Car rental service – an example of Inversion of Control

Lars Bergqvist

Requirements

- Make a service that can handle registrations of rentals and rental returns of vehicles
- After a rental return, the service should be able to calculate the rental cost
- The rental cost has a dedicated formula for each vehicle type
- It should be easy to add new vehicle types to the system
- It should be possible to change the storage type for the rental registrations to any persistence type (any type of database e.g.)

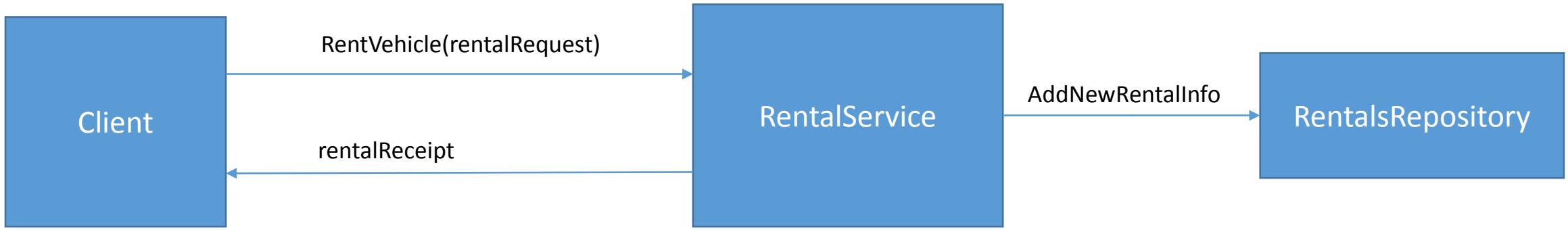
Design goals

- The code should be easily extendable
- The code should be testable
- The code should be easy to understand
- Apply Inversion of Control
  - No dependencies to concrete implementations
  - Dynamic loading of implementations
  - Can add new vehicle types without recompiling the service
  - Can change persistence layer without recompiling the service

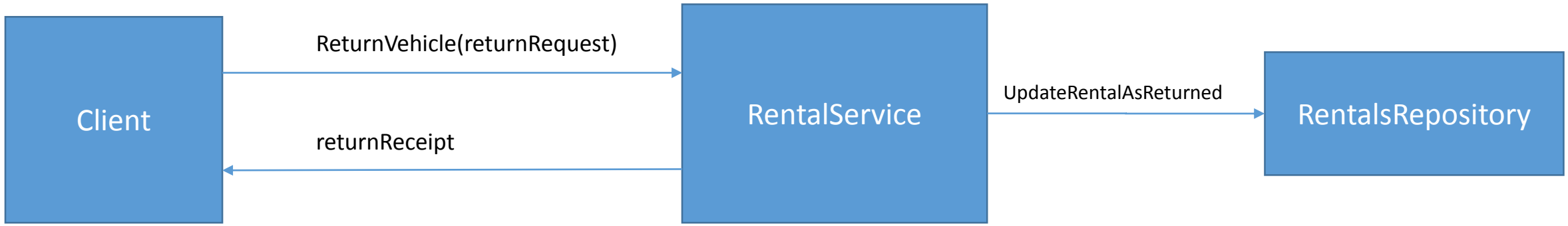
Design

- RentalService.Contract
  - Interfaces and entities for handling rental registrations
  - IRentalService
    - Handles calls for rental registrations and rental returns
    - Returns registered rentals
  - IVehicleTypesRepository
    - Returns all available vehicle types
- RentalsRepository.Contract
  - Interfaces and entities for storing and updating rental registrations
- VehicleTypes.Contract
  - IVehicleType
  - Implemented by the concrete vehicle types
  - Contains a Factory that loads all implementations of IVehicleType from available assemblies in the system dynamically

