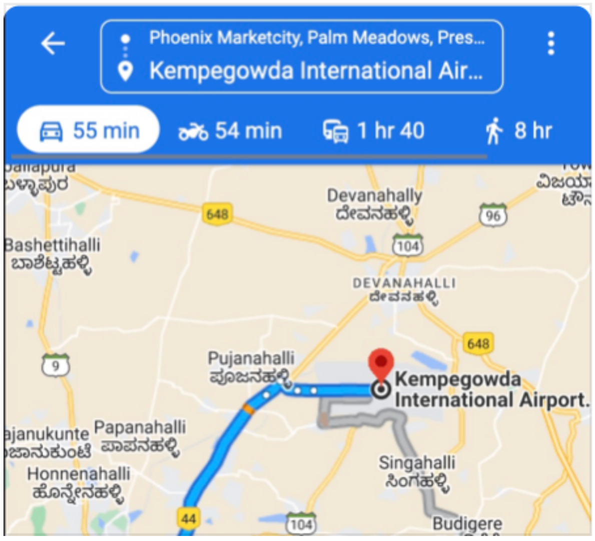
7. Strategy Design Pattern

Behavioural Design Patterns

- Behavioural design patterns are concerned with algorithms and the assignment of responsibilities & communication between objects.

Google maps example

- In google maps, when we search for route from A (source)—> B (destination), it returns the ***shortest path from location A —> B*** and also we will have different transportation modes (by car, by walking, by bike)
- The main operation that google maps does is to find the ***shortest path from location A —> B***. If we choose Bike transportation mode then google maps returns ***shortest path from location A —> B*** for bikes. Similarly we will get different ***shortest path from location A —> B*** for each transportation modes that we choose.
- But no matter what transportation mode we select, the main operation that google maps have to do is to find the ***shortest path from location A —> B***.
- Google maps perform same operation differently for different transportation modes which is to find the ***shortest path from location A —> B***.

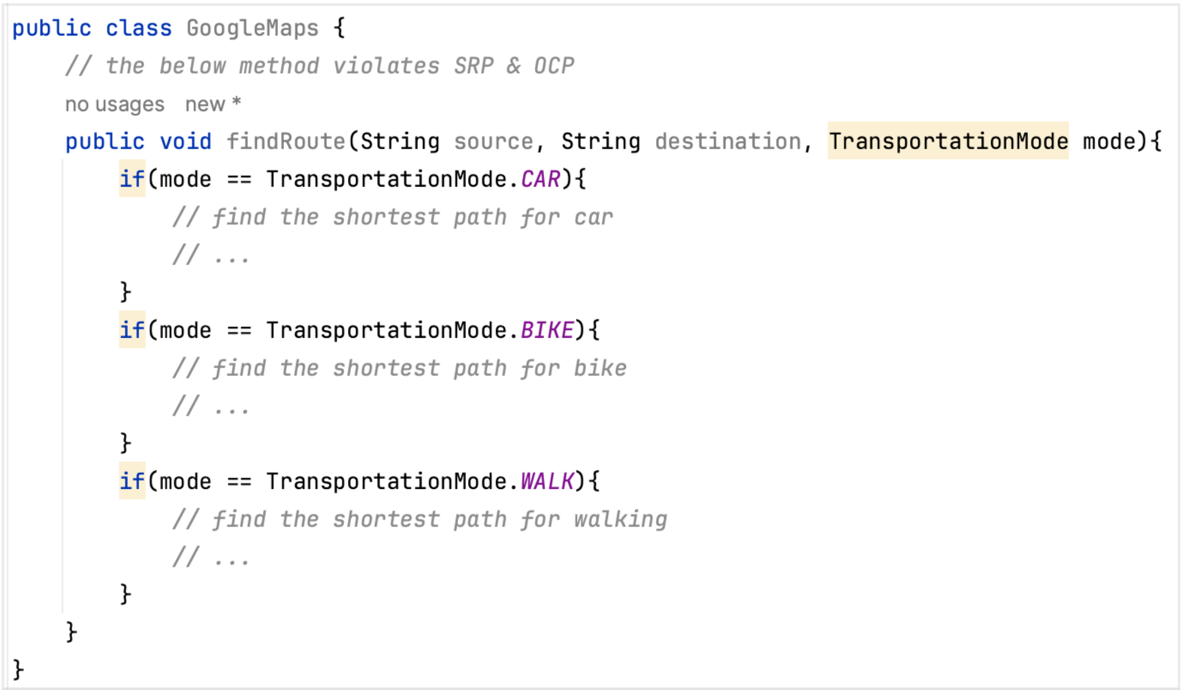


Strategy Design Pattern

- When there are multiple ways to do something, consider using ***strategy design pattern***. As this will make our code much more flexible & loosely coupled.
- Do not implement all the different behaviours inside one method using if else blocks as it will violate both SRP & OCP principles.
  - Implement all the behaviours in separate classes.
- In case of google maps example, do not put all the ways to find the

