Assignment 5: Class-Based Ride Sharing System

Nguyen The Duy Khanh

005019181

MSCS-632 - Advanced Programming Languages

Vanessa Cooper

March 29, 2025

Table of Contents

Introduction

Object-Oriented Programming (OOP) is a paradigm that organizes software design around data, or objects, rather than functions and logic. In OOP, objects represent real-world entities and the interactions between them. This paradigm is guided by four key principles: Encapsulation, Inheritance, Polymorphism, and Abstraction.

OOP in Class-Based Ride Sharing System

Encapsulation

Encapsulation is the concept of hiding an object's internal state and requiring all interaction to be performed through well-defined methods. This principle ensures that the object's data is protected from unauthorized access and modification.

- **C++ Implementation:** In the C++ code, encapsulation is achieved by using private members and public methods. For example, the Driver class keeps its assignedRides list private to prevent external access. The list of rides is accessed and modified through the addRide method, which enforces controlled access to the rides assigned to a driver. Similarly, the Rider class uses private attributes like riderID and requestedRides, ensuring that ride history can only be modified or viewed through specific methods (requestRide and viewRides).

```
// Driver class
class Driver {
    private:
        int driverID;
        string name;
        double rating;
        vector<Ride*> assignedRides;
    public:
        Driver(int id, string n, double r) : driverID(id), name(n), rating(r) {}
        void addRide(Ride* ride) { assignedRides.push_back(ride); }
        void getDriverInfo() const {
            cout << "Driver ID: " << driverID << "\nName: " << name << "\nRating: " << rating << "\nRides: " << assignedRides.size() << "\n";
        }
};
```

Figure 1: Encapsulation within Driver in C++

- **Smalltalk Implementation**: Smalltalk also emphasizes encapsulation by defining private variables and using accessor methods to interact with them. In the Driver class, the assignedRides collection is private, and external code must interact with it through methods like addRide. Smalltalk's approach to encapsulation is similar to C++ in that it hides the data but allows interaction through carefully designed methods.

```
Object subclass: Driver [
    | driverID name rating assignedRides |

    Driver class >> new: id name: nm rating: rtg [
        ^ super new initialize: id name: nm rating: rtg
    ]

    Driver >> initialize: id name: nm rating: rtg [
        driverID := id.
        name := nm.
        rating := rtg.
        assignedRides := OrderedCollection new.
    ]

    Driver >> addRide: ride [
        assignedRides add: ride.
    ]

    Driver >> getDriverInfo [
        ^ 'Driver ID: ', driverID asString, ' | Name: ', name,
    ]
]
```

Figure 2: Encapsulation within Driver in Small talk

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass, primarily through method overriding. It enables one interface to represent different underlying forms (types).

- **C++ Implementation:** Polymorphism is demonstrated when the fare() method is called on objects of different ride types (e.g., StandardRide and PremiumRide). Although both classes are instances of the Ride class, calling fare() will execute the overridden method in the respective class. This is made possible by using virtual functions and dynamic binding, which ensure that the correct version of the method is called based on the actual object type, not the pointer type.

```cpp
// Derived class StandardRide
class StandardRide : public Ride {
public:
    StandardRide(int id, string pickup, string dropoff, double dist) : Ride(id, pickup, dropoff, dist) {}
    double fare() const override { return distance * 1.5; } // Standard rate per mile
};

// Derived class PremiumRide
class PremiumRide : public Ride {
public:
    PremiumRide(int id, string pickup, string dropoff, double dist) : Ride(id, pickup, dropoff, dist) {}
    double fare() const override { return distance * 2.5; } // Premium rate per mile
};
```

Figure 3: Polymorphism within Standard Ride and Premium Ride in C++

- **Smalltalk Implementation**: Smalltalk handles polymorphism through dynamic method dispatch. In the case of Ride, both StandardRide and PremiumRide objects can be treated as Ride objects. When the fare or rideDetails method is called, Smalltalk dynamically determines which method implementation to use based on the actual class of the object, not the variable type. This makes polymorphism seamless in Smalltalk.

```
Ride subclass: StandardRide [
    StandardRide >> fare [
        ^ distance * 1.5
    ]

    StandardRide >> rideDetails [
        ^ super rideDetails, ' Fare: $', (self fare) asString, ' (Standard Ride)'
    ]
]

Ride subclass: PremiumRide [
    PremiumRide >> fare [
        ^ distance * 2.5
    ]

    PremiumRide >> rideDetails [
        ^ super rideDetails, ' Fare: $', (self fare) asString, ' (Premium Ride)'
    ]
]
```

Figure 4: Polymorphism within Standard Ride and Premium Ride i in Small talk

Inheritance

Inheritance allows a class (child class) to inherit properties and behaviors (methods) from another class (parent class), promoting reusability and hierarchical relationships between classes.