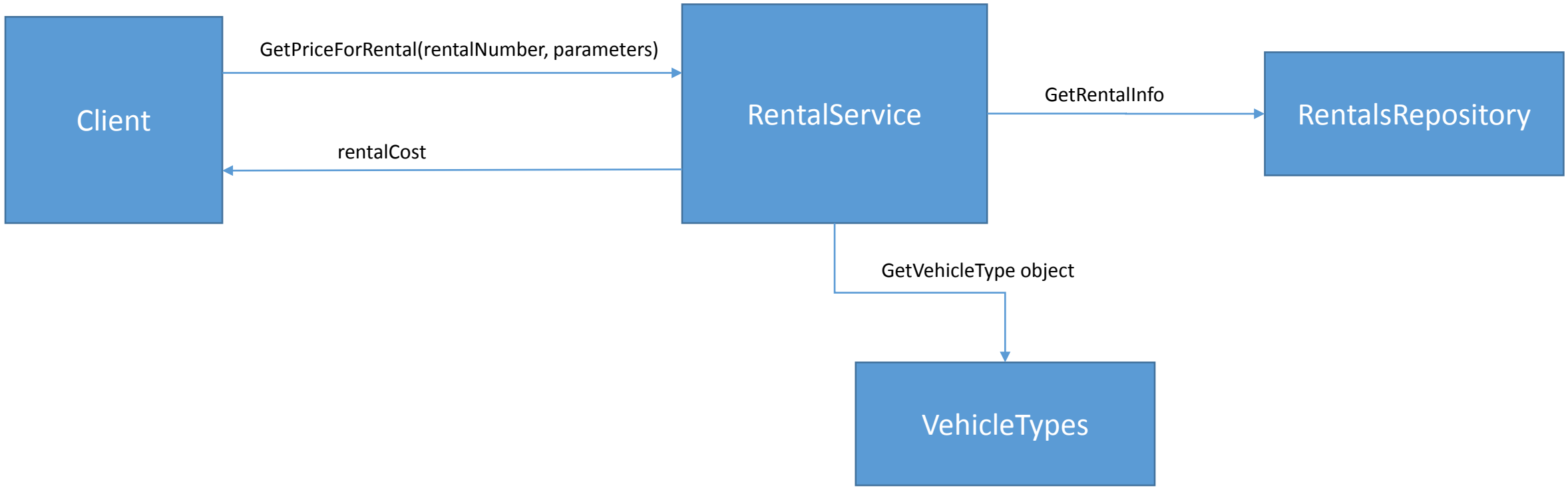
Registering a rental, RentVehicle



