1.GitHub Link: https://github.com/vtadigotla35101/MSCS-632_Assignment5

2.Execution output in C++:

Rider Name: Vishnu

Driver: Tadigotla

main.cpp

```
111 };
112
113 // --- 5. Main Function (System Functionality) ---
114 int main() {
115     // Instantiate Driver and Rider
116     Driver* myDriver = new Driver("D001", "Tadigotla", 4.9);
117     Rider* myRider = new Rider("R99", "Vishnu");
118
119     // 5. Demonstrate Polymorphism by storing different types in a list
120     std::vector<Ride*> systemRides;
121
122     systemRides.push_back(new StandardRide("Trip-STD", "Cinemark", "AMC", 15.2));
```

input

```
=== Rider History: Vishnu ===
Standard Ride [ID: Trip-STD] From: Cinemark To: AMC (15.2 miles)
Premium Ride  [ID: Trip-PRE] From: Chipotle To: Chick-Fil-A (5.5 miles)
Standard Ride [ID: Trip-STD2] From: T-Mobile To: Verizon (8 miles)

=== Driver Profile ===
Name: Tadigotla | Rating: 4.9/5.0
Completed Rides:
Standard Ride [ID: Trip-STD] From: Cinemark To: AMC (15.2 miles) | Fare: $19.00
Premium Ride  [ID: Trip-PRE] From: Chipotle To: Chick-Fil-A (5.50 miles) | Fare: $18.75
Standard Ride [ID: Trip-STD2] From: T-Mobile To: Verizon (8.00 miles) | Fare: $10.00


...Program finished with exit code 0
Press ENTER to exit console.
```

3.Execution output in Smalltalk:

Rider Name: Tadigotla

Driver: Vishnu

```
    rideDetails [
        ^ 'Premium  ', super rideDetails
    ]

'--- 3. Driver Class (Encapsulation) ---'
Object subclass: Driver [
    | name assignedRides |

    Driver class >> new: aName [
        ^ (super new) init: aName
    ]

    init: aName [
        name := aName.
        assignedRides := OrderedCollection
    ]

    addRide: aRide [
        assignedRides add: aRide
    ]

    getDriverInfo [
        Transcript show: '=== Driver Profile
```

STDIN

Input for the program (Optional)

Output:

```
--- Rider History: Tadigotla (R-99) ---
  - Standard Ride ID: Trip-STD | Distance: 12.5 miles
  - Premium  Ride ID: Trip-PRE | Distance: 25.0 miles
  - Standard Ride ID: Trip-STD2 | Distance: 8.0 miles

--- Driver Profile ---
Name: Vishnu
Completed Rides:
  - Standard Ride ID: Trip-STD | Distance: 12.5 miles | Fare: $12.5
  - Premium  Ride ID: Trip-PRE | Distance: 25.0 miles | Fare: $80.0
  - Standard Ride ID: Trip-STD2 | Distance: 8.0 miles | Fare: $8.0
```

UC Student ID: 005035101

This report analyzes the implementation of the three core pillars of Object-Oriented Programming like Encapsulation, Inheritance, and Polymorphism within a Ride Sharing System developed in both C++ and GNU Smalltalk.

1. Encapsulation: Data Hiding and Integrity

Encapsulation is about bundling data with the methods that operate on it and restricting direct access to prevent accidental corruption.

- **In C++:** I have used the private keywords. By making assignedRides private in the Driver class, we ensure that external code cannot bypass our logic like skipping a validation step.
- **In Smalltalk:** Encapsulation is enforced by the language design itself. Instance variables are always private to the object. The only way to interact with an object's state is by sending it a message.

2. Inheritance: Code Reuse and Hierarchy

Inheritance allows us to define a general class and then create specialized versions of it.

- **In C++:** I have used the : syntax like class StandardRide : public Ride. This creates a rigid relationship that the compiler checks. The subclasses inherit the members of the base class, reducing code duplication for shared attributes like rideID.
- **In Smalltalk:** I have used the subclass: message. Because it is a live environment, you can actually modify the base class while the program is running, and the subclasses will immediately reflect those changes.

3. Polymorphism: Flexible Behavior

Polymorphism allows a single interface to behave differently depending on the object it is called upon.

- **In C++:** This is achieved through Virtual Functions. When you store StandardRide and PremiumRide objects in a std::vector<Ride>, C++ uses a vtable which is a virtual table to look up the correct function address at runtime.
- **In Smalltalk:** This is achieved via Dynamic Message Passing. When you send the message calculateFare to an object in a collection, the Smalltalk virtual machine searches the object's class for a matching method.

Conclusion

While both languages successfully implement the core OOP principles, their philosophies differ. C++ prioritizes explicit control and compile-time efficiency, making it ideal for high-performance SRE/DevOps tools. Smalltalk prioritizes simplicity and runtime flexibility, treating every interaction as a conversation between objects.