- **C++ Implementation:** In the C++ code, inheritance is used to create subclasses from the base class Ride. Both StandardRide and PremiumRide inherit from Ride and have access to its attributes like rideID, pickupLocation, and dropoffLocation. These subclasses then override the fare() method to implement their specific behavior (pricing rules). The use of the virtual keyword ensures that the correct method is invoked depending on the object type (polymorphism).

```
class Ride {
    protected:
        int rideID;
        string pickupLocation;
        string dropoffLocation;
        double distance;
    public:
        Ride(int id, string pickup, string dropoff, double dist) : rideID(id), pickupLocation(pickup), dropoffLo
        virtual double fare() const = 0; // Pure virtual function
        virtual void rideDetails() const {
            cout << "Ride ID: " << rideID << "\nPickup: " << pickupLocation << "\nDropoff: " << dropoffLocation
        }
        virtual ~Ride() {}
};

// Derived class StandardRide
class StandardRide : public Ride {
public:
    StandardRide(int id, string pickup, string dropoff, double dist) : Ride(id, pickup, dropoff, dist) {}
    double fare() const override { return distance * 1.5; } // Standard rate per mile
};
```

Figure 5: Inheritance within Ride in C++

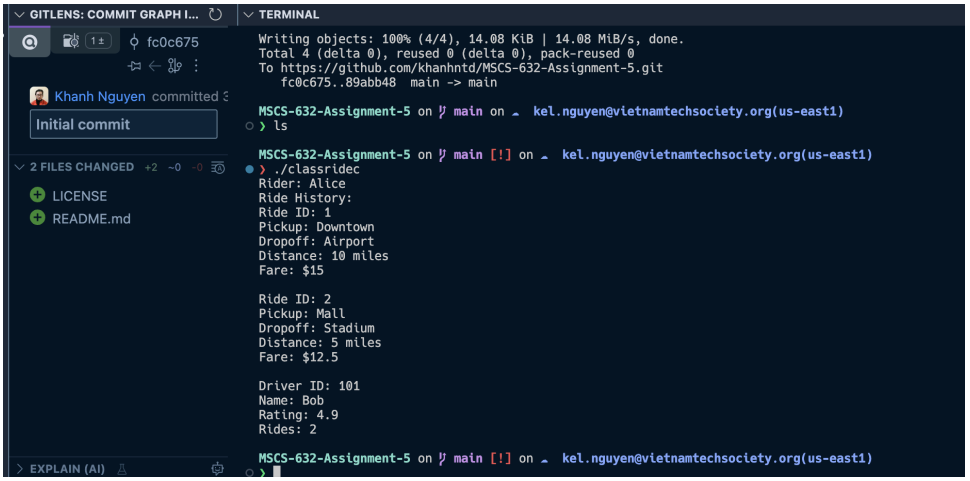
- **Smalltalk Implementation:** In Smalltalk, inheritance is similarly utilized by defining subclasses. For example, StandardRide and PremiumRide both inherit from the base Ride class and inherit common attributes and methods. The subclasses can override the fare method to define their specific fare calculation logic. Smalltalk makes it easy to extend and customize behavior via inheritance by allowing subclass methods to override or augment parent methods.

```
Object subclass: Ride [

    Ride class >> new: id pickup: pickup dropoff: dropoff distance:
        ^ super new initialize: id pickup: pickup dropoff: dropoff d
    ]

    Ride >> initialize: id pickup: pickup dropoff: dropoff distance:
        rideID := id.
        pickupLocation := pickup.
        dropoffLocation := dropoff.
        distance := dist.
    ]

    Ride >> fare [
        self subclassResponsibility
    ]

    Ride >> rideDetails [
        ^ 'Ride ID: ', rideID asString, ' | Pickup: ', pickupLocatio
    ]
]

Ride subclass: StandardRide [
    StandardRide >> fare [
        ^ distance * 1.5
    ]
```

Figure 6: Inheritance within Ride in SmallTalk

Sample Output

When running the following command with , the artifact will show:



References

Khanh Nguyen. https://github.com/khanhntd/MSCS-632-Assignment-5