RMI Session 2 - design decisions & diagrams

Design Overview

The **SessionManager** is responsible for life cycle management and does this by keeping track of **Session** objects (base class of sessions like **ManagerSession** and **ReservationSession). NamingService** is used to find a **CarRentalCompany** by name (this class needs to register itself first to be able to do so). Finally, there is a client which communicate with the servers. The class **Main** represents this client.

Note that in the Class Diagram and Deployment Diagram SessionManager, NamingService and CarRentalCompany are shown as being seperate entities. In the code however the Naming Service Creates a SessionManager and CarRentalCompany objects.

Design Decisions

Which classes are remotely accessible and why? Which classes are serializable and why?

Remote:

- SessionManager: It is used to manage different sessions and thus needs to be able to get called by clients that want a session
- NameService: This class provides an interface to call distributed classes by name and thus needs to be able to get called by classes that want to register their interface with some name
- CarRentalCompany: This class acts as a server in which information can be stored and updated

Serializable:

- Quote, Reservation, CarType, ReservationConstraints, Car: These classes are returned by or sent to remote methods and thus require that they are serializable

Which remote objects are located at the same host (or not) and why?

SessionManager:

- Session, ManagerSession, ReservationSession: These sessions are all managed by the SessionManager since it needs to create/delete them
- RentalStore: This is a helper-class that the Session classes use to contact the needed CarRentalCompany more easily
- SessionManager

NamingService:

- NamingService

CarRentalCompany:

- Has none beside the usual classes (Car, CarType, etc.).
- CarRentalCompany

Which remote objects are registered via the built-in RMI registry (or not) and why?

All remote objects are registered. If they are not then an address change would result in the class that would call it not finding the object. By registering the object, the address can be changed without the calling address not finding it, since it would just needs to do another lookup.

Briefly explain the approach you applied to achieve life cycle management of sessions.

Sessions can be created and removed by the client invoking the SessionManager. If a client has not responded for a specific amount of time, the SessionManager will remove the session itself.

At which places is synchronization necessary to achieve thread-safety?

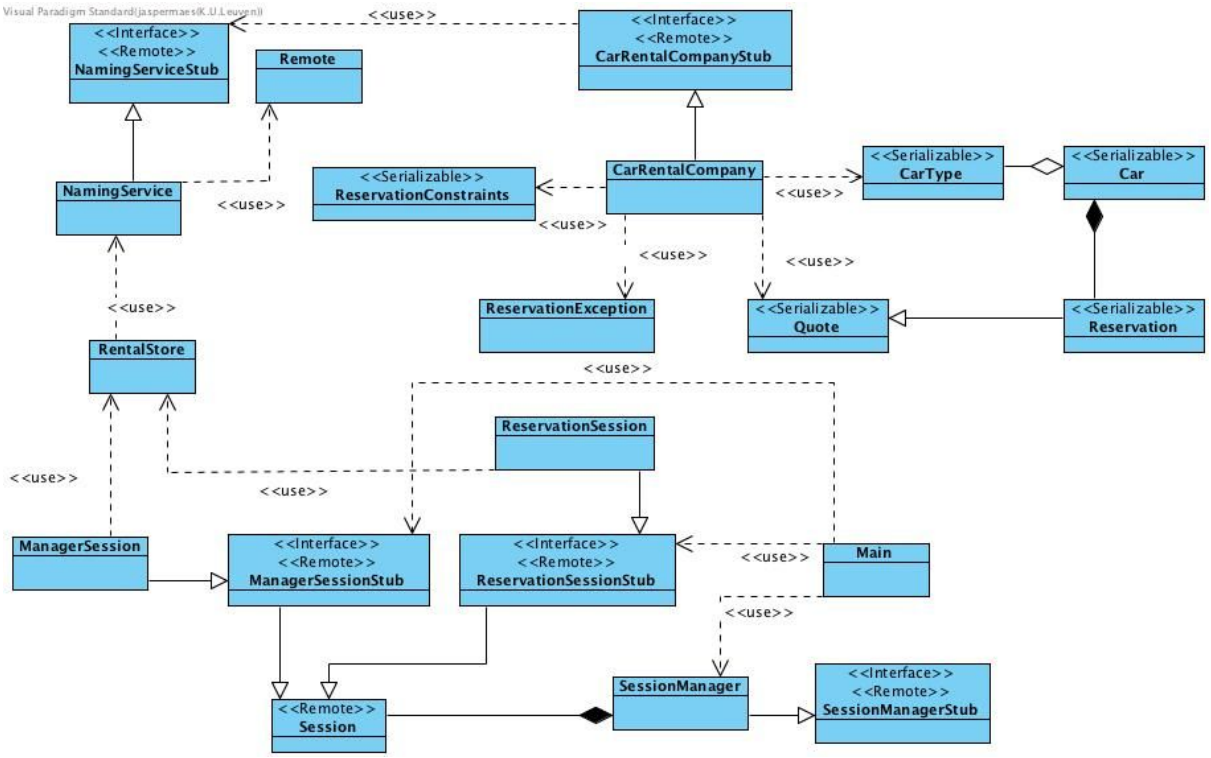
- At the SessionManager since the addition of multiple sessions can result in race conditions.
    - Synchronised methods: getSession(), newManagerSession(), newReservationSession(), removeSession()
- At the CarRentalCompany since confirming multiple quotes at the same time can result in race conditions.
    - Synchronised methods: cancelReservation(), getAvailableCarTypes(), createQuote(), confirmQuote(), cancelReservation()
- At the NameService since the registration of multiple new names can result in a race condition.
    - Synchronised methods: register(), unregister() and getAll() methods.

