## Coupling:

**Interaction Coupling:**

law of Demeter:

1. Methods associated with an object can use other methods associated with the same object: none of the methods of our classes call other methods of the same class.
2. Objects can send messages to other objects that are contained in an attribute of them one of their super-classes:
    - Plan has instances of classes City, Travel Service and User (as its requester) so it can communicate with these classes.
    - Rate Plan has an instance of class Plan.
    - Comment has an instance of User (as its author).
3. Objects can send messages to other objects that are as parameters to their methods: In our case, none of the methods of any classes have parameters of other classes.
4. Objects can send messages to other objects that are created by their methods: In our Diagram, none of the methods of any classes create objects of other classes, they merely create objects of the same class or they don't create any object at all.
5. Objects can send messages to objects that are stored in global variables: for the sake of OOP we do not use global variables.

**Inheritance Coupling:**

This type of coupling deals with how tightly coupled the classes are in an inheritance hierarchy.

Using the properties of a superclass is the primary purpose of inheriting from it in the first place. However, from a design perspective, the developer needs to optimize the trade-offs of violating the encapsulation and information-hiding principles and increasing the desirable coupling between subclasses and its super-classes. The best way to solve this conundrum is to ensure that inheritance is used only to support generalization/ specialization (a-kind-of) semantics and the principle of substitutability: In our case, the classes Accommodation and Transportation are sub-classes of the class Travel Service (they inherit its attributes and their retrieve information method calls a method with the same name in this class.) which have generalization relationship with it and can be substituted with their super-class.

To sum up, it seems that our classes are almost loosely coupled.

## Cohesion:

Cohesion refers to how single-minded a module (class, object, or method) is within a system.

**Method cohesion** addresses the cohesion within an individual method (i.e. how single-minded a method is):

All of the methods of our classes are functional i.e. they perform single problem-related tasks, except the request destination method in the class Plan which invokes two other functions of other related classes (retrieve information and search cities) that initialize two of the attributes (travel service and city) of this class therefore it is a classical method.

**Class cohesion** is the level of cohesion among the attributes and methods of a class (i.e. how single-minded a class is).

Glenford Meyers's principles for class cohesion:

1. The class should contain multiple methods that are visible outside the class: most of our classes possess this feature except the classes Admin, Travel Service (and its sub-classes) and Rate plan.
2. Each visible method should perform only a single function: it was discussed in method cohesion.
3. All methods should reference only attributes or other methods defined within the class or one of its super-classes: this feature was discussed in the second law of Demeter and it holds for all of our classes.
4. The class should not have any control couplings between its visible methods: none of the methods of any of our classes control other methods.

All of the types of class cohesion in our classes are one of these two: ideal (the ones which do not have any attributes that are instances of other objects) or Mixed-Role (the ones that contain some instances of other objects which are in the same layer (problem domain layer)).

To conclude, we can claim that our classes are nearly highly cohesive.