

Distributed systems

Java RMI 2-3 Exercise Session Report

Academic year 2014-2015 Bavo Goosens & Michiel Vandendriessche

**Real world deployment**

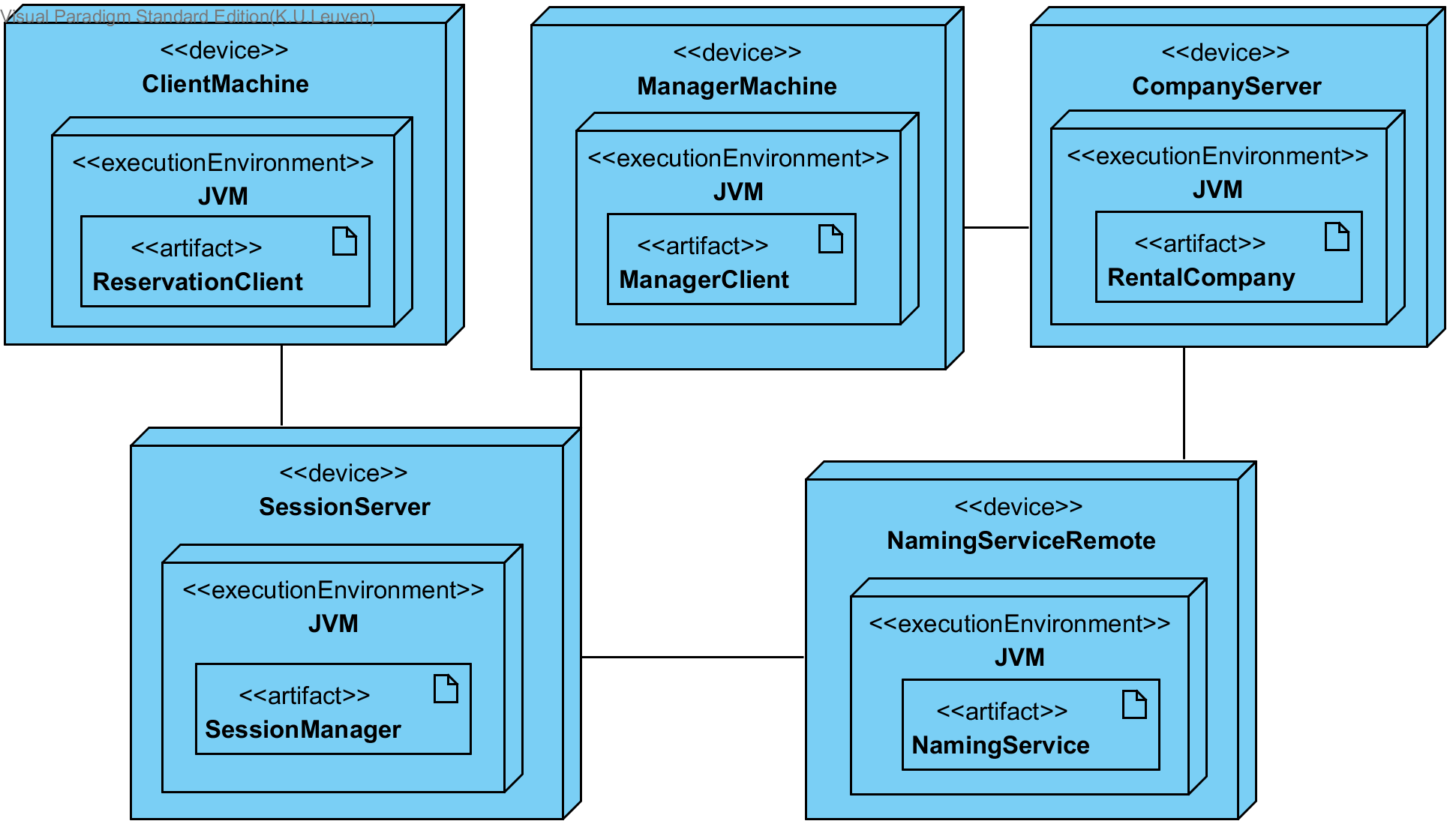


Figure 1: Deployment diagram

Figure 1 shows how our application would be deployed in a real world situation. First of all we have two types of client applications. One that can be used by the manager to start up a new rental company or remove a registered company. Furthermore it offers several useful methods a manager would use in a day to day working environment. The ManagerClient artifact runs in a different JVM (Java Virtual Machine) than the actual rental company server. Initially this will not be the case. When the manager wants to start and register a new rental company on his local server this will be done via the same JVM, but as soon as the manager exits the client the rental company will keep running on that JVM. The client can then be used to access other information via a different JVM for example via the personal computer of the manager instead of a server. The second type of client is a reservation client that offers useful methods to clients who want to order cars or want to check up on the progress of their reservations.

