Assignment A3
Analysis and Design Document

Student: Linca Paul Tudor
Group: 30432

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

Use JAVA/C# to design and implement an application for an online Parking Request System. The main goal of the application is to provide clear and transparent way of requesting and assigning parking spots in the parking lots of Cluj-Napoca. Each citizen can make a new request for a parking spot. The request can be done only for one car that the citizen owns, but he can select multiple parking lots that would suit him good. Each parking lot has a certain number of parking spots. If the citizen has multiple cars, he must file multiple parking requests.

The application should have two types of users: a regular user represented by the citizen and an administrator user, represented by the city clerk responsible for assigning the free parking spots. Both kinds of uses have to provide an email and a password in order to access the application.

1.2 Functional Requirements

The main objective of this assignment is to allow students to become familiar with one of the MV* architectural patterns: MVC, MVP or MVVM.

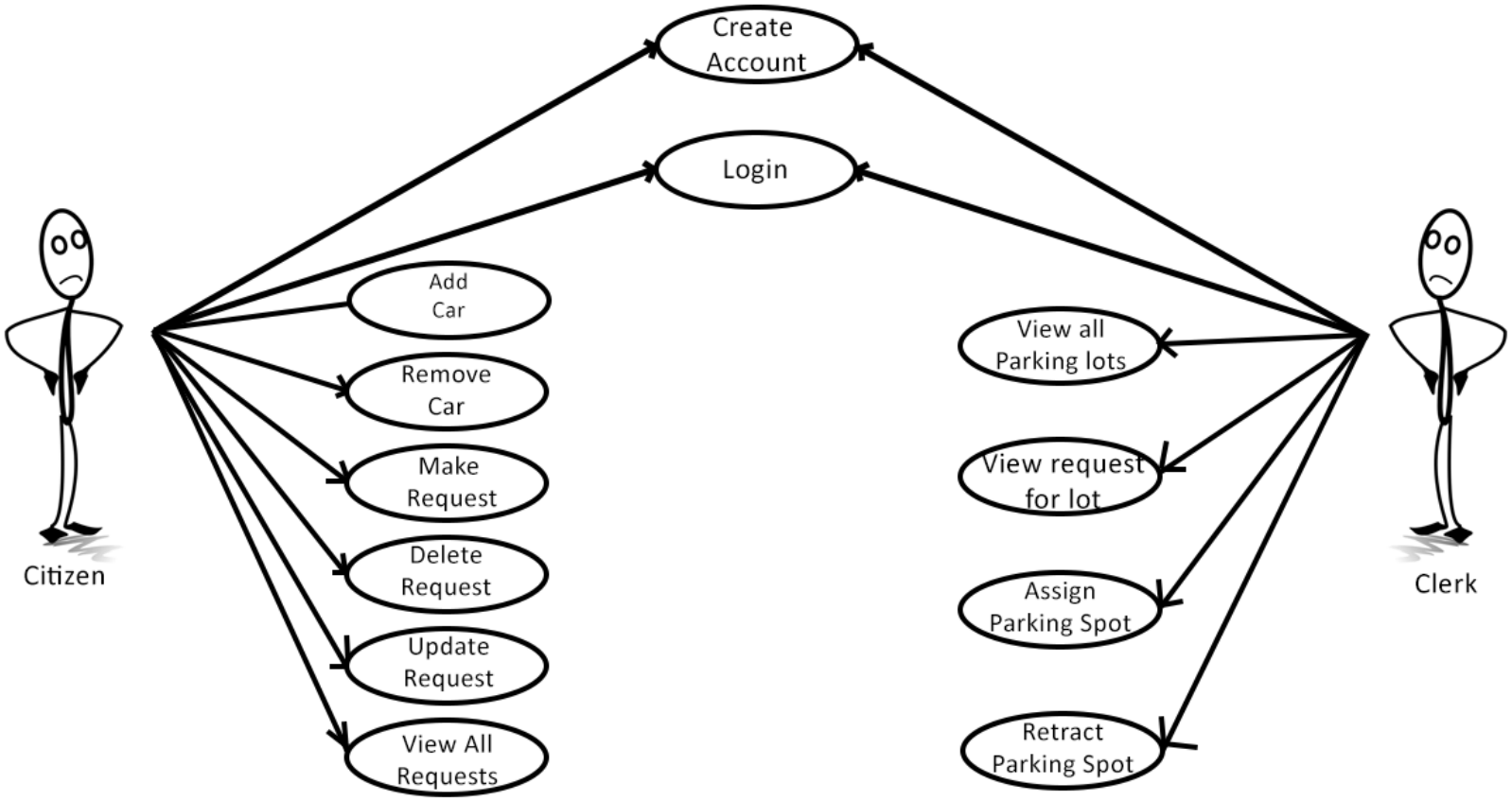
1.3 Non-functional Requirements

- The application should be accessible and easy to use.
- The application must be secure
- The user interface must update instantly (in under 1 second)
- The application must not jam

2. Use-Case Model

Use case: Create new parking request
Level: user-goal level
Primary actor: Citizen
Main success scenario:
1. Open Application.
2. Provide username and password
3. Press the “Login” button.
4. Select the “Make Request” tab.
5. Select the car and parking lots for the request.
6. Press the “Make request” button.
7. The system confirms the request and updates the database.

Extensions: If the PTI of the car is not valid, the system will reject the request and provide a message to the user.



3. System Architectural Design

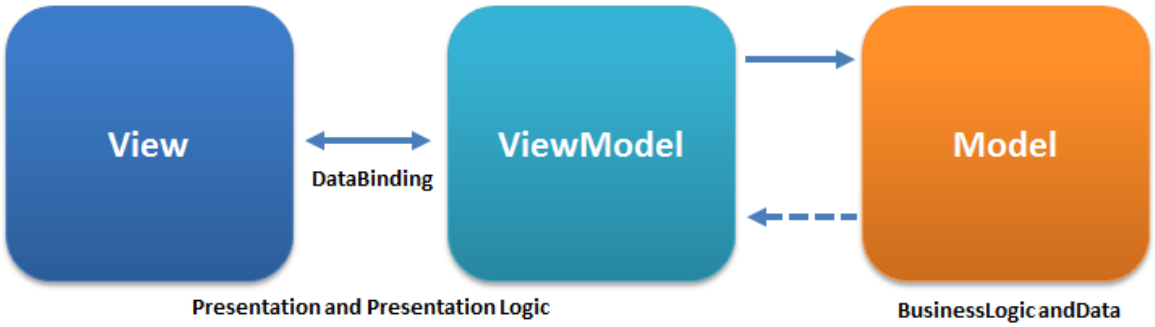
3.1 Architectural Pattern Description

Model–View–ViewModel (MVVM) is a software architectural pattern. MVVM facilitates a separation of development of the graphical user interface from development of the business logic or back-end logic (the data model).

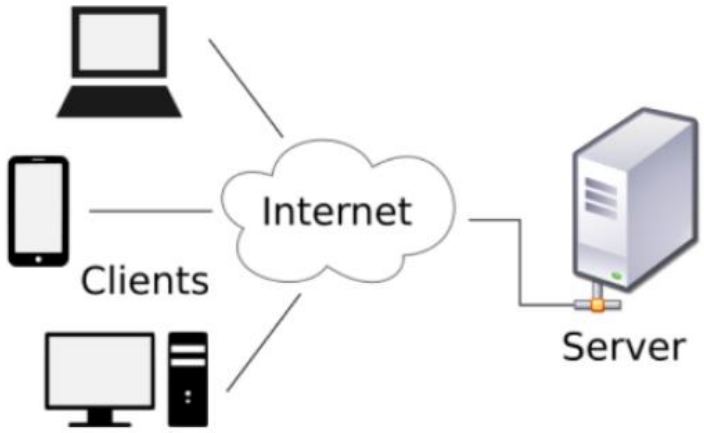
Model refers either to a domain model, which represents real state content, or to the data access layer, which represents content.

The **View** is the structure, layout, and appearance of what a user sees on the screen. It displays a representation of the model and receives the user's interaction with the view, and it forwards the handling of these to the view model via the data binding that is defined to link the view and view model.

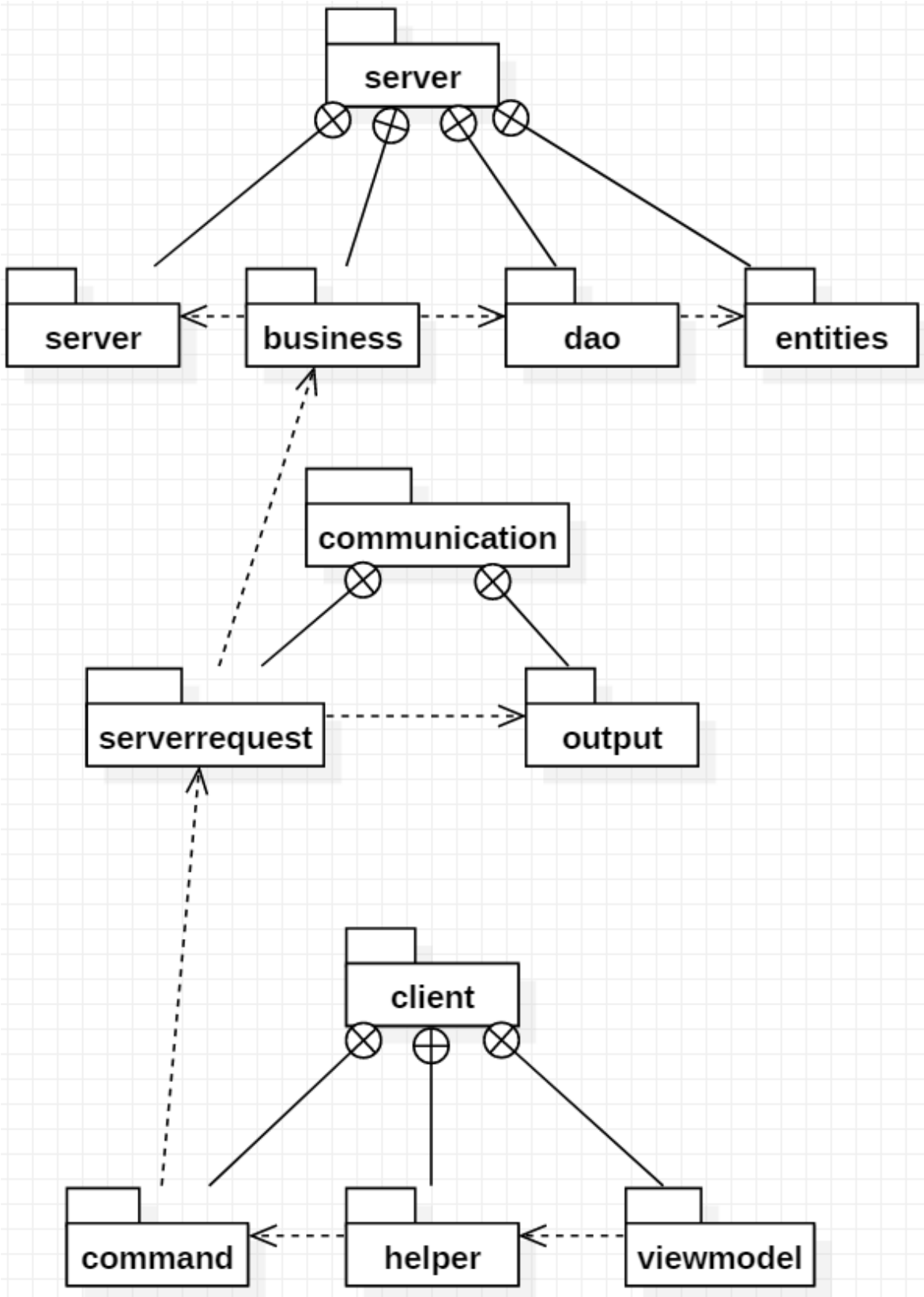
The **View Model** is an abstraction of the view exposing public properties and commands. MVVM has a binder, which automates communication between the view and its bound properties in the view model. The view model has been described as a state of the data in the model.



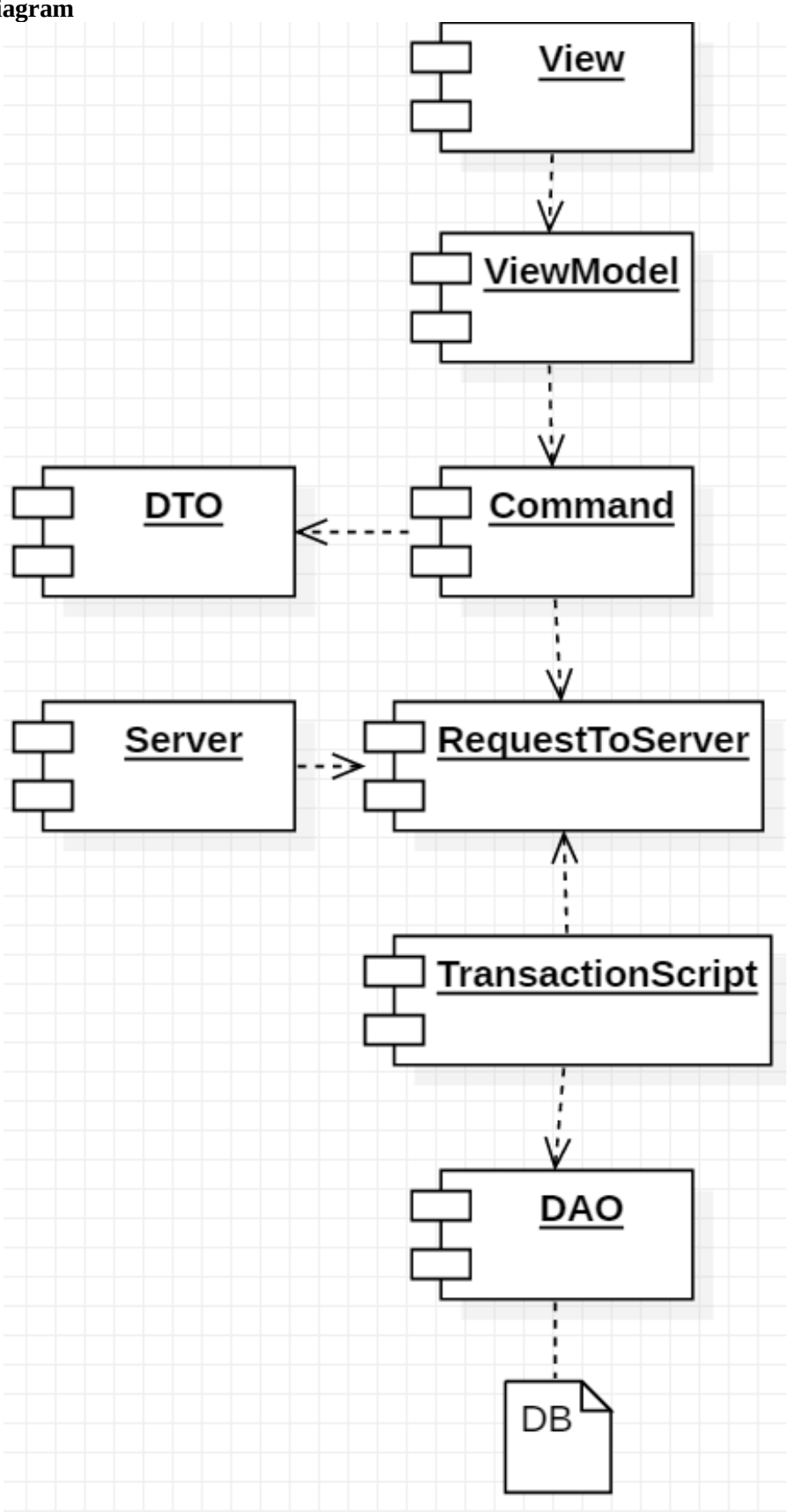
The **client–server** model of computing is a distributed computing structure that partitions tasks or workloads between the providers of a resource or service, called **servers**, and service requesters, called **clients**. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests. Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web.



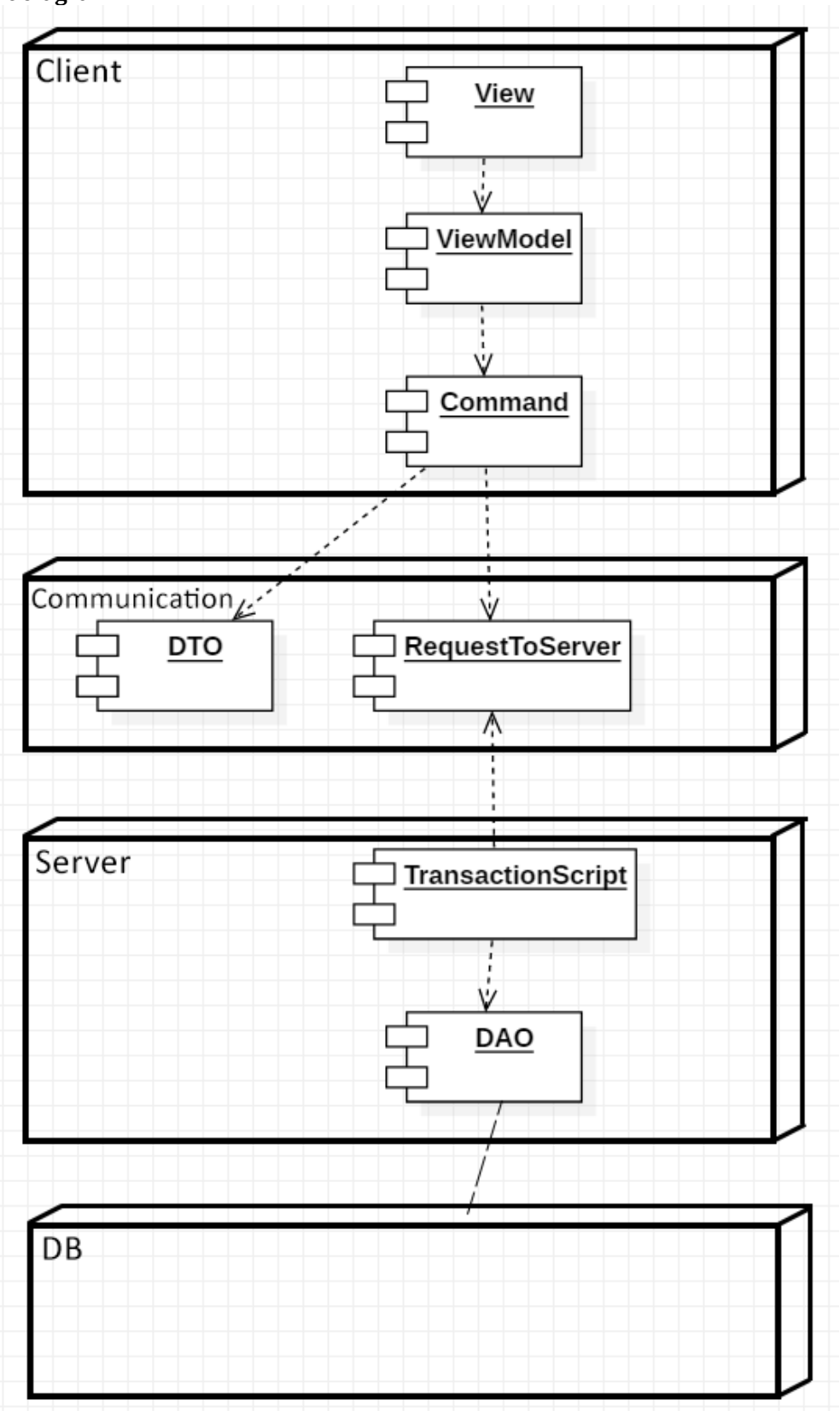
Package Diagram



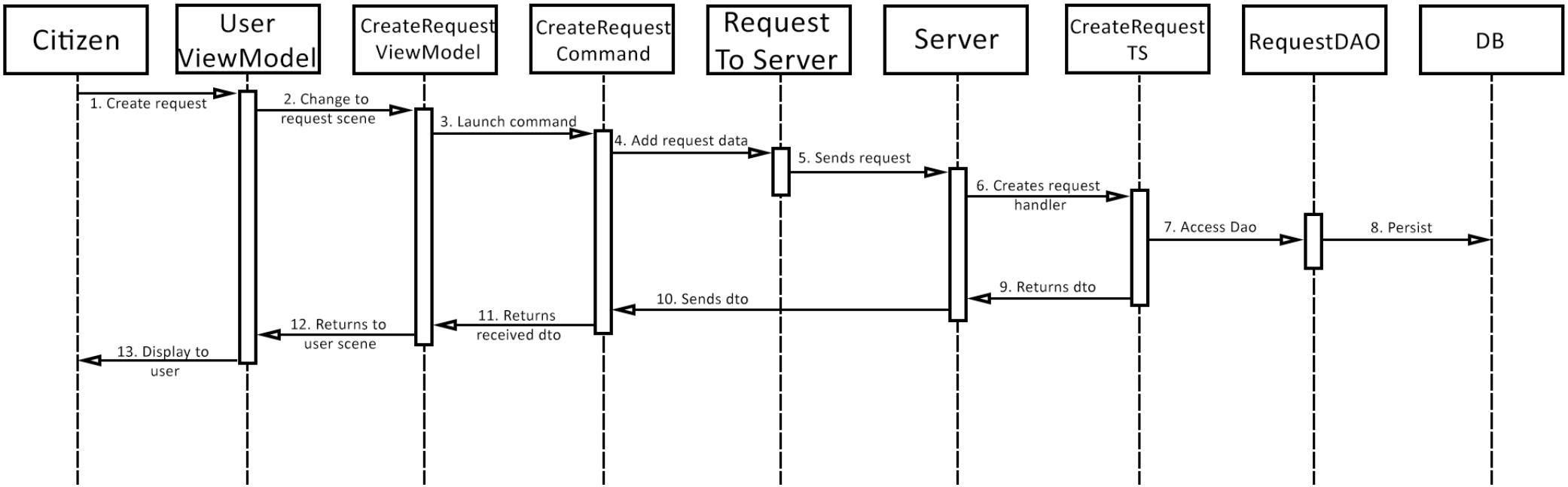
Component diagram



Deployment diagram



