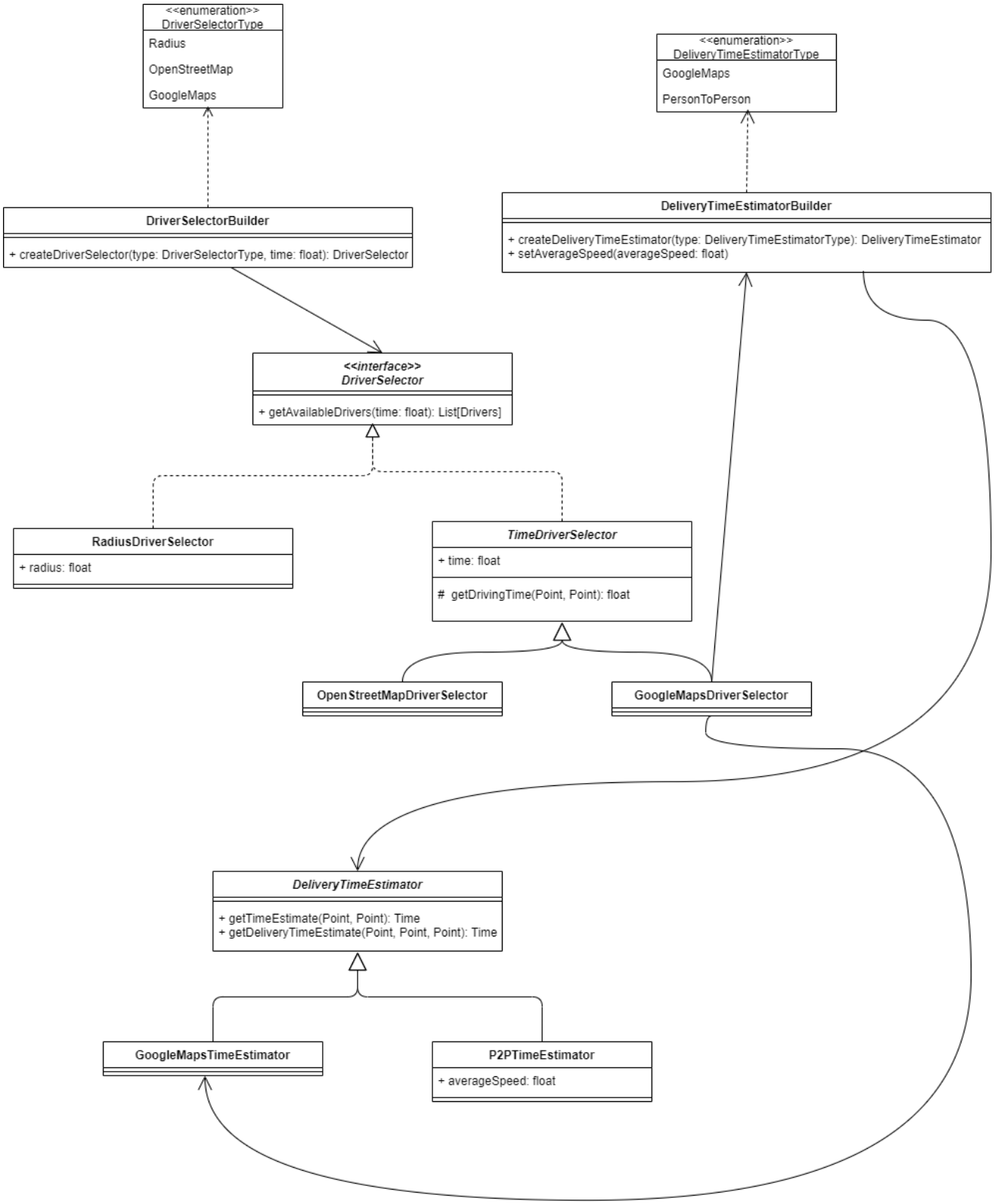
Class Diagram:



Design Choices:

The first design choice made for the above class diagram in the context of agile programming is the use of the single-responsibility principle, each class has only one responsibility. There are two separate builder classes for two different tasks; selecting a driver to deliver an order and estimating the delivery time. These two classes are then inherited by subclasses that implement the task in different ways, such as using a radius or an application programming interface such as OpenStreetMap. Using this principle will make it easier to split up the design implementation amongst the various members, by having each member implement a class individually rather than everyone trying to implement one large class that does everything.