A different server will run a SessionManager artifact. This session manager offers two types of session. A stateful one for the ReservationClient and a stateless one for the ManagerClient. This could be a place where multithreading could be used to increase the performance of the overall system. Since the clients need to know where the SessionManager is running, this information will need to be provided to the client artefacts. All the interaction a client of the system needs will be offered through these sessions. This makes the application transparent to the users and gives us more freedom in how we internally keep track of all the companies.

The last important artefact is the NamingService. It is used to register and unregister the rental companies. It will however never be accessible to a RentalCompany. The SessionManager knows where to find and use the NamingService artefact. This artefact will run on its own server and JVM on that device.

**Remotely accessible classes:**

**NamingServiceRemote:**

Through this interface sessions can register new rental companies, unregister registered rental companies, acquire a rental company by name or acquire all registered rental companies from the naming service, which runs on its own dedicated server. This interface is remote because it is used by all sessions. The sessions run on a different server to allow for more flexibility in the deployment of the system. Letting the session manager keep track of the registered companies would also be a mixing of responsibilities.

**ManagerSessionRemote:**

This remote interface is used by the manager clients. Manager clients can invoke methods on their manager sessions to register new rental companies or get important information about their company. Since Manager clients can run on both servers and personal computers, this interface needs to be remotely accessible to those JVM’s.

**ReservationSessionRemote:**

Reservation clients use this interface to execute all their operations. It needs to be remote to be accessible from different JVM’s running on the client machines. Useful methods include but are not limited to: creating quotes, getting the cheapest CarType, confirming the requested quotes.

**SessionManagerRemote:**

This remote interface is used by all clients. Through this interface clients can acquire new ManagerSessionRemotes or new ReservationSessionRemotes. It also handles the destruction of these sessions by removing the reference to the session and letting the java garbage collector do the hard work.

**ICarRentalCompany:**

This interface is remote because it needs to be accessible from a lot of different places. ManagerClients are used to create new rental companies. The proxies to these companies get stored in the NamingService. When a session needs a rental company it will look the company up in the naming service. So sessions also need remote access. This interface offers sessions methods for manipulating registered companies.

**The serializable objects:**

All the exceptions are serializable since the extend Exception which is always serializable. This makes sense because exceptions need to be thrown over the wire to the client of the service which caused the exception. A Quote needs to be returned to a session which requested the quote from a rental company. It thus needs to be serializable. The same reasoning can be applied to our choice to make CarType, Reservation, Car, Tuple and ReservationConstraint serializable.

**Remote objects located at the same host:**

Only ManagerSession, ReservationSession and their SessionManager run on the same host. All the other remote objects (can) run on separate hosts. The session manager creates and manages the specific session and distributing them would make this less performant. The reason why we chose for this design is with multithreading in the back of our heads. We could run a thread pool with session threads on this server. It would work in the same way that a webserver processes HTTP(Hyper Text Transfer Protocol) requests. For the other remote objects we went with this solution for increased flexibility and distribution. Rental companies can run on local servers and the Naming service runs on its own centralized location. If the naming service became a bottleneck we could opt to run this service on the same host but on a different JVM. This would greatly lower up the request time.

**The following remote objects are registered via the built-in RMI registry:**

**NamingService:**

The naming service is registered via the built-in RMI registry. That way the host that runs all the sessions can acquire a proxy object for the naming service so that the sessions can invoke methods on it (via the NamingServiceRemote interface). In a real world situation this service could run on its own machine via the web or on a different server in the building. A user of the service would need to know where to look for the naming service. A well-known host and port for a RMI registry offers the required solution.

