Assignment 5: Developing a Class-Based Ride Sharing System

Github link:- https://github.com/ndhinaharan36295/MSCS-632_Assignment-5

Encapsulation

Encapsulation is an Object Oriented Programming (OOP) principle that refers to the bundling of data and methods operating on the data within a single unit. This principle involves restricting direct access of some of the fields in an object, achieved usually through access modifiers (Stackify, 2020)

```
class Driver {
    private:
        string driverID;                          // Unique identifier for the driver
        string name;                              // Name of the driver
        double rating;                            // Driver's rating
        vector<shared_ptr<Ride>> assignedRides;   // List of rides assigned to the driver

    public:
        Driver(string id, string name, double rating);
        void addRide(shared_ptr<Ride> ride);
        void getDriverInfo() const;
};
```

5.1 Encapsulation in C++ implementation

Figure 5.1 demonstrates the Encapsulation principle in my C++ implementation. As shown above, in C++ implementation, encapsulation is enforced through access specifiers such as private, protected, and public. The methods in the above class are all defined as public, and the fields of the class are defined as private and accessible only from within this class.

```
// Method to add a ride to the driver's list of assigned rides
void Driver::addRide(shared_ptr<Ride> ride) {
    assignedRides.push_back(ride);
}
```

Figure 5.2 Public ‘addRide’ method in C++ implementation

As shown in Figure 5.2, the ‘assignedRides’ field which is defined as a private field is accessible only through the public method ‘addRide’.

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

    "Initialize the Driver instance with ID, name, rating, and an empty list of assigned rides"
    initializeWithID: id name: driverName rating: rate [
        driverID := id.
        name := driverName.
        rating := rate.
        assignedRides := OrderedCollection new.
    ]

    "Add a ride to the driver's list of assigned rides"
    addRide: ride [
        assignedRides add: ride.
    ]
```

Figure 5.3 Encapsulation in Smalltalk implementation

Encapsulation in Smalltalk is conceptually enforced through instance variables and method access. There are no access specifiers (like in C++), but instance variables like ‘assignedRides’ are only accessed through methods, as shown in Figure 5.3.

Inheritance

Inheritance is an OOP principle that establishes hierarchical relationships between classes, which allows the reuse of code by referencing the behaviors and data of an object. An inherited class is called a subclass, and the class being inherited is called either a parent class or superclass (Codecademy Team, 2023)

```cpp
class PremiumRide : public Ride {
    public:
        PremiumRide(string id, string pickup, string dropoff, double dist);
        double fare() const override;
        void rideDetails() const override;
};
```

Figure 5.3 Inheritance in C++ implementation

Figure 5.3 shows the demonstration of inheritance in my C++ implementation. Here, I defined a parent class - ‘Ride’, which is then inherited by a subclass - ‘PremiumRide’ and also ‘StandardRide’. As you can see here, the methods fare() and rideDetails() are defined in Ride and then inherited into PremiumRide. The method definitions can be reused from the parent class Ride, if needed. If explicit customization of these methods is required for this particular subclass, we need to ‘override’ those methods, as shown in Figure 5.3

```smalltalk
"PremiumRide Class"
Ride subclass: PremiumRide [
    "Override fare calculation for premium rides (rate is 3.0 per unit distance)"
    fare [
        ^ distance * 3.0.
    ]

    "Display details specific to premium rides"
    rideDetails [
        '[Premium Ride]' displayNl.
        super rideDetails.
    ]
]
```

Figure 5.4 Inheritance in Smalltalk implementation

Figure 5.4 shows the demonstration of inheritance in my Smalltalk implementation. As you can see here, the methods fare() and rideDetails() are defined in Ride and then inherited into PremiumRide. The methods are overridden for the PremiumRide class, as shown in Figure 5.4.

Polymorphism

Polymorphism is an OOP principle that allows us to access objects of different types through the same interface. Each type can provide its own independent implementation of this interface. This enables a dynamic behavior of the object based on its type (Thorben, 2025)

```cpp
class Ride {
    protected:
        string rideID;              // Unique identifier for the ride
        string pickupLocation;      // Pickup location for the ride
        string dropoffLocation;     // Dropoff location for the ride
        double distance;            // Distance of the ride

    public:
        Ride(string id, string pickup, string dropoff, double dist);
        virtual double fare() const;
        virtual void rideDetails() const;
        virtual ~Ride();
};
```

Figure 5.5 Polymorphism in C++ implementation

Figure 5.5 shows the demonstrations of polymorphism in my C++ implementation. As shown here, in C++, polymorphism is enforced through the ‘virtual’ keyword. The ‘Ride’ class defines virtual fare() and rideDetails() methods.

