**Analysis of OOP principles:-**

1. **Encapsulate what varies:-**

Varies means “may change over time due to changing requirements.” This is done to make a code such that a change in some parameter won’t lead to changes at a lot of places in the code. The setter and getter methods are used in almost all classes to implement this. For example in the **city.txt** if the rows or columns change, the changes will automatically occur at all places in the code since it is not hard coded. Also the data is divided in different classes and hence changes are to be done only in the classes rather than in the main method. But since we haven’t made the data members private at some places in certian classes, there are some flaws in our code which can be rectified.

While calculating fare we have hard coded the available drivers criteria as well as the fare rate per meter which leads to violation of the above principle because if in the future the fare rates have to be revised the amount has to be changed at all the places in the code. Similarly for the number of drivers criteria the total drivers may increase in the future and the minimum drivers for which fare rates are changed may increase. We could have made them global variables but that would imply that the rate is same at all places which is not the case always.

Also we have not used any other function in the main class which has decreased the readability of the code but we could have used functions to improve the readability. For example asking for user current location and destination should be implemented in the customer class rather than in main method.

Encapsulation is done to limit the unnecessary details in the code.

1. **Favor composition over Inheritance:-**

Composition means “has a” relation whereas inheritance means “is a” relation. Composition is advisable in case we don’t need all the methods of a class. In our **Customer class** we have made an object of **Landmark** rather than extending landmark because customer “has a” pickup Landmark. Same is the case with driver class.

In the city class we have made a **list of landmarks** since cities “has a” list of landmarks rather than saying city “is a” list of landmarks. Similarly in the journey class we have made objects of other classes rather than extending the classes.

We have extended **cab** class in **driver** which we shouldn’t have done because the driver has a cab. Rather we should have made an object of the cab in the driver class.

The disadvantage with inheritance is that we can’t change the implementation inherited from super classes at runtime and also it violates the encapsulation principle.

1. **Program to an interface not implementation :-**

It basically emphasizes that we should use abstraction rather than implementation directly. Interface can be used to control the way any class will behave and hence is useful to have control over the objects. It is useful if a method can change drastically with conditions in the future.

For example in the driver class we have used the **calculateEta** method which calculates the estimated time of arrival based on very simple formula of time=distance/speed. But since we know there are various factors which effect the time like traffic on that day, weather conditions, it would be better if we would have made an interface **calculateETA()** and then made classes **calculateETA\_dueToTraffic** and **calculateETA\_dueToWeather** and **calculateETA\_dueToSpeed** which will extend this interface and later an object of the required classes based on the available conditions can be made in the driver class to calculate actual ETA. As a customer all we have to worry about is the **calculateETA**() rather than the changes in it due to external conditions.

Also, in our main method, the Customer and the Driver are both logging in using an almost similar procedure. We could have created an abstract method which helped the user login into the app and modified it according to customer and driver objects. This would have made our code less redundant and improved its readability.

1. **Classes should be open for extension and closed for modification:-**

It means to create system in which new features are added by adding new code without changing any of the previous code written. Perfect implication of this rule is nearly impossible because it requires complete foresight about what will be required in the future.

For example while writing this code we have added the cancelBooking method in customer class at last moment without changing any other method. It basically means to avoid constants in the code that are prone to change in future.

Another example could be that if we want to change the availability status of driver in the database, we would have to add this method in our class but it would change the way we are handling the files in our code right now, because to do modification in a file we have to create a separate copy of the file with given changes. Similar problem occurs if we want to change the current location of driver in the database.

**Observations regarding a design pattern being followed :-**

According to our understanding of the builder pattern, we feel that it is being followed to a certain extent in our code. This is because almost all of the classes we have implemented contain complex Constructors which have multiple arguments. If the customer accessed these classes directly, they would have to follow a very robust procedure if they want the program to run correctly. However, we have tried to make this application easy to use for the customer by trying to follow a concept which is similar to the builder pattern.

The journey of a customer between their pickup and destination landmarks consists of many parts like the current location of driver, speed of the driver, availability of the driver, the exact coordinates of the locations, distance between the 2 landmarks, etc. However, the customer need not worry about all these small details. All the customer has to do is enter their current location and their destination and the rest is handled by the code.

When they enter the names of these locations, they only have to write the names of the landmarks and need not write the x and y coordinates. A landmark object is created and is initialized with the coordinates which match with the database. Using such Landmark objects from both the customer and the driver objects, we are able to display things like the ETA and the fare.

Even though we have not written the ‘Builder’ interface and ‘ConcreteBuilder’ and ‘Director’ classes in our code, the functioning of our code made us relate to this design pattern the most.

**Future work possible :-**

1. Option to calculate the fare according to the time of booking cab.
2. The availability status of the driver is changed only in the code and not in the database whereas ideally it should change in both places.
3. Option to login any driver rather than only the one which is booked by the customer can be provided.
4. No option to change the driver’s available status in the database i.e. once the driver is marked unavailable in database there is no way to make the driver available again.
5. Option to add new driver.
6. Option to give rating for the selected driver.
7. Option to add new landmarks rather than the one mentioned in the database.
8. Option to share the cab.
9. Option to select the number of seats in the car.
10. Option to give feedback.
11. Can be linked with google maps to show the fastest route.
12. Fare hike in case of bad weather by linking with weather app.
13. Password can be made safer by ensuring certain conditions are met like it must contain a capital letter and some number.
14. Option to send OTP to confirm the authenticity of the user.
15. Option for various payment options like netbanking, paytm, GPay, cash etc. by linking with appropriate apps.
16. Display the drivers details like his mobile number, languages spoken by him etc. to the user once the ride has been confirmed by the driver.
17. Display continuously the location of driver rather than just the ETA.
18. After the journey is over the current location of the driver should be changed.