Java RMI Session 2

Michiel Ariens and Julie Wouters

October 31, 2014

1 Design Overview

The system is divided into three main parts: the users, the central server and the company servers. Users act as clients to both the central server as well as the separate companies specifically depending on the desired operation. This separation allows unnecessary message flows through the central server to be limited reducing the load on that part. The deployment diagram below illustrates the organisation. Each part will be discussed in detail below.

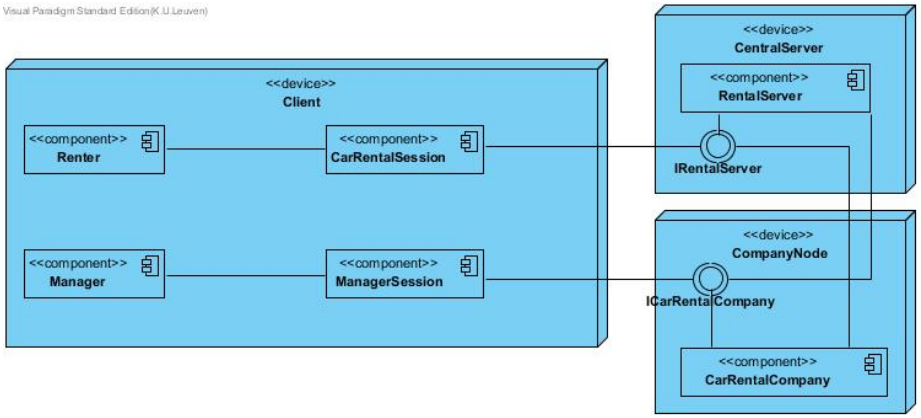

Figure 1: Deployment diagram

1.1 Central Server

The central server is remotely accessible via the RMI registry. The central server fulfils two main roles. It acts as a name server to allow the location of company nodes and it acts as a coordinator for operations which can not be processed by the company nodes alone. Its function as a name server is fulfilled by providing a repository where car rental companies can be registered and retrieved based on a certain key. This functionality is provided through `addCarRentalCompany(crc : CarRentalCompany)` and `getCarRentalCompany(name : String) : CarRentalCompany`. Alternatively we could have used the RMI-registry itself as a repository by using specific naming conventions for the keys. We found our approach to be more defensible from a software design perspective as the code is cleaner and easier to understand.

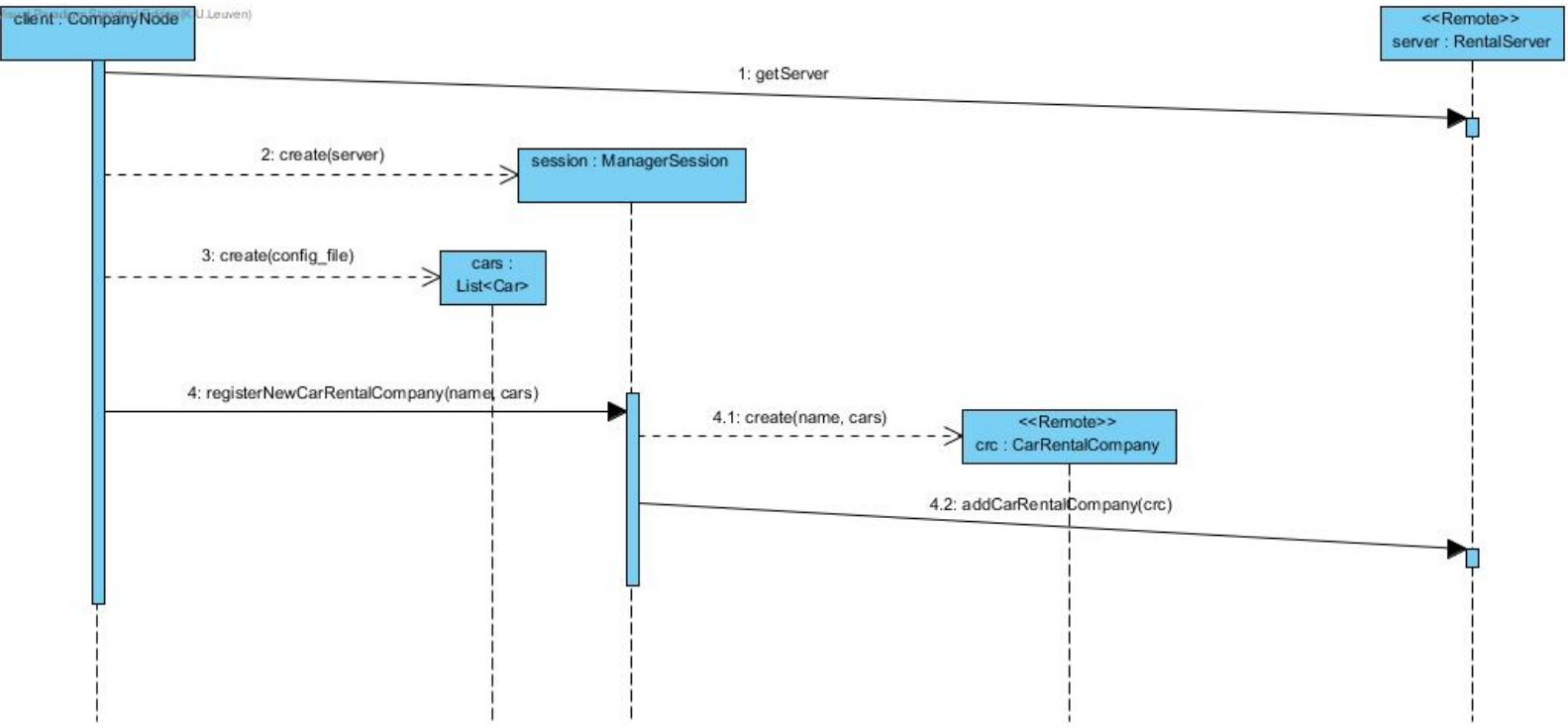

Figure 2: Sequence diagram: Register new company

1.2 Company servers

Each company is able to host its own node. This requires the manager to call `registerNewCarRentalCompany(name : String, cars : List<Car>)` in `ManagerSession`. This will automatically instantiate a `CarRentalCompany` object, export it as a remote object and register it with the central server using the methods in the previous paragraph. A sequence diagram for this functionality is provided:

1.3 Clients and Session Management

We have opted for a relatively heavy client and a relatively lightweight server. Sessions are handled exclusively on the client side. First of all, the client does a lookup in the RMI registry to obtain the RentalServer. The client then creates a session. This can either be a CarRentalSession or a ManagerSession, depending on the type of client. When a session is created, the RentalServer is saved in this session as an attribute. This is necessary because access to specific car rental companies is provided by the RentalServer. The sessions are local, which has several advantages. First of all, there is less load on the server and secondly, communication between client and session is faster and more straightforward. On the other hand, it also has the disadvantage that when the client crashes, the session is lost. The advantages seem to outweigh the disadvantages, however, so we chose to keep the sessions local. Furthermore, the RentalServer doesn't

contain any information specifically relevant to the sessions. The user is the information expert for the session so we keep it close by. The session provides an interface between the user and the system: All communication between user and systems goes through the session.

1.4 Communication

Communication happens through RMI. To use RMI we need to decide which classes are remotely accessible and which ought to be marshalled. The RentalServer and CarRentalCompanies are remotely accessible because they are shared by several clients at once. Classes like ReservationConstraints or Quote, on the other hand, contain static information specific to a single client: The system will not change the state of these objects in a way that needs propagation back to the interested parties. Classes like Car and the Session subtypes are never sent over the wire and need be neither serializable nor remotely accessible. For a full overview of the classes and their stereotypes see figure 4 on page 6.

2 Synchronisation

There were different possibilities for synchronisation between the different clients. We discussed locking, transactions and global synchronisation. Locking, where a client receives a lock for a set of CarRentalCompanies had the disadvantage of possible deadlocking especially when nodes fail. Building a system to counter this comes close to building a full transactional system. While we agreed that a transactional system would provide the best results it falls outside of the scope of this exercise. Instead we provide a globally synchronised method `confirmQuotesForAll(quotes : List<Quote>) : List<Reservation>` in `RentalServer`. This method can not be run concurrently and thus failures are easy to track and remedy. Obviously this is at the cost of performance. Quote confirmations are processed one at a time globally. The scalability of this system system is poor but the only way to remedy this is by implementing a full transactional system.

`confirmQuotesForAll(quotes : List<Quote>) : List<Reservation>` works by organising the quotes per company and sending batches of quotes to the companies for processing using `confirmQuotes(quotes : List<Quote>) : List<Reservation>`. On completion the RentalServer stores the reservations in case they need cancelling. If nothing goes wrong the Rental server returns the reservations and the operation was a success. If something goes wrong the active CarRentalCompany will cancel all reservations in the batch it is processing and then notify the RentalServer of the failure. The RentalServer will then return to each CarRentalCompany that has confirmed quotes for the current query and call `cancelReservations(reservations : List<Reservation>)`. In case of a client failure during this operation the state change is committed on the server and the different companies. This way the client can start a new session, query the system for his reservations and continue as if nothing went wrong. If a company node crashes all other nodes will revert and so no reservations will be committed. If the central server crashes the state is ambiguous and may not be recoverable.

3 Conclusion and Personal Remarks

Given the size of the project we believe we've built a workable solution to the problem but we also realise its shortcomings. If we had more time we might have gone for a different approach. These sessions help to underline the importance of middleware.

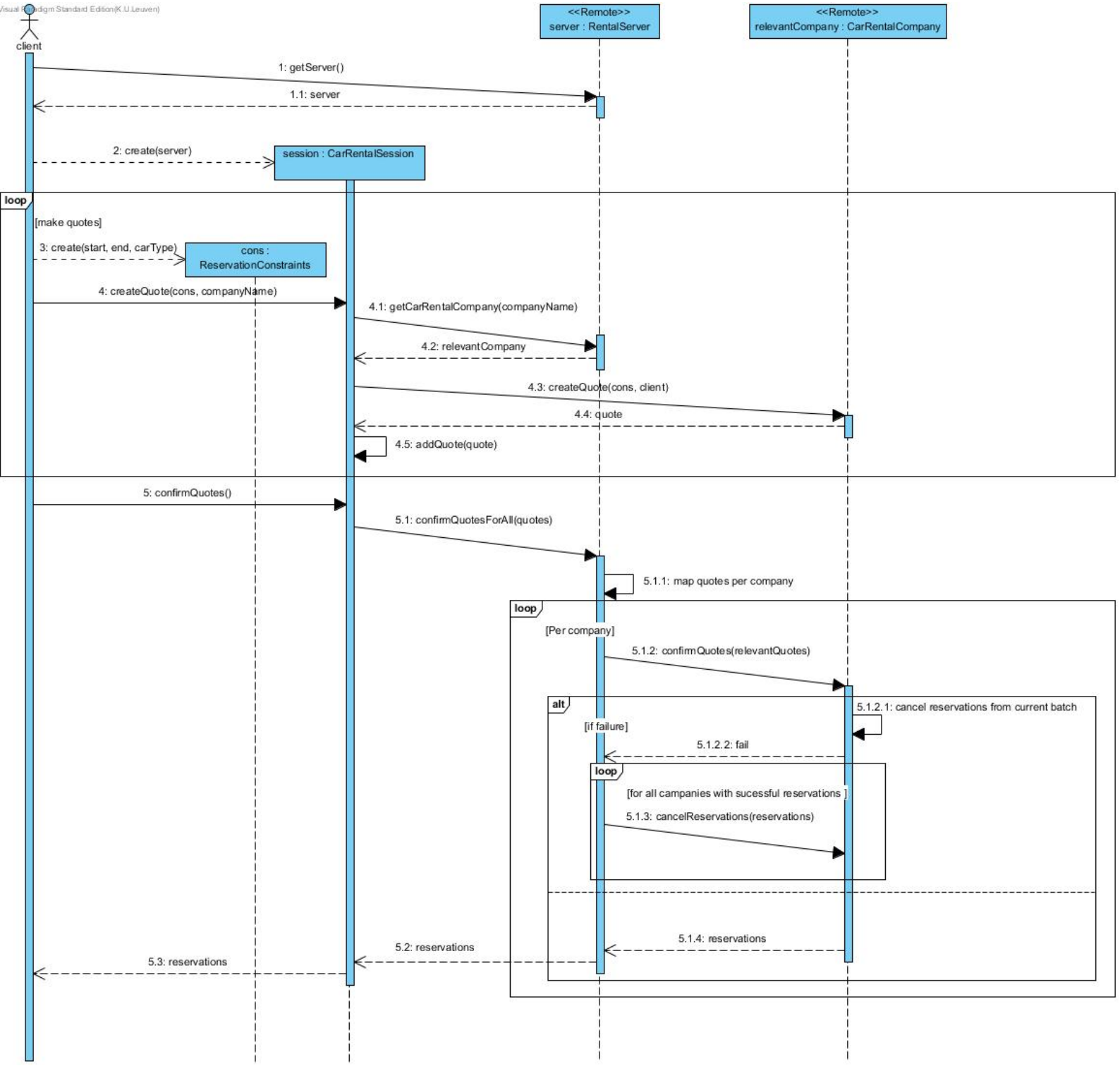

Figure 3: Sequence diagram: typical car rental session

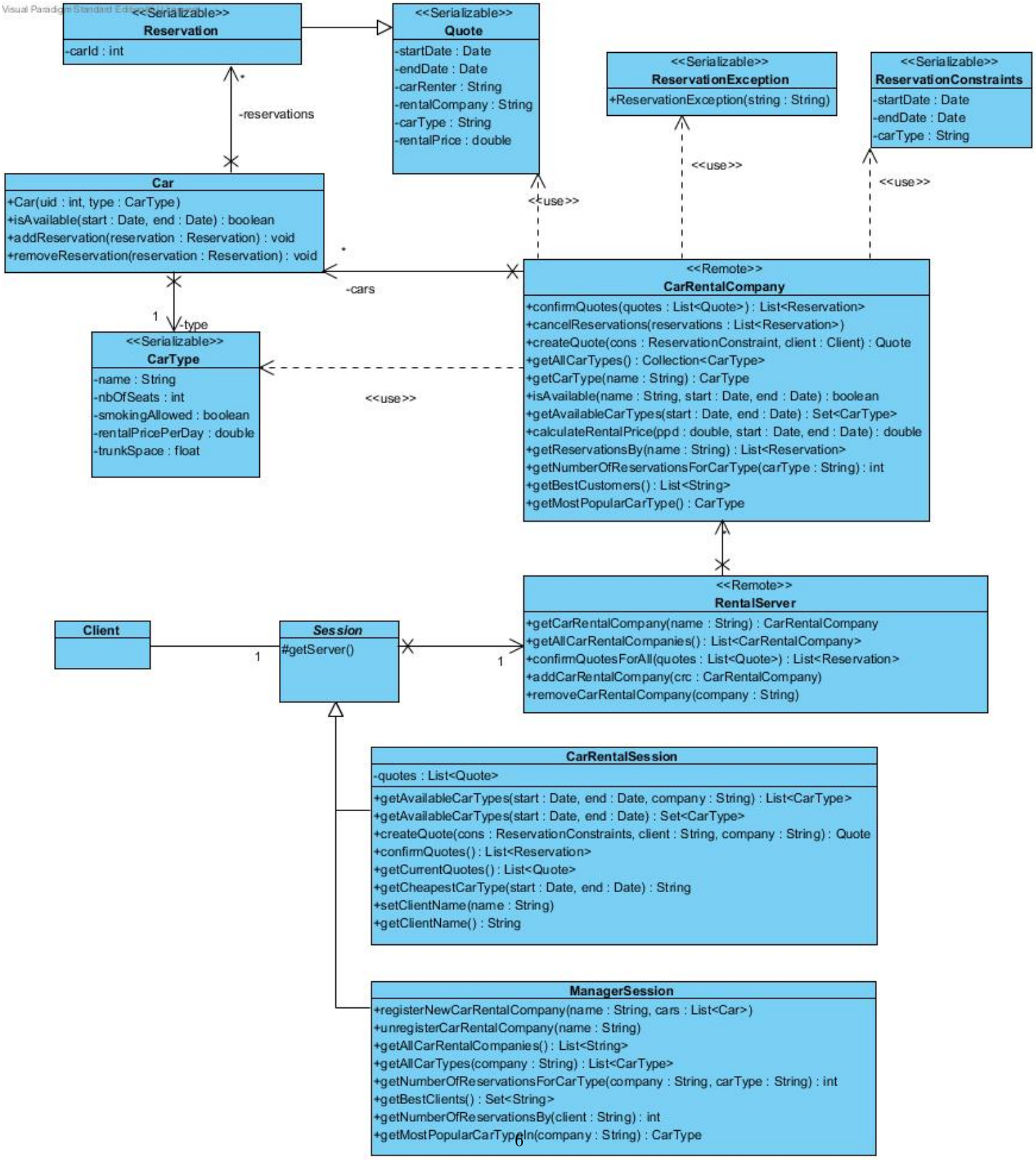

Figure 4: Class diagram