Code evaluation based on the OOP principles and design patterns

OOP Principles

1. Encapsulate what varies
Encapsulation refers to siphoning a unit of data and methods into a separate class or interface. In our code, there are three things that are primarly open for modification: the City class, Edges class and Algorithm class. Hence, their contents have been encapsulated separately. We can use whatever algorithm we wish to measure the shortest distance without (significantly) changing the City or Edge class.

2. Favour composition over Inheritance
Both the A* algorithm and Dijkstra algorithm class require members of both the CIty class and Edges class. Multiple inheritance is not possible in Java. Now, either both City class and Edges class can be made interfaces, but then their access modifiers would be public only and can't be changed. So if we want, say, the City id to not be alterable, we cannot make it so. Hence, we have created objects of City and Edges class, references them in the A* and Dijkstra classes, and accessed their members through the objects.

3. Strive for loose coupling between objects that interact
Neither of the classes heavily depend on each other. Major modifications can be made to either of City or Edges or even the algorithm classes without affecting each other. Why? Because each one of them require only a few functionalities of other classes which, as long as they are untouched, are always satisfied. So altering any of these classes doesn't affect the other classes. This benefits us, because in the future one might need to add more features to City objects (say population), and there is no fear of disturbing the programme as a whole.

4. Depend on abstraction, not on concrete classes
According to the needs of the project, it was not found necessary to use abstraction say for City class. In the future, one might want to have different types of cities (say small city, mega city, coastal city). It would've made sense to made an abstract class named City and create concrete classes of the type of city required. In our case, however, since we only need to find the shortest distance between cities (without care of what city they are), abstraction has not been used.

Design Patterns:

Analysing through the lens of Strategy pattern:
In strategy pattern, if we have to create many types of objects but which are not very different from each other, we have two or three options:

1. we include all functions in each object of that class, and then take some input from the user. Based on the input some amongst all those functions shall be executed and we will have the required object. But in this case, we will have to change the parent class every time we introduce a new type of object, or feature. Plus, there will be too many redundant lines in objects which don't need them.
2. We can create different classes for each type, say one class for large city and one class for small city, and then have a new class extend it to get say small coastal city class. However, the problem with this is the more the number of features that define a city, the more the permutations that will be required to create a new class.
Say, other than size and location, we also have population as a factor.
Then we would have to create 8 different classes:
Small coastal more-populous city
Small coastal less- populous city
Small landlocked more populous city
Small landlocked less populous city
Large coastal more populous city
... and so on.
This can create huge difficulties for the programmer in the future.

3. Hence, it would be most sensible to make an abstract class Population, then create concrete classes extending it like small Population class, large population class, and then referencing objects of these classes in the New City class. We can create whatever type of city with whatever features we want, and it is very easy to scale.

Why is the type of city important?
Because the infrastructure is different, facilities are different, so the cost of approaching a city from the other also varies due to these factors, and the answer output by the A* and Dijkstra algorithm might change.

However, as mentioned above, it was not deemed necessary within the scope of this project to utilise abstraction. It is great practice and we will try to create a scalable project in the future.

By

Sahil Gupta- 2019B3A70154P

Ishvit Bhasin- 2019B5A70226P