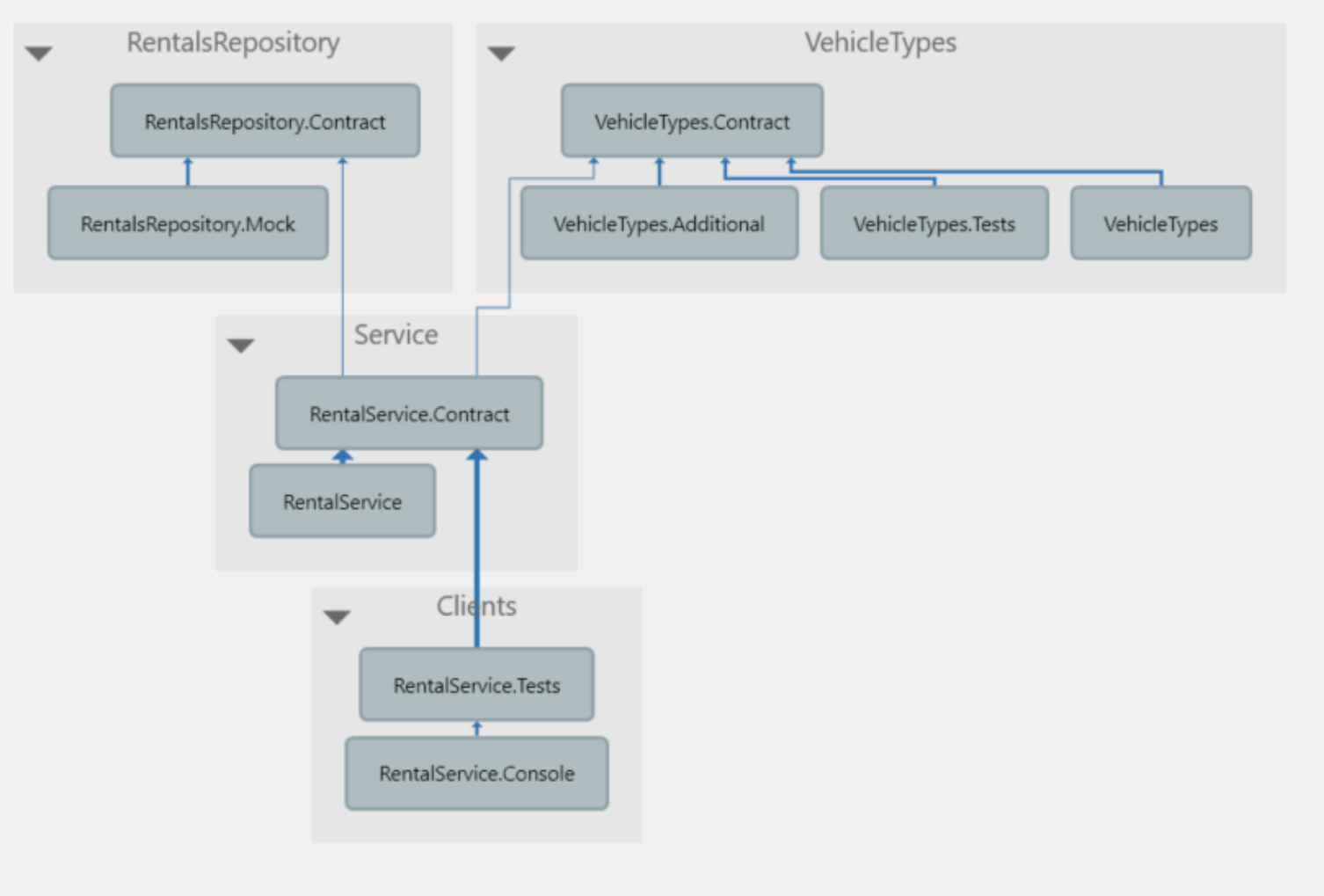
Registering a rental return, ReturnVehicle