```cpp
class StandardRide : public Ride {
    public:
        StandardRide(string id, string pickup, string dropoff, double dist);
        double fare() const override;
        void rideDetails() const override;
};

// Method to calculate the fare for a premium ride (overriding the base method)
// Premium rides have a fare rate of 2.5 per unit distance
double StandardRide::fare() const {
    return distance * 2.5;
}
```

```cpp
class PremiumRide : public Ride {
    public:
        PremiumRide(string id, string pickup, string dropoff, double dist);
        double fare() const override;
        void rideDetails() const override;
};

// Method to calculate the fare for a premium ride (overriding the base method)
// Premium rides have a fare rate of 3.0 per unit distance
double PremiumRide::fare() const {
    return distance * 3.0;
}
```

Figure 5.6 Dynamic Implementation of virtual methods in C++

Figure 5.6 shows the implementations of the virtual ‘fare()’ method in each of the subclasses. As shown here, the methods are overridden and explicitly implemented in the specific subclasses - StandardRide and PremiumRide. This shows the dynamic behavior of the ‘fare()’ method when invoked from each of the subclasses.

```
"StandardRide Class"
Ride subclass: StandardRide [
    "Override fare calculation for standard rides (rate is 2.5 per unit distance)"
    fare [
        ^ distance * 2.5.
    ]

    "Display details specific to standard rides"
    rideDetails [
        '[Standard Ride]' displayNl.
        super rideDetails.
    ]
]

"PremiumRide Class"
Ride subclass: PremiumRide [
    "Override fare calculation for premium rides (rate is 3.0 per unit distance)"
    fare [
        ^ distance * 3.0.
    ]

    "Display details specific to premium rides"
    rideDetails [
        '[Premium Ride]' displayNl.
        super rideDetails.
    ]
]
```

Figure 5.7 Polymorphism in Smalltalk implementation

Figure 5.7 shows the demonstrations of polymorphism in my Smalltalk implementation. The ‘Ride’ class defines fare() and rideDetails() methods, and the subclasses override the implementations within each subclass.