Will those places become a bottleneck by applying synchronization?

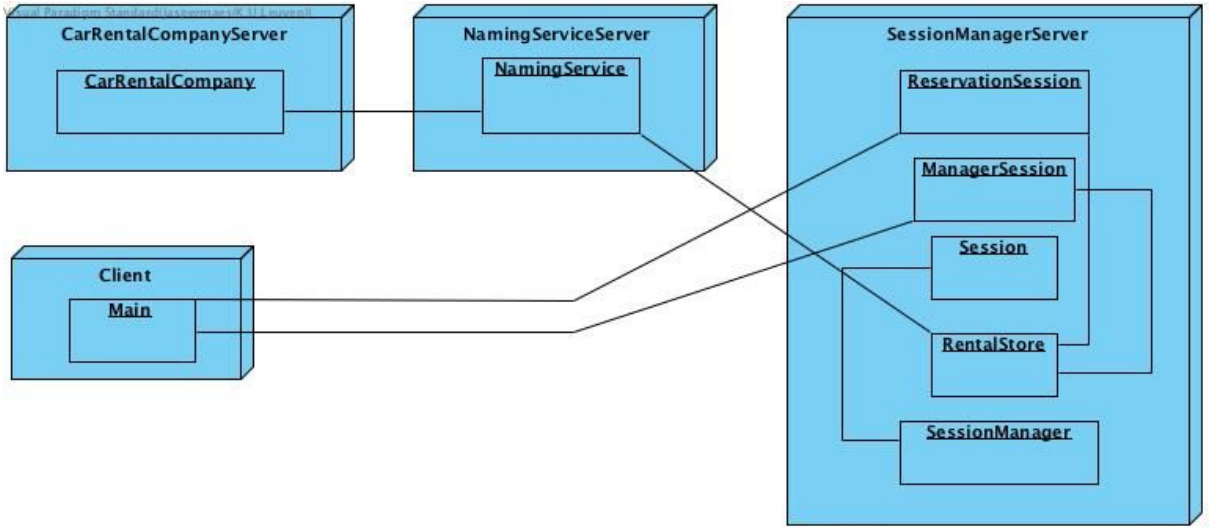
Yes, you will always lose time by waiting, acquiring locks...