Calculate the rental cost, GetPriceForRental



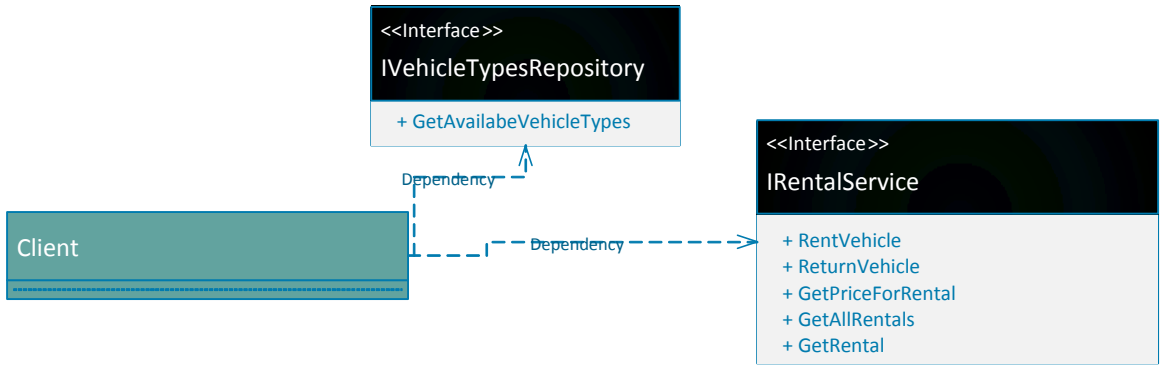
Assemblies and dependencies



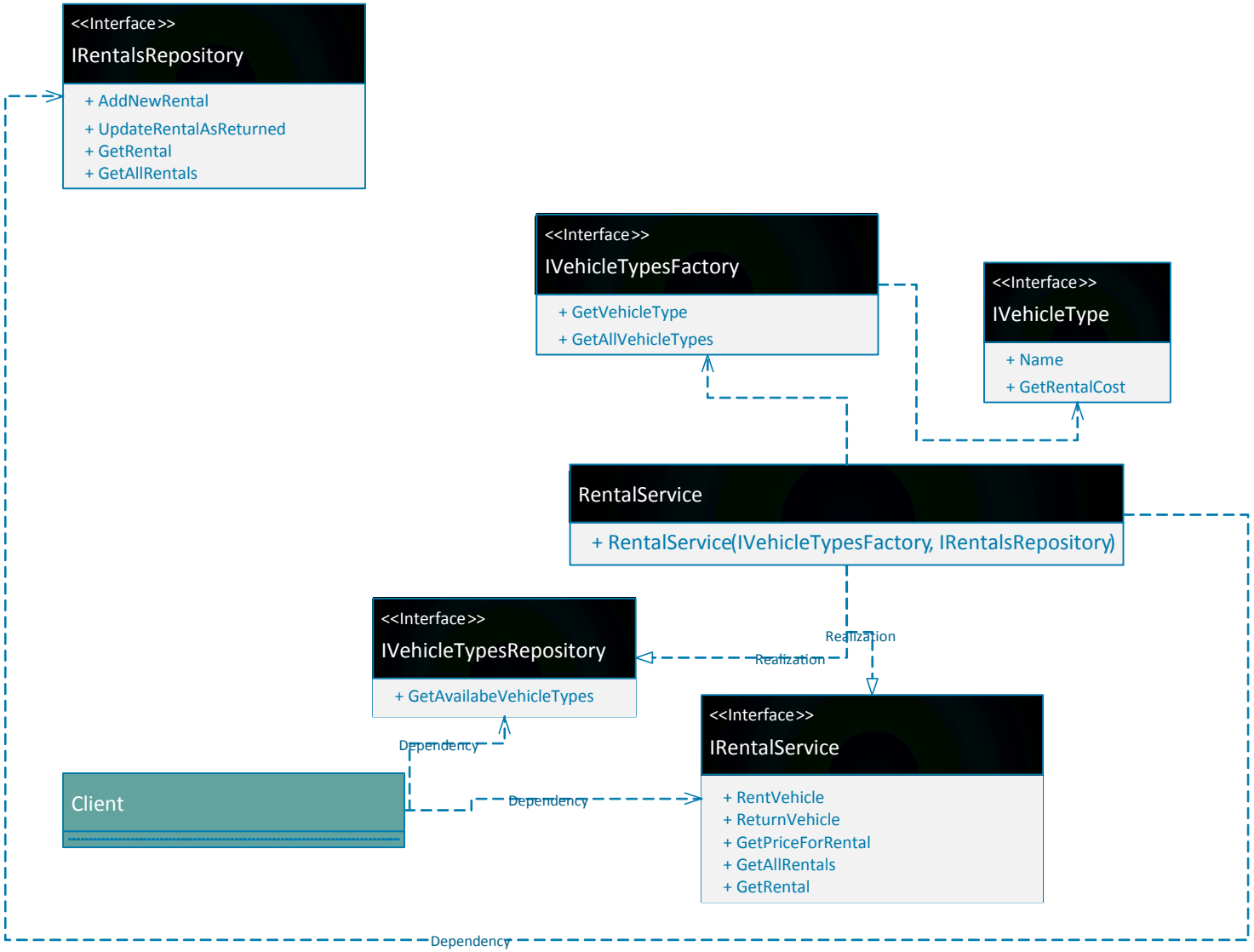
Customers can get new vehicle types and/or new persistence layer without changing the exsiting syststem.

RentalService can be mocked.