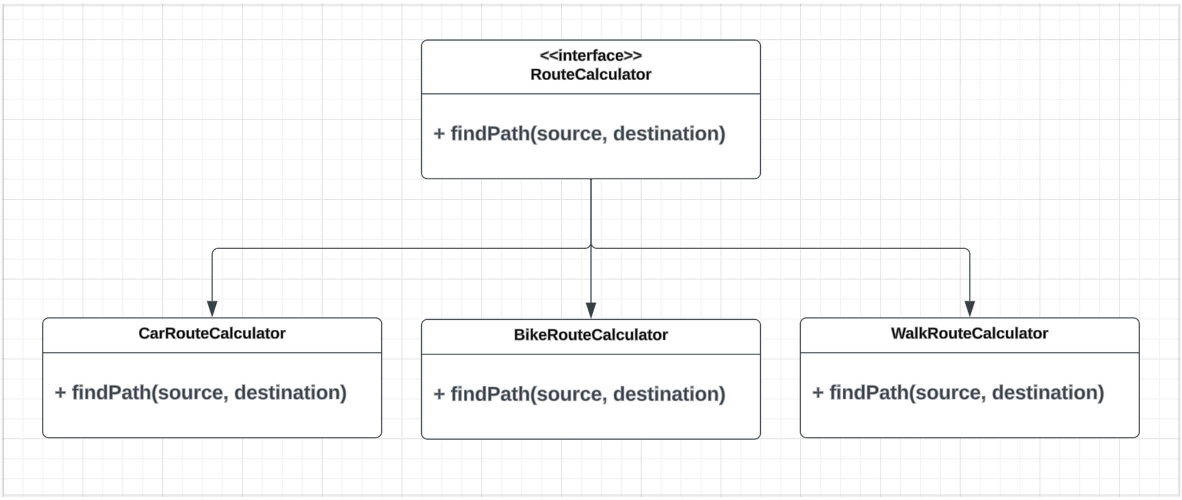
shortest path from location A —> B in one method using if else.
- Create multiple classes for each transportation modes (BikeRouteCalculator, CarRouteCalculator, WalkRouteCalculator)

Avoid this



```
public class GoogleMaps {
    // the below method violates SRP & OCP
    no usages   new *
    public void findRoute(String source, String destination, TransportationMode mode){
        if(mode == TransportationMode.CAR){
            // find the shortest path for car
            // ...
        }
        if(mode == TransportationMode.BIKE){
            // find the shortest path for bike
            // ...
        }
        if(mode == TransportationMode.WALK){
            // find the shortest path for walking
            // ...
        }
    }
}
```

Use this


<<interface>>
RouteCalculator
+ findPath(source, destination)
CarRouteCalculator
+ findPath(source, destination)
BikeRouteCalculator
+ findPath(source, destination)
WalkRouteCalculator
+ findPath(source, destination)

- Since all the car, bike, walk route calculators responsibility is to find the ***shortest path from location A —> B,*** so use an interface to group all the related entities based on the supported behaviour.
- Also the GoogleMaps class will use the Car, Bike & Walk calculators so to avoid using concrete types we need an interface to adhere to Dependency Inversion principle.

Use Factory & Singleton patterns

- In Version 1, we wrote the logics to find routes for all types of Transportation modes in a if else inside a method.
- In Version 2, we introduced the strategy design pattern by creating different classes for finding the shortest path for different transportation modes, but still we have the if else condition to create the different RouteCalculator objects based on the TransportationMode input.
- In Version 3, we removed the logics to create the correct RouteCalculator object based on the Transportation mode argument & introduced a practical factory implementation to create & return the correct RouteCalculator object.
- In Version 4, the issue in the V3 code is, every time a new instance of RouteCalculator is created & returned when the factory method was called, so we used Singleton pattern to avoid creating multiple objects using eager loading approach (as we don't need to pass any arguments to Car, Bike, Walk objects when creating objects)
- Registry pattern can also be used in place of Singleton without doing the cloning part.