Diagrams

Class diagram

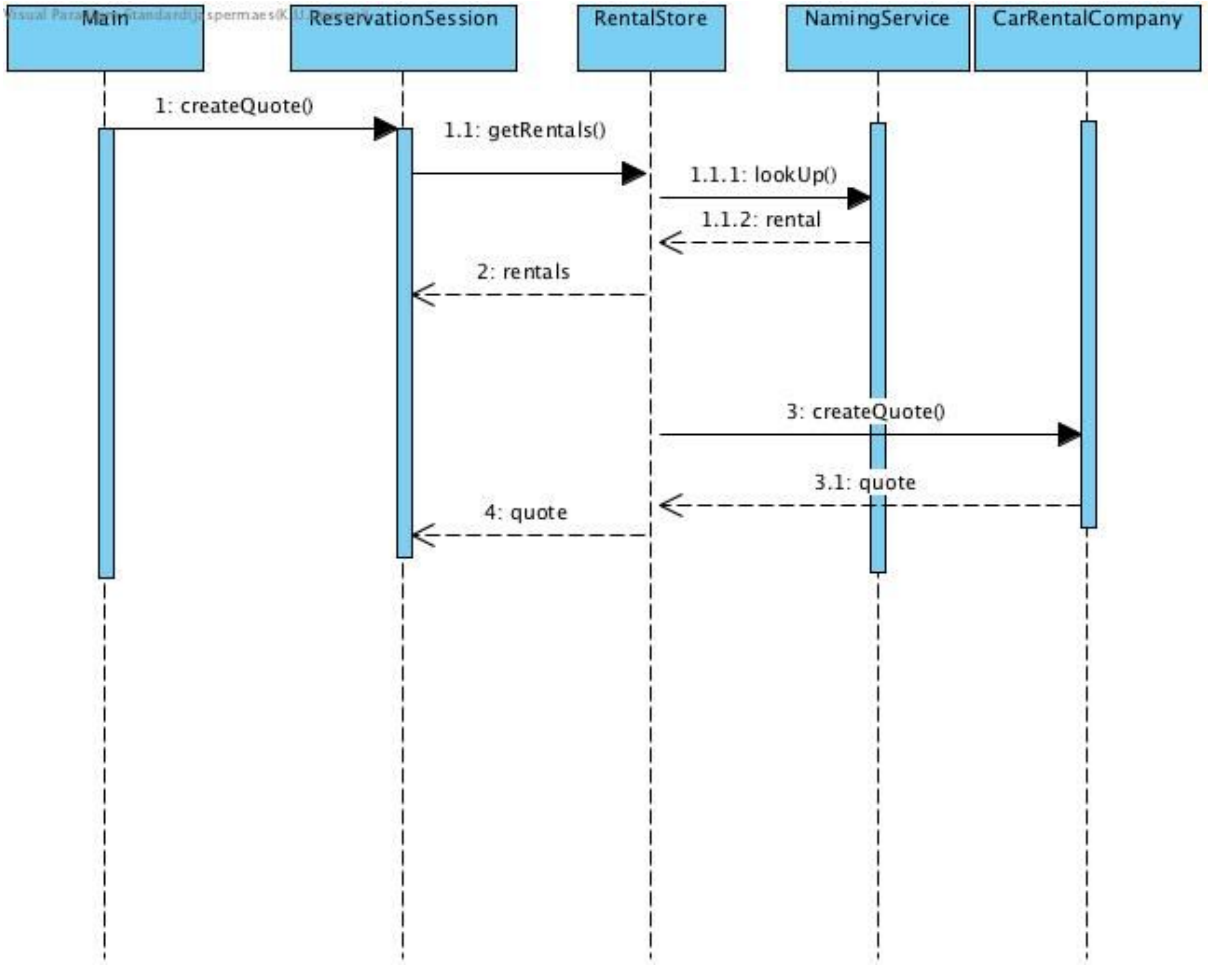


Deployment diagram

