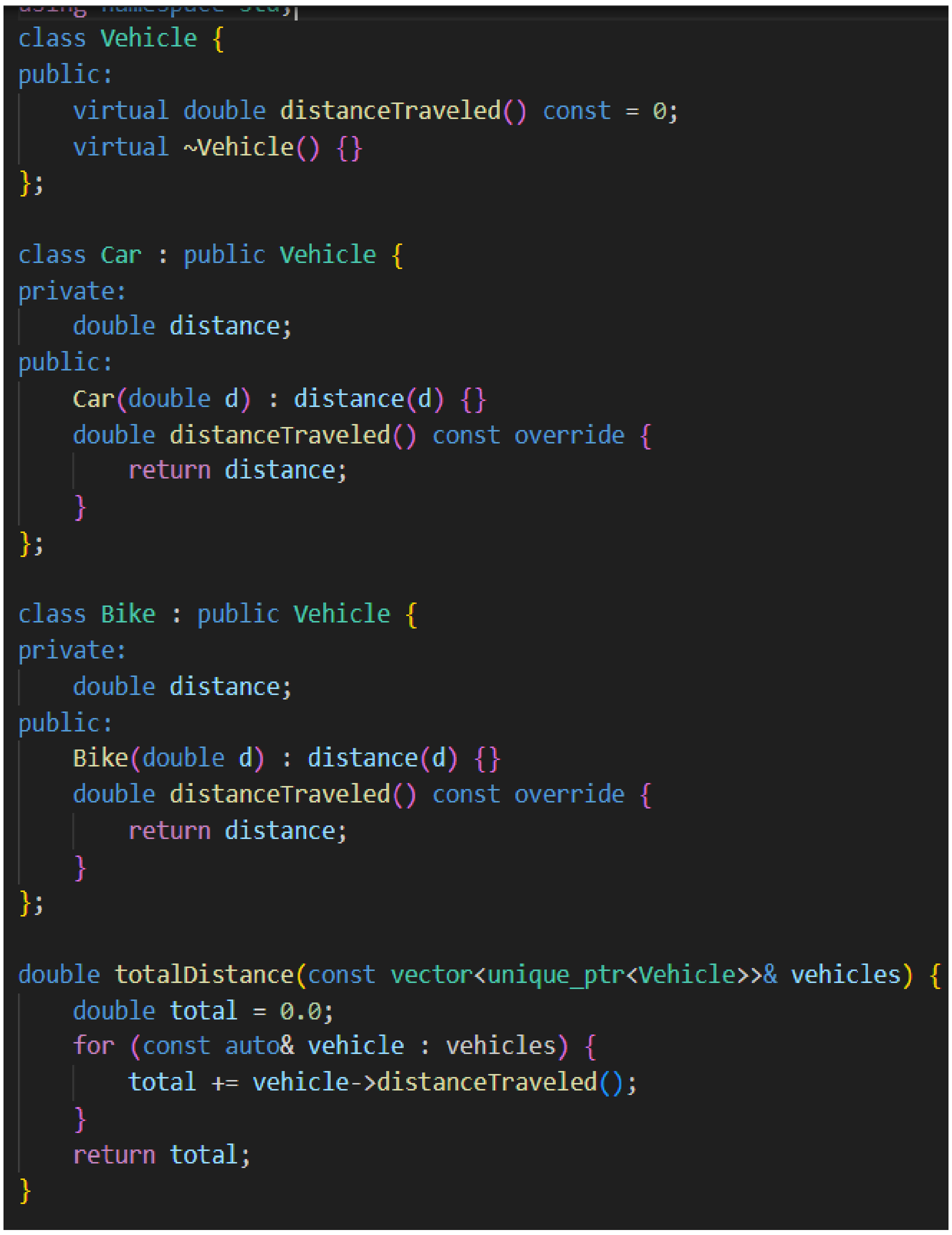
BSSE23058

How is "Program to Interface" different from "Program to Implementation"? Write

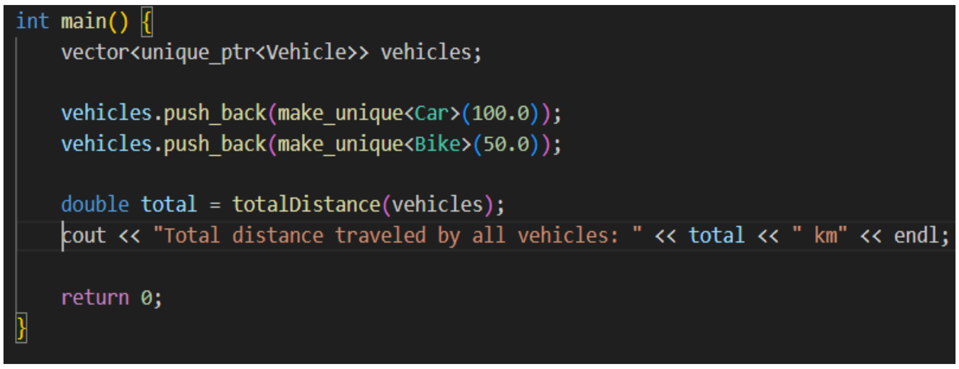
a simple code example and briefly explain your code.