```
"Display the details of the ride, including ID, locations, distance, and fare"
rideDetails [
    ('Ride ID: ', rideID) display.
    (', Pickup: ', pickupLocation) display.
    (', Dropoff: ', dropoffLocation) display.
    (', Distance: ', distance printString, ' miles') display.
    (', Fare: $', (self fare) printString) displayNl.
]
```

Figure 5.8 Dynamic usage of the fare method in Smalltalk

As shown in Figure 5.8, the 'fare' method is invoked with the 'self' keyword. This ensures the dynamic behavior of the 'fare' method based on the implementations on each of the subclasses.

Sample Output

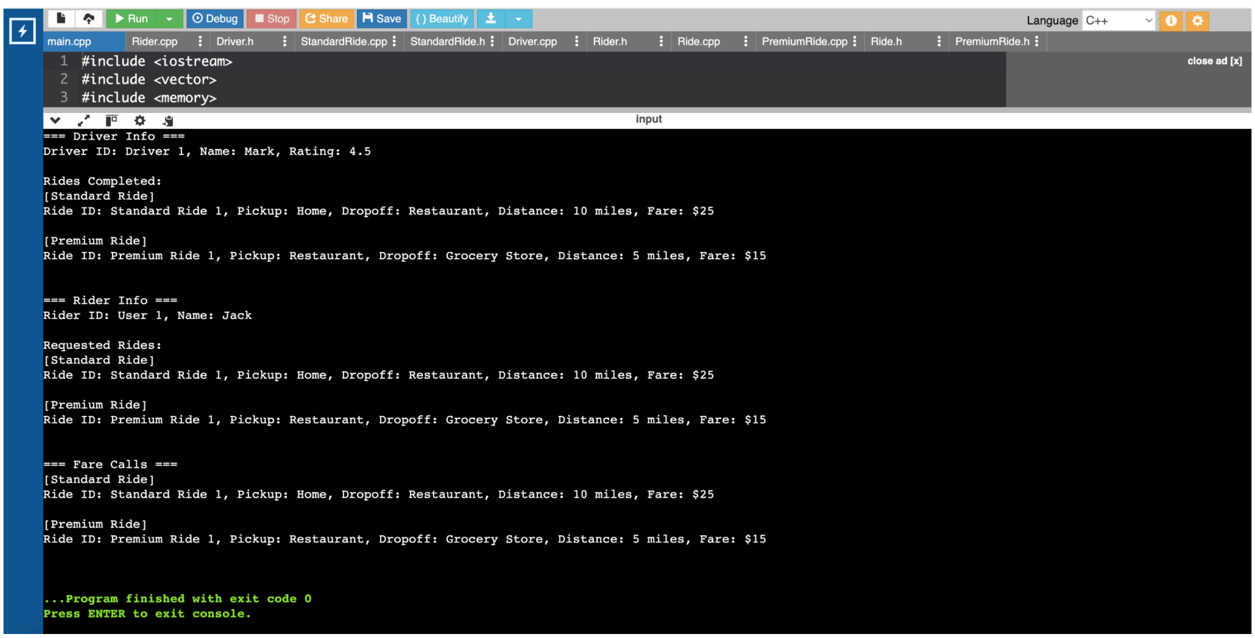

Figure 5.9 Sample output of the C++ implementation

Figure 5.9 shows the output of some sample inputs defined in the **main.cpp** file in my C++ implementation.

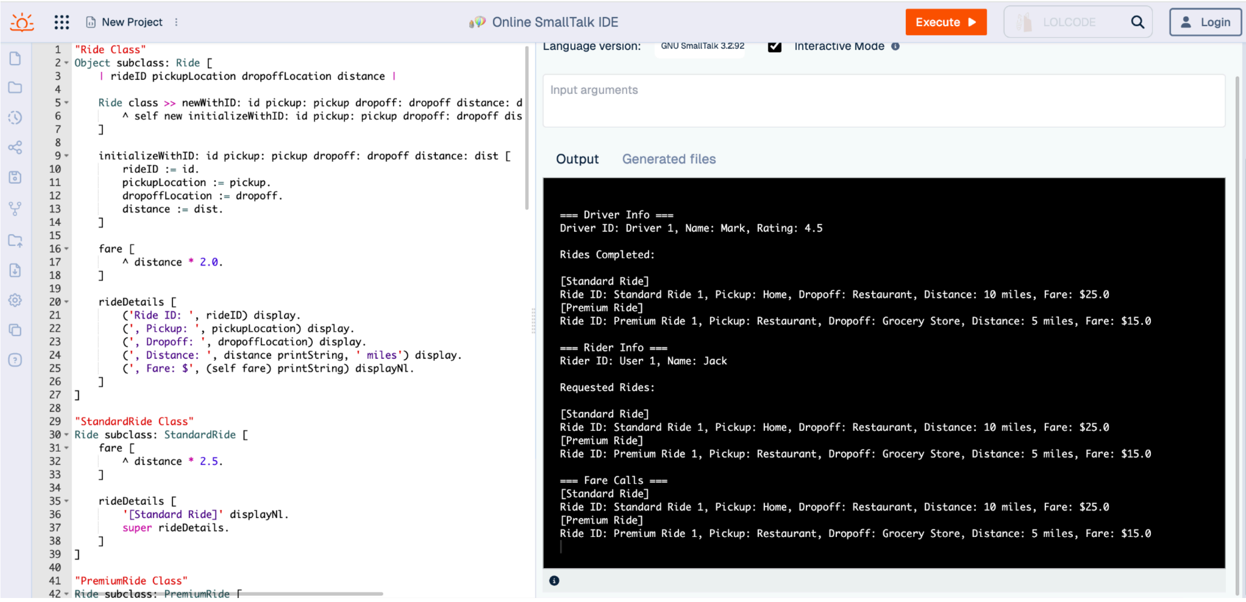

Figure 5.10 Sample output of the Smalltalk implementation

Figure 5.10 shows the output of some sample inputs defined in the **ride_sharing_system.st** file in my Smalltalk implementation.

References

Codecademy Team. (2023, July 11). *What is inheritance in object-oriented programming?* Codecademy. https://www.codecademy.com/resources/blog/what-is-inheritance/

Stackify. (2020, February 18). *OOP concept for beginners: What is encapsulation?* https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/

Thorben. (2025, February 10). *OOP concepts for beginners: What is polymorphism*. Stackify. https://stackify.com/oop-concept-polymorphism/