**SessionManager:**

The session manager is registered via the built-in RMI registry. That way the host that runs the client code can acquire a proxy object for the session manager so that the client can invoke methods on it and acquire sessions from it (via the SessionManagerRemote interface). The afore mentioned reasoning is used to make this decision.

CarRentalCompanies and Sessions are respectively stored and created on our own management implementations (SessionManager and NamingService) . They thus need not be registered to a RMI registry instance. This also keeps our code cleaner and the user does not need to remember to start his RMI registry.

**Life cycle management of sessions:**

Sessions are created when a client invokes the getManagerSessionRemote or getReservationSessionRemote methods on the session manager. Sessions only live on the host where they are created (there is a separate host which runs all sessions). Only proxy objects for sessions are sent over the wire and returned to the client.

Sessions are closed when the client invokes the closeManagerSession or closeReservationSession method on the session manager. When these methods are invoked, the session manager removes all references to that session.

We assume the client closes the session itself. Although, we can easily implement an automatic session closing system which uses a time out.

**Synchronization:**

Synchronization is necessary at the confirmQuote method in a CarRentalCompany. After all, multiple clients can all have their own individual session at the same time. That session can invoke methods on the same CarRentalCompany-object. When the company tries to confirm a quote, it checks whether it is able to do that. When multiple clients invoke this method at the same time, the integrity of the system is compromised (imagine two clients making the same reservation at the same time). Because of that reason, the method needs to be synchronized.