Program to Interface:

Programming to an interface means that the code uses interfaces to define contracts that classes must adhere to. This allows for flexibility and modularity, as different implementations of the same interface can be used interchangeably without affecting the rest of the codebase.



```
class Vehicle {
public:
    virtual double distanceTraveled() const = 0;
    virtual ~Vehicle() {}
};

class Car : public Vehicle {
private:
    double distance;
public:
    Car(double d) : distance(d) {}
    double distanceTraveled() const override {
        return distance;
    }
};

class Bike : public Vehicle {
private:
    double distance;
public:
    Bike(double d) : distance(d) {}
    double distanceTraveled() const override {
        return distance;
    }
};

double totalDistance(const vector<unique_ptr<Vehicle>>& vehicles) {
    double total = 0.0;
    for (const auto& vehicle : vehicles) {
        total += vehicle->distanceTraveled();
    }
    return total;
}
```

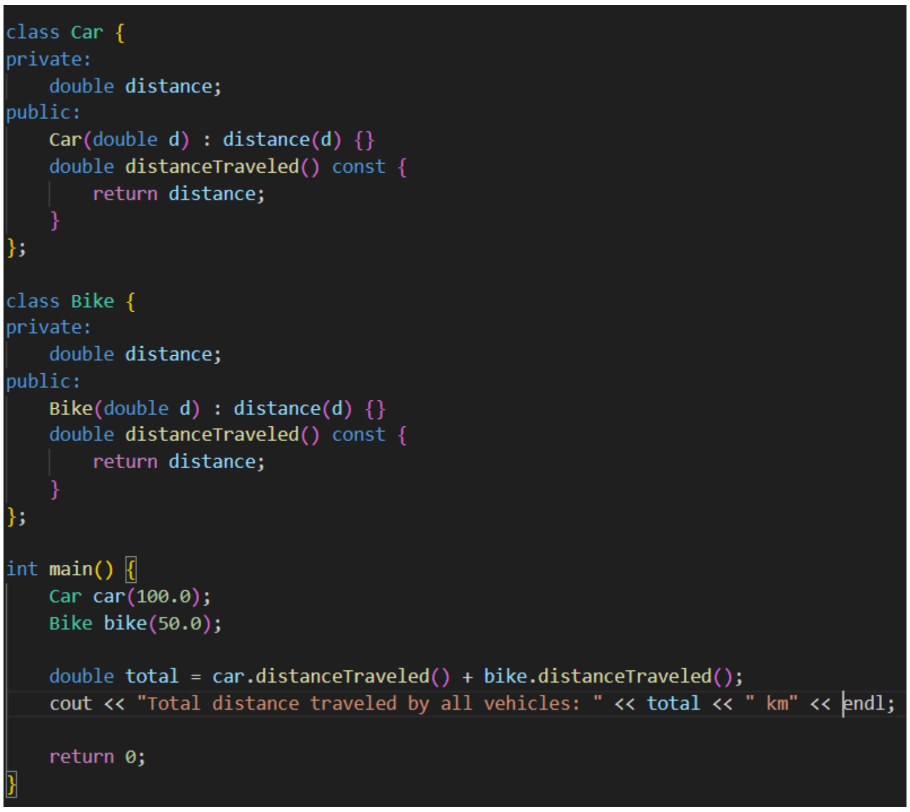


```
int main() {
    vector<unique_ptr<Vehicle>> vehicles;

    vehicles.push_back(make_unique<Car>(100.0));
    vehicles.push_back(make_unique<Bike>(50.0));

    double total = totalDistance(vehicles);
    cout << "Total distance traveled by all vehicles: " << total << " km" << endl;

    return 0;
}
```

In this example, we define a Vehicle interface with a distanceTraveled() method. Both Car and Bike classes implement this interface. We then create a totalDistance() function that takes a vector of Vehicle pointers and calculates the total distance traveled by all vehicles. In main(), we create instances of Car and Bike and store them in a vector of Vehicle pointers.

Program to Implementation:

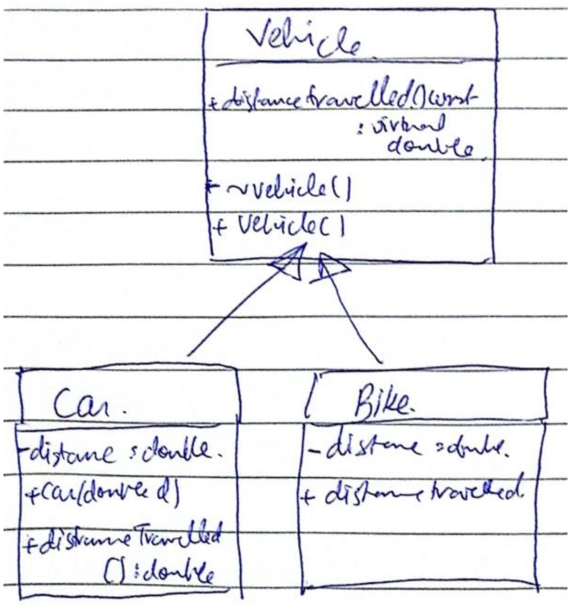
Programming to an implementation means that the code directly references and uses specific implementations of classes, rather than relying on interfaces or abstract classes.



```
class Car {
private:
    double distance;
public:
    Car(double d) : distance(d) {}
    double distanceTraveled() const {
        return distance;
    }
};

class Bike {
private:
    double distance;
public:
    Bike(double d) : distance(d) {}
    double distanceTraveled() const {
        return distance;
    }
};

int main() {
    Car car(100.0);
    Bike bike(50.0);

    double total = car.distanceTraveled() + bike.distanceTraveled();
    cout << "Total distance traveled by all vehicles: " << total << " km" << endl;

    return 0;
}
```

In this example, we directly create instances of Car and Bike in main() and call their distanceTraveled() methods to calculate the total distance traveled. There's no abstraction or interface involved, and the code is tightly coupled to the specific implementations of Car and Bike.

BSSE23058


Vehicle.
+distance travelled() const : virtual double.
+~vehicle()
+Vehicle()
Car.
-distane : double.
+car(double d)
+distance Travelled () : double
Bike.
-distane : double.
+ distance travelled.