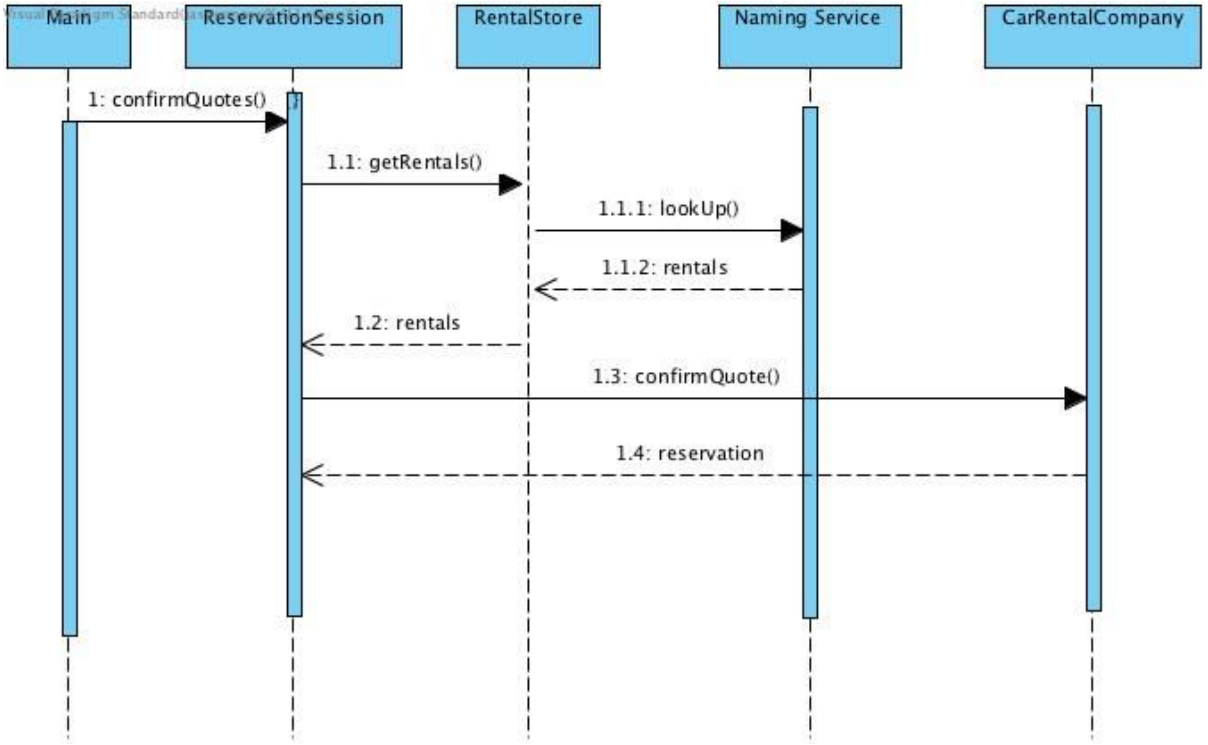


Sequence diagram

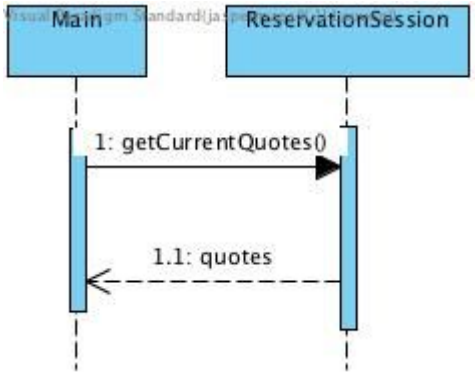
createQuote()



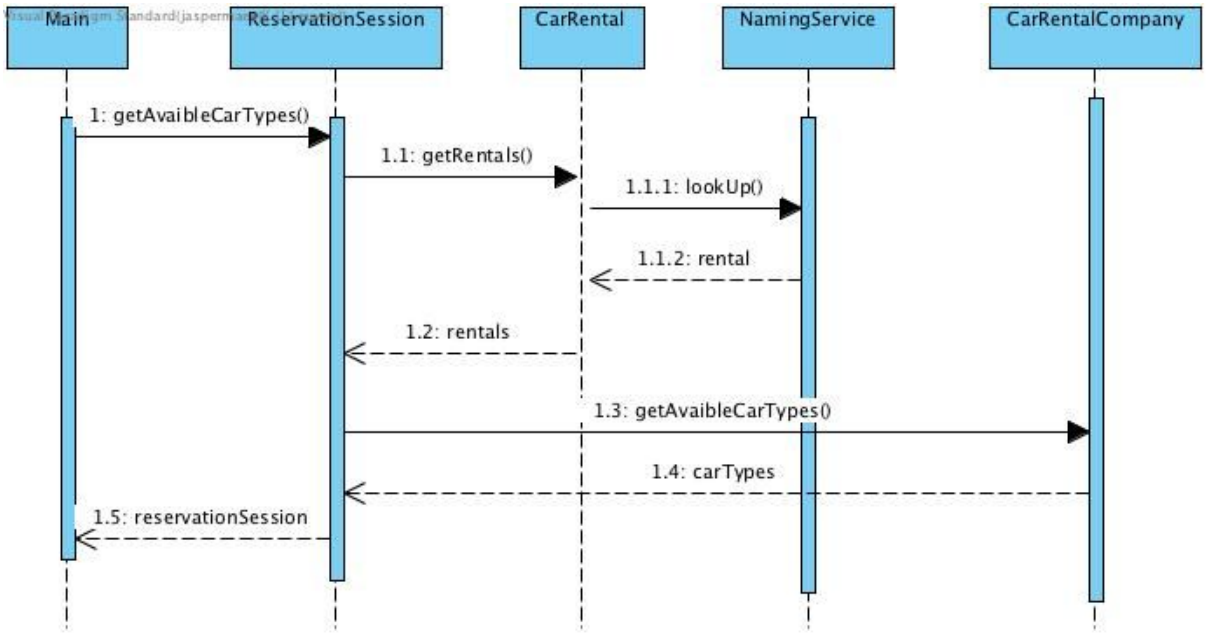
confirmQuote()



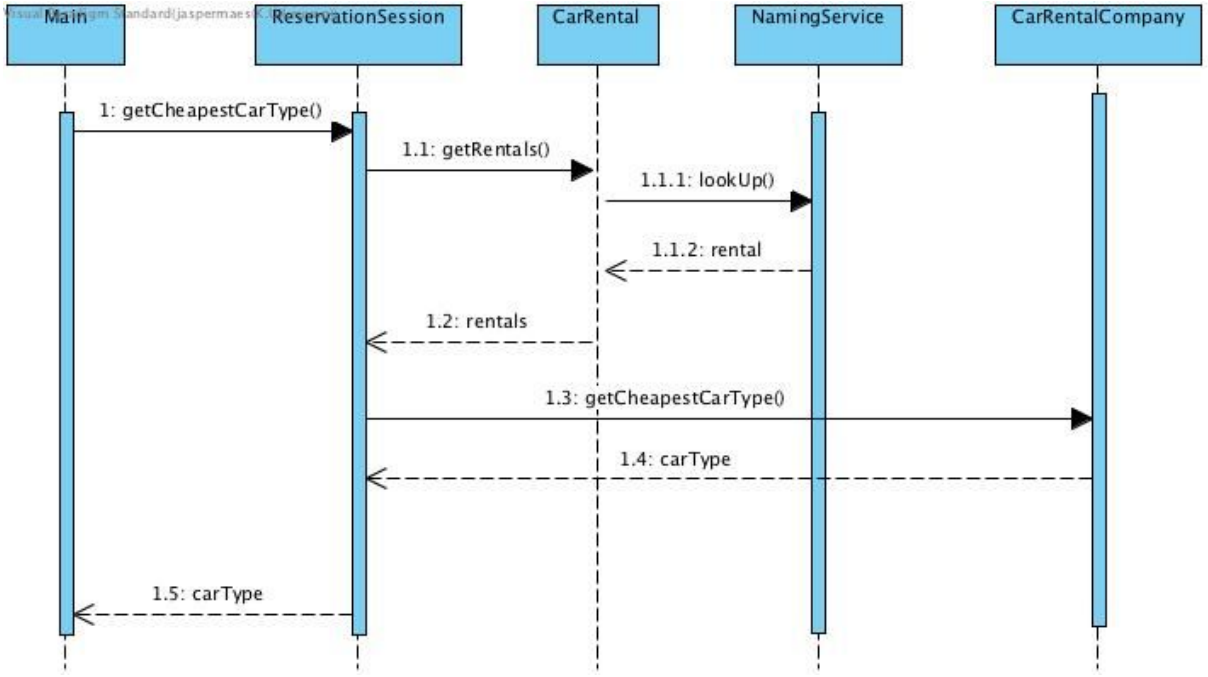
getCurrentQuotes()



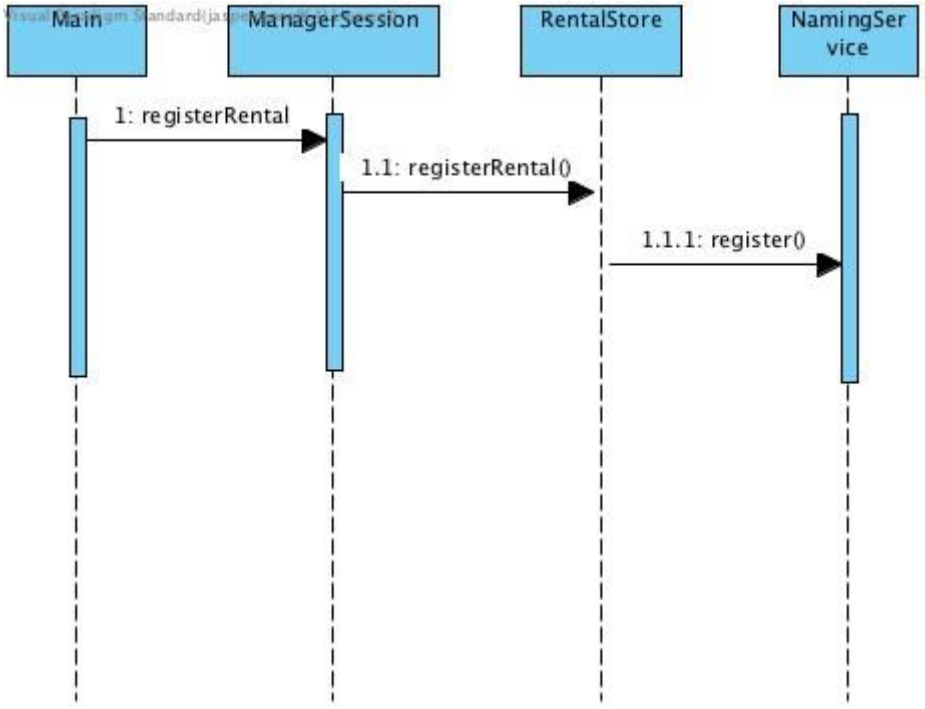
getAvailbleCarTypes()



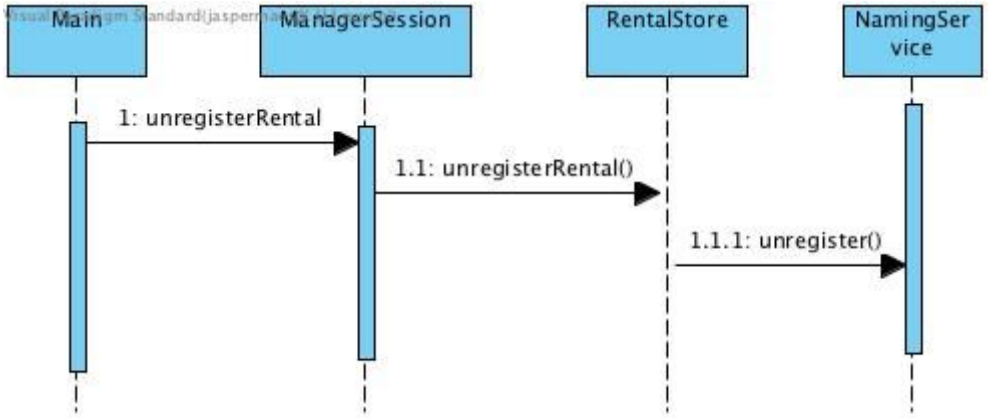
getCheapestCarType()



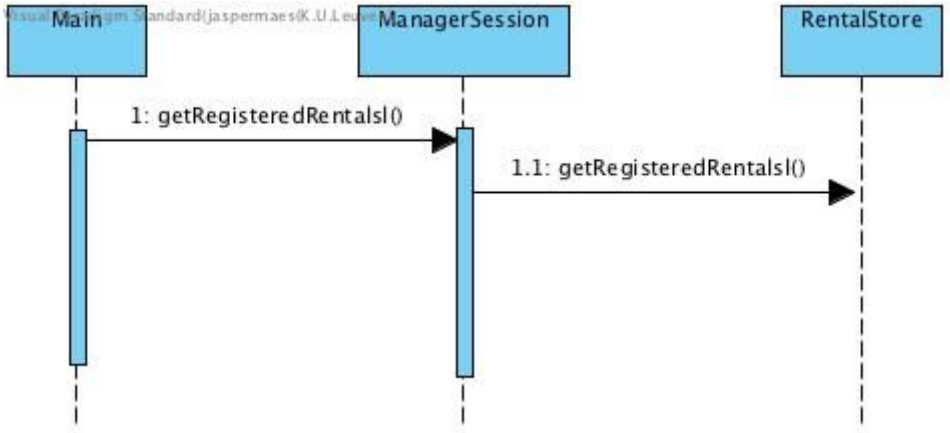
registerl()



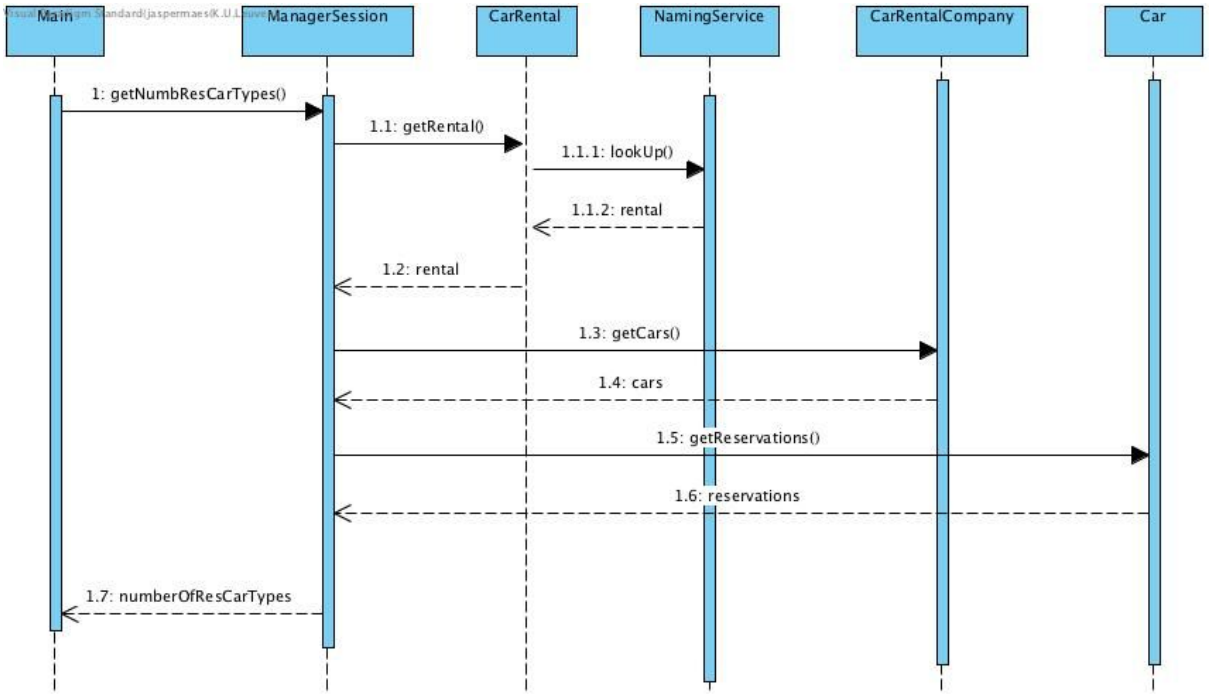
unregister()



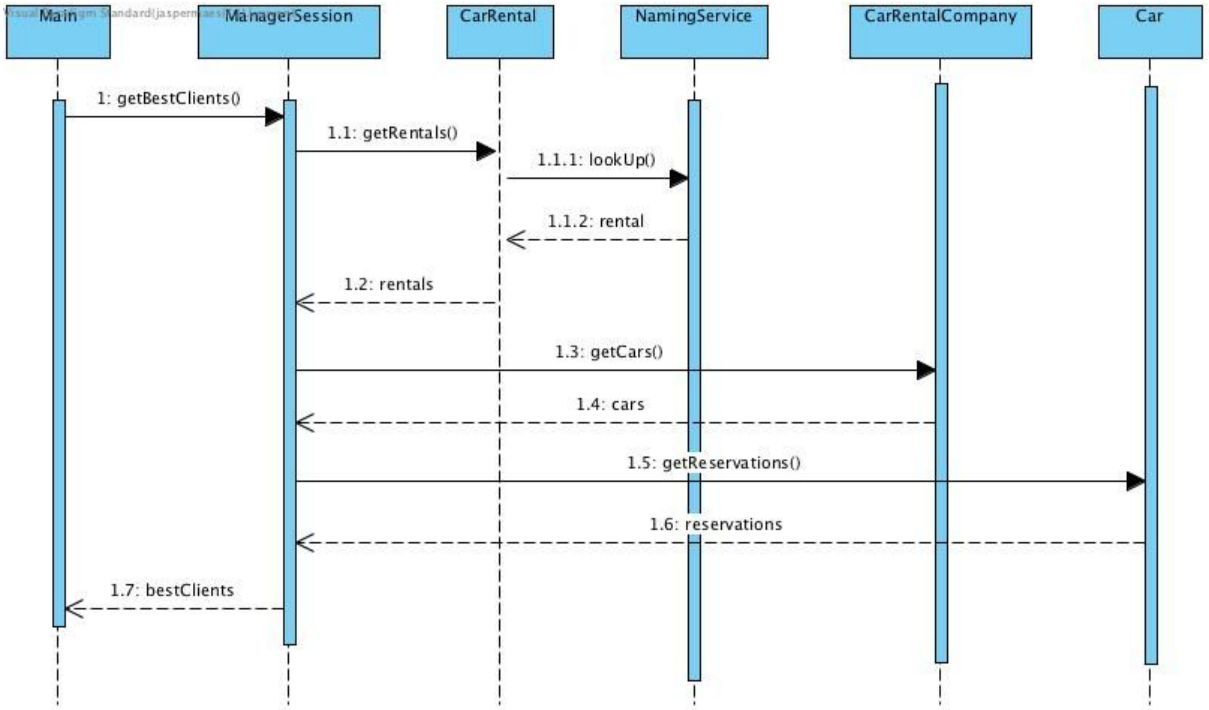
getAll()



getNumberOfReservationsForCarType()



getBestClients()



getMostPopularCarTypeIn()