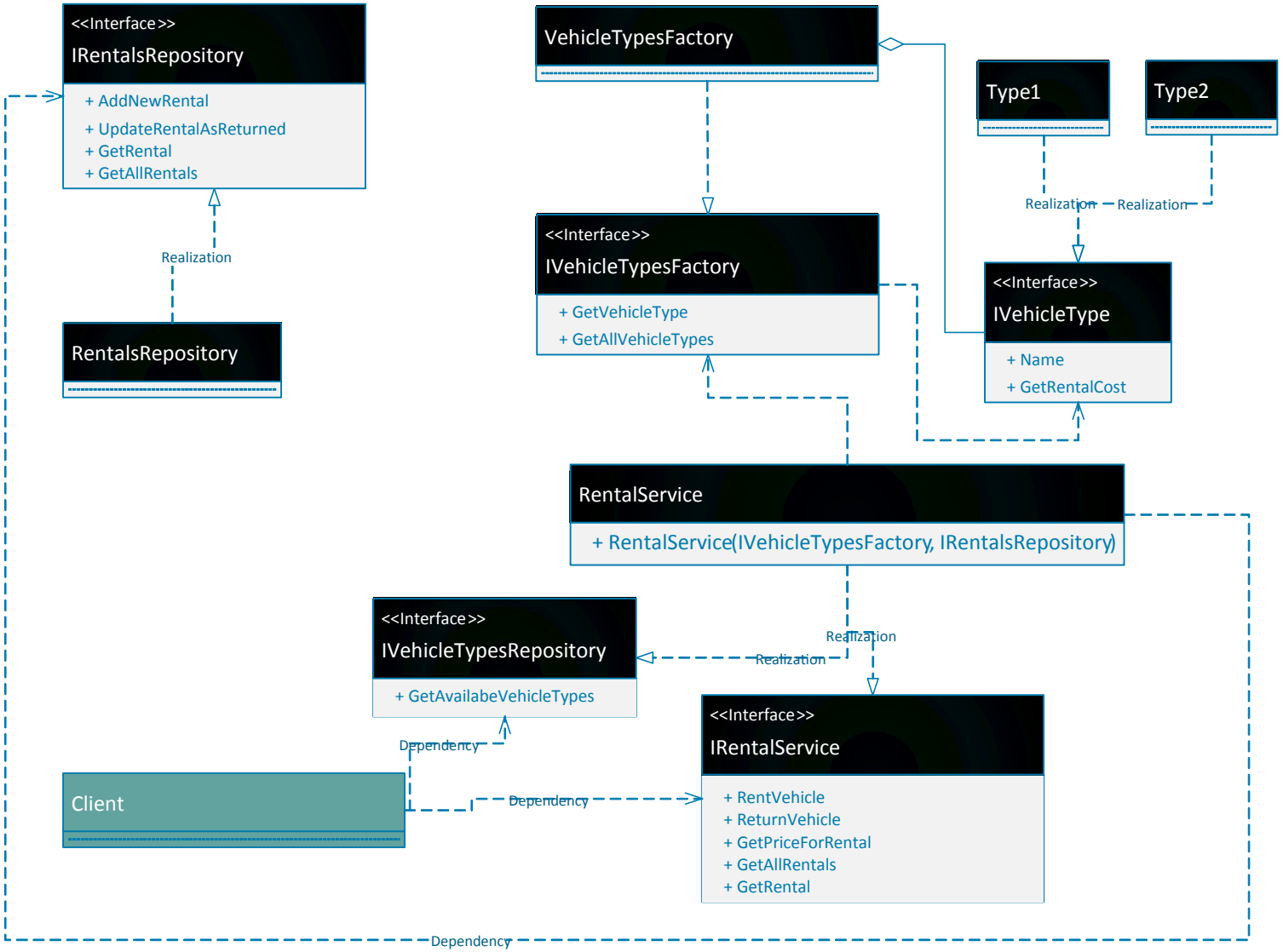
The client code uses two interfaces when working with rentals



