GITHUB LINK: https://github.com/anushka-2143/OOPAssignment

Analysis of OOP Principles and Design Pattern

1) Encapsulate what Varies

Here, by "varies" we mean that something that may change over time due to changing requirements. If that "something is there in our code" then one should separate that piece of code. Because if the code is highly interconnected and if that "something" is used frequently at many places then it will be difficult to implement the changes, if one wants to make any. That's why we should Encapsulate what may vary in future.

In A* Algorithm, the heuristic Function may vary as per the requirements of the user. As of now, we have implemented it using Euclidean Distance as asked in the problem statement. But if later on, we need to change it to Manhattan Distance or any other method then it will be difficult to implement the changes, as it has been used at many places in the AStar Class. It would have been better if we would have encapsulated the Heuristic function. So that in future, if we want to provide user with different ways to calculate Heuristic function then we can append different cases in that separate piece of code.

2) Favor composition over inheritance

One should prefer Composition over Inheritance because Composition is easy to modify later, multiple inheritance is not possible in java (we can only extend one class in java), inheritance break encapsulation (super class behavior changes the behavior of sub class).

In this project, we have made use of inheritance while making Dijkstra and AStar Subclasses inherited from the super class ShortestPathAlgorithm. In future, if any developer wants to add any other subclass for new Algorithm to find shortest path, then if he would add any new method in the superclass ShortestPathAlgorithm as per the requirement of new subclass then it would affect the functioning of Dijkstra and AStar Subclasses or if there is any function in the superclass

ShortestPathAlgorithm which is not required in the newly created subclass then the inheritance can cause problems. This is the reason we couldn't declare the calculateHeuristic method inside the superclass. Instead, we could have favoured composition over inheritance then adding necessary and common methods among all the subclasses in the ShortestPathAlgorithm would have sufficed and for adding new methods in the subclasses which are not common among all the subclasses, we could have created new classes and then create their references in the subclass. This would have solved the problem of Encapsulation what varies for the Heuristic function as well.

3) Program to interfaces not implementations

We have added 4 classes each of which has a supertype. Two of the classes have interface as a supertype and two of the classes have abstract classes as a supertype. Currently there are only two classes that implement these supertypes, so we don't get much benefit from creating a supertype. But, on adding further functionalities like different algorithms, we can store the reference of the supertype and then dynamically link these reference variables with the objects of a subclass. This is how programming to an interface will help us dynamically access different algorithms of the code.

4) Classes should be open for extension but closed for modification

This principle states that we should be able to extend the module depending on the changing requirements and new feature requirements, without causing any changes to the source code. This can be achieved by making the source code as abstract classes and then extending it to the sub classes in which new modifications can be added or making new sub classes according to the changing requirement which extend the abstract class, so the features of the base class do not get affected by the new changes.
We had implemented this in our program. As we had made an abstract class ShortestPathAlgorithm which contain all the general features required by any algorithm to calculate the shortest distance. So, in future if we want to calculate distance by using any other algorithm, we can directly add that algorithm as the sub class which can extend the ShortestPathAlgorithm for using the general features defined in it. And can have more of its own features in its own class. So, because of this even after adding a completely new algorithm into the code, we don't need to modify the existing code of Dijkstra and A star algorithm.

Design Pattern

The factory pattern is one of the most widely used design patterns in Java. This model provides one of the best ways to create an object. We have used basic functioning of this method design pattern in our code by City interface and concrete classes CityDijkstra and CityAStar implementing this interface. And also, abstract class ShortestDistAlgorithm extended by classes AStar and Dijkstra. Then we are forming object of concrete classes CityDijkstra and CityAStar inside the classes AStar and Dijkstra.

Limitations

1) we have taken Data variable for cities as Integer so we can't enter nodes of any other data variable.
2) It would have been better if we would have encapsulated the Heuristic function. So that in future, if we want to provide user with different ways to calculate Heuristic function then we can append different cases in that separate piece of code.
3) we could have favoured composition over inheritance then adding necessary and common methods among all the subclasses in the ShortestPathAlgorithm would have sufficed and for adding new methods in the subclasses which are not common among all the subclasses, we could have created new classes and then create their references in the subclass. This would have solved the problem of Encapsulation what varies for the Heuristic function as well.
4) Used priority queue which is not an optimal data structure when it comes to the Time complexity.

CONTRIBUTION

Payal Basrani(2019B5A70809P)- Made the City interface, CityAStar class, AStar class in the code and also recorded the video.

Anushka Jain(2019B5A70809P)- Made the ShortestPathAlgorithm class, CityDijkstra class, Dijkstra class in the code and also wrote the Readme file.