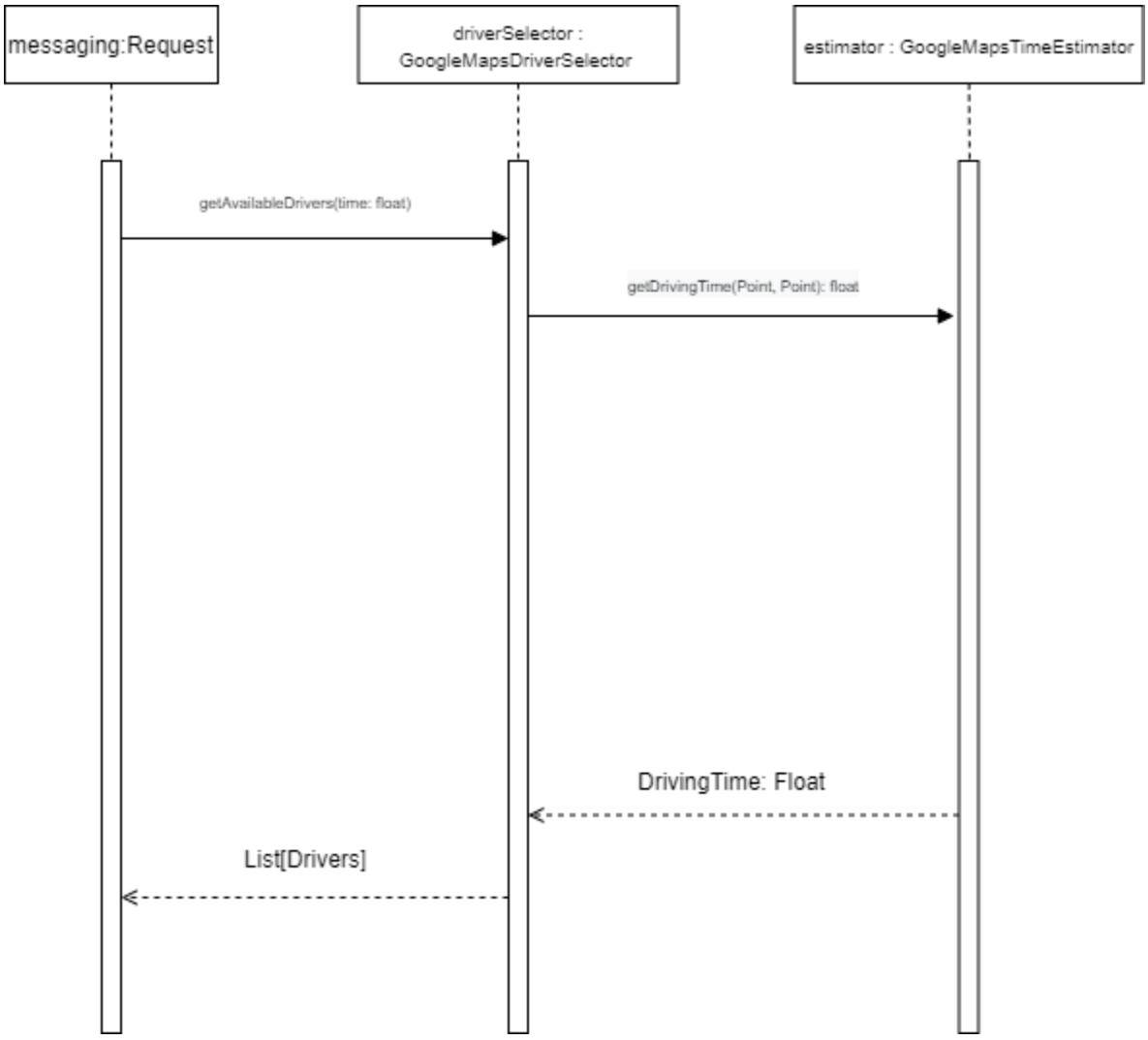
The second design choice made in the design of the above class diagram in the context of agile programming is the usage of the factory method design pattern. It can be seen that there are numerous different classes under the two builder classes that all implement the same methods using different application programming interfaces or radius. The factory method design pattern allows us to create the object we want to use when a request is received for either task. Other parts of the application do not have to worry about which object they want to create when they send the request. Using this principle will make it easier for our classes to be integrated and used with other parts of the application. It also provides a mechanism for error handling; if an application programming interface is unavailable, the next object will be created and used.