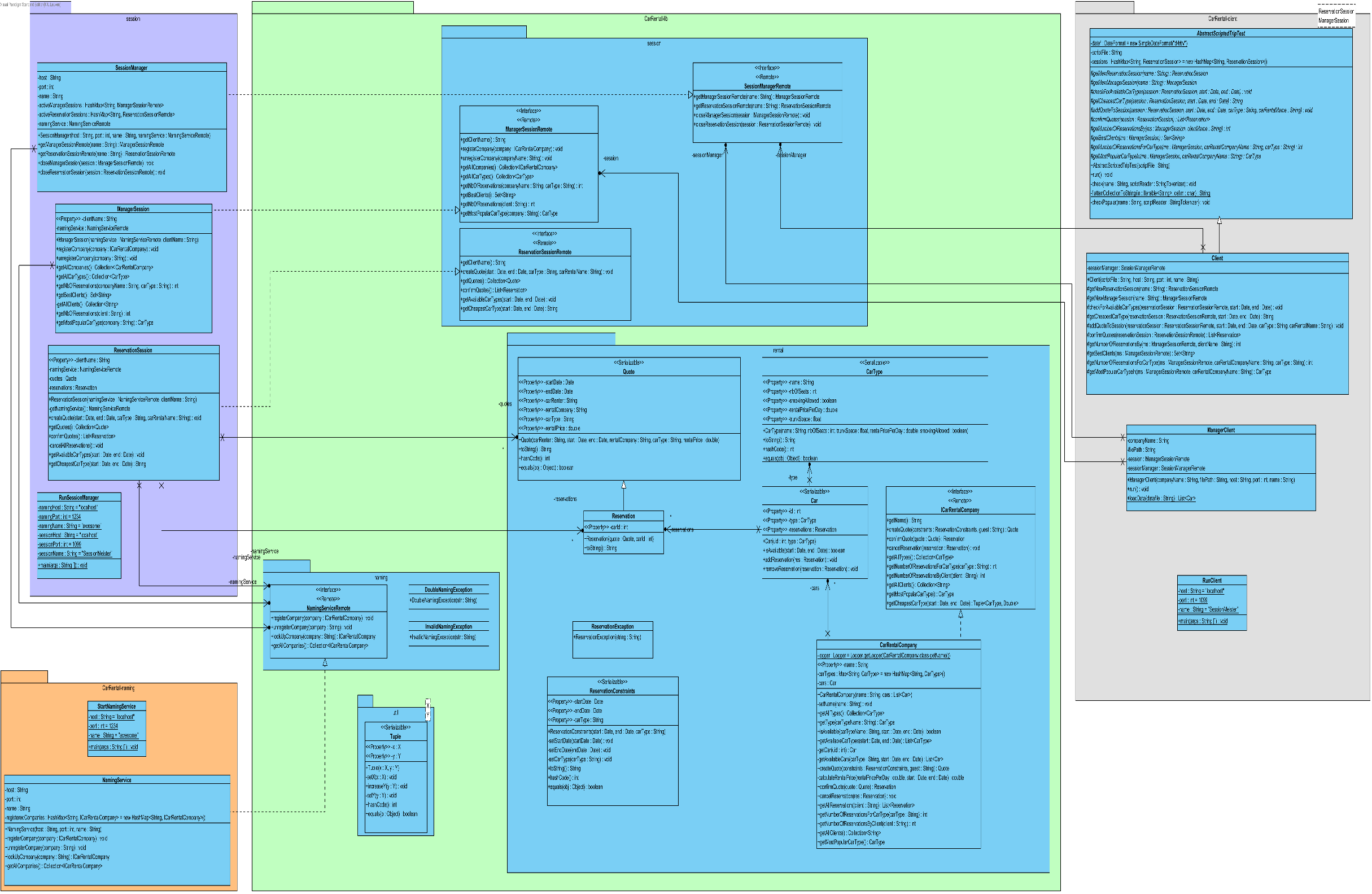


Figure : Full Class Diagram

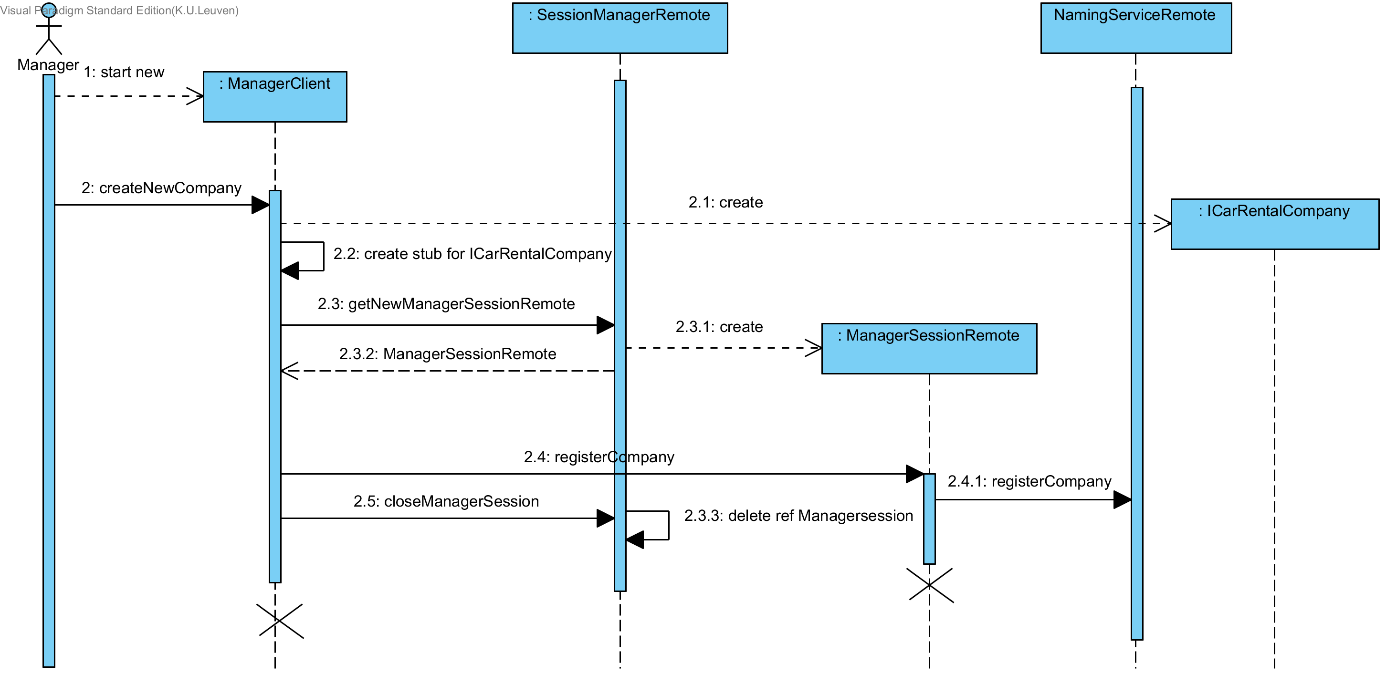