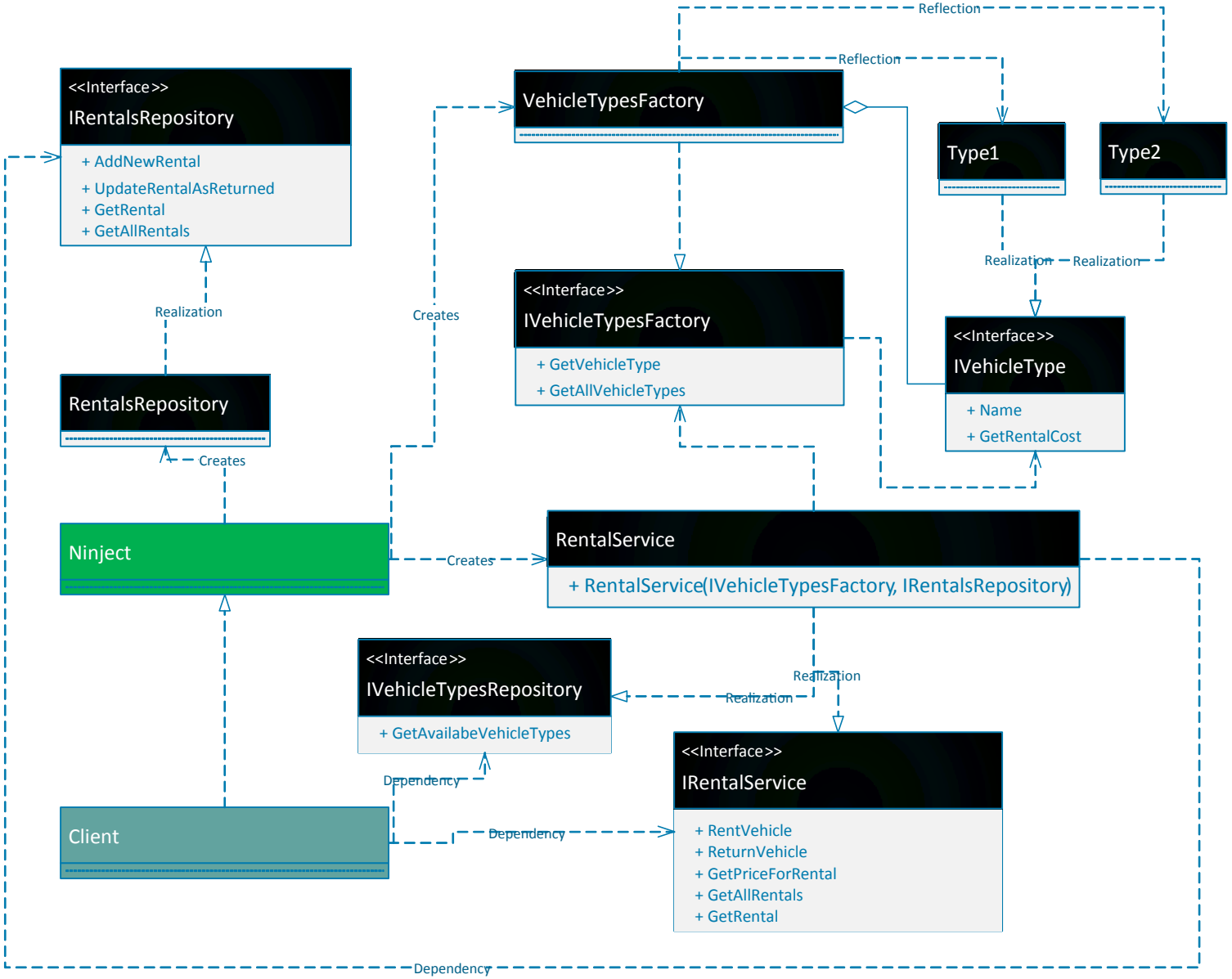
These interfaces are implemented by RentalService that in turn depends on two interfaces



And these interfaces have implementations as described below



The implementations are loaded dynamically with Ninject



To be able to bind several different implementations to IVehicleType in an easy way, a dedicated piece of reflection code is used.

Technologies and frameworks

- Visual Studio 2013
- .NET 4.5, C#
- NUnit for testing
- Ninject for dependency injection

Discussion

- If this was a real project/product, this design would probable be BDUF if implemented in this way from the start
- A better approach would be to start with the core values like programming against interfaces and wait with refactoring into separate assemblies + dynamic assembly loading until the system grows