The third design choice made in the design of the above class diagram in the context of agile programming is the use of the open-closed principle. It can be seen that the classes that are inherited from the two builder classes are simply extending their respective builder class without modifying the methods in them. Using this principle will make it easier for us to manage the risk of bugs in our code as the design is implemented as we are not modifying the inherited classes. Instead, we are simply extending the inherited classes from the base class, which will make it easier to debug if necessary.

Use Cases:

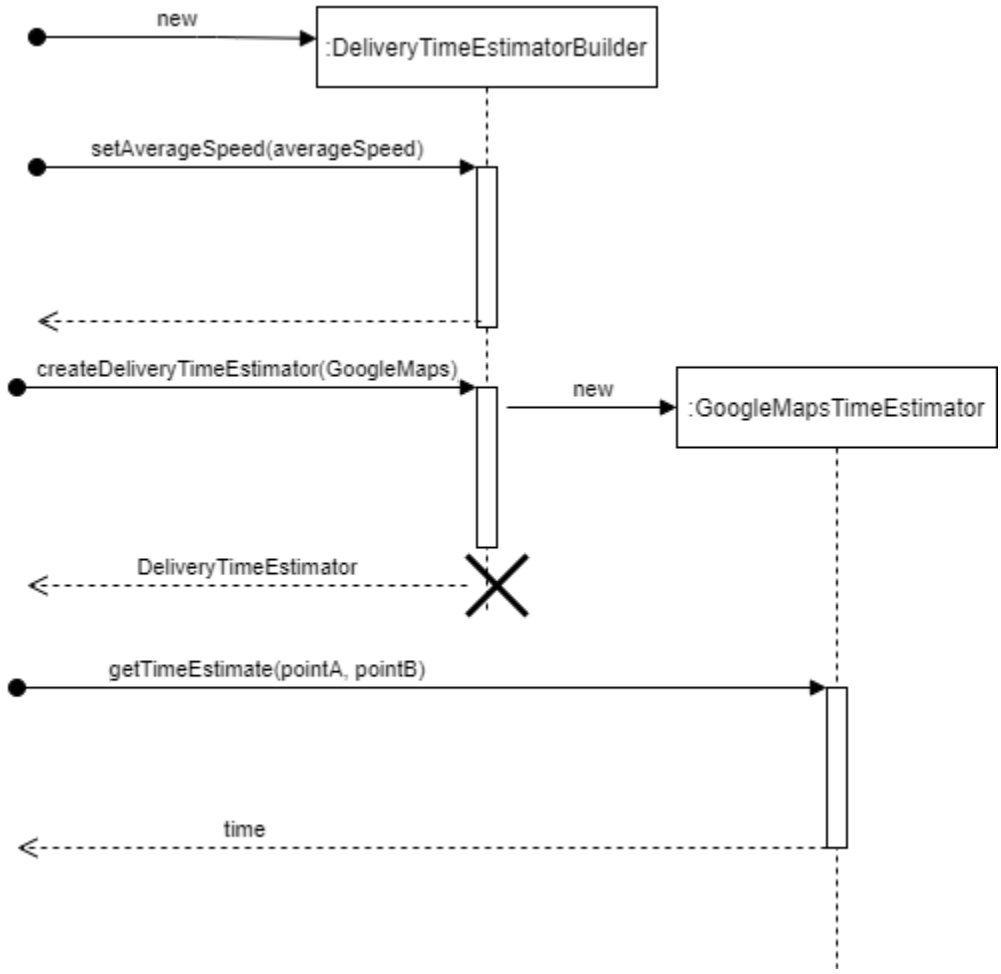
Use Case 0 - getAvailableDrivers()

The first use case is that a request is sent by another portion of the application to get a list of available drivers; this calls the getAvailableDrivers() method, which returns a list of available drivers to the application. This use case is seen in the below sequence diagram.



Use Case 1 - getTimeEstimate()

The second use case is that a request is sent by another portion of the application to get an estimate of the time until the driver arrives at the restaurant for an order. The getTimeEstimate() method is called with the driver's current location and the restaurant's location passed in as parameters that return the estimated time. This use case is seen in the below sequence diagram, showing the creation of the Google Maps based time estimator.



Use Case 2 - getDeliveryTimeEstimate()

The third use case is that a request is sent by another portion of the application to get an estimate on the total delivery time. The getDeliveryTimeEstimate() method is called with the driver's current location, the restaurant's location, and the customer's location passed in as parameters that return the estimated time. This use case is seen in the below sequence diagram.