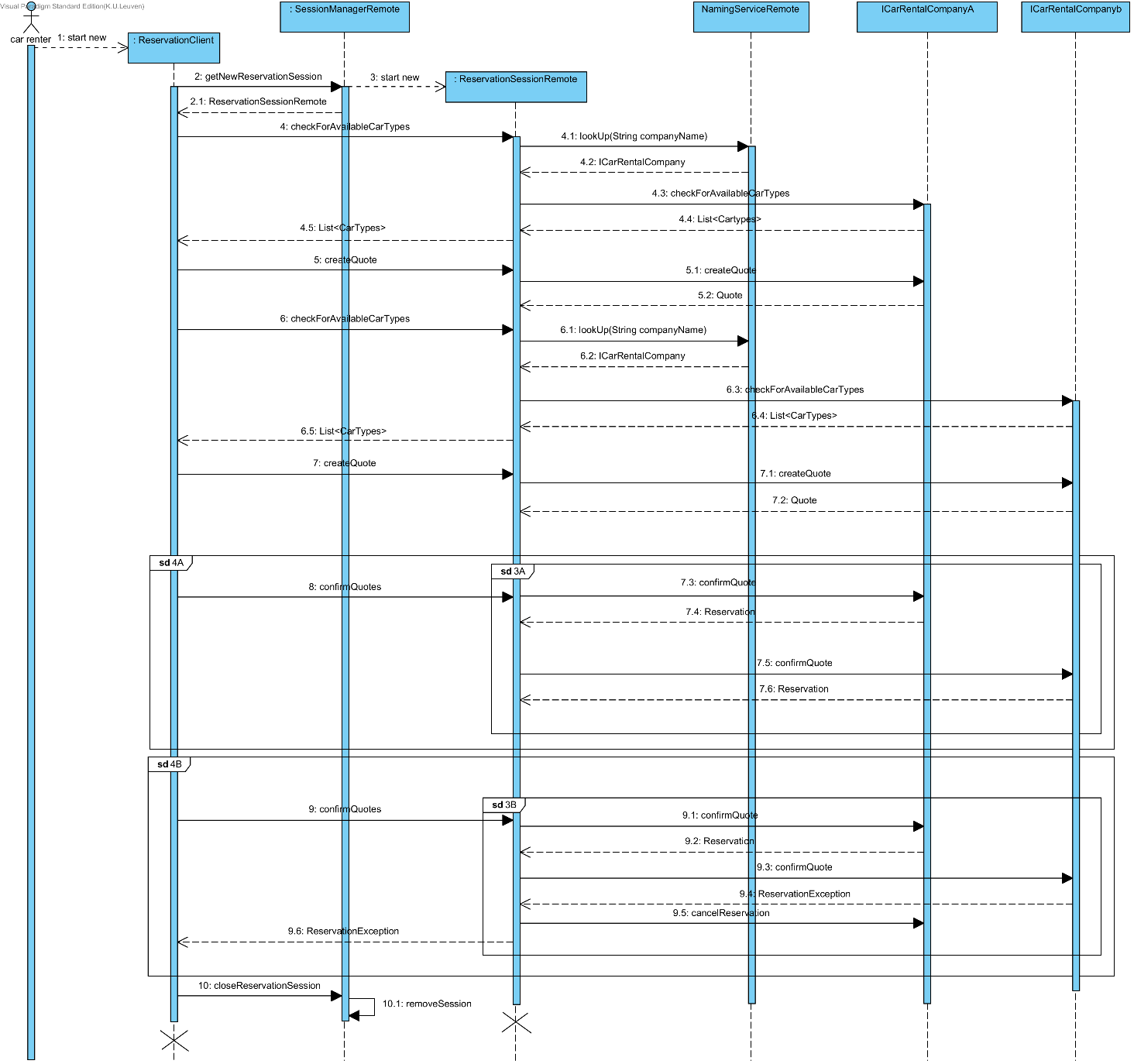


Figure 3: System Sequence Diagram for example story

Figure 4: Creation of a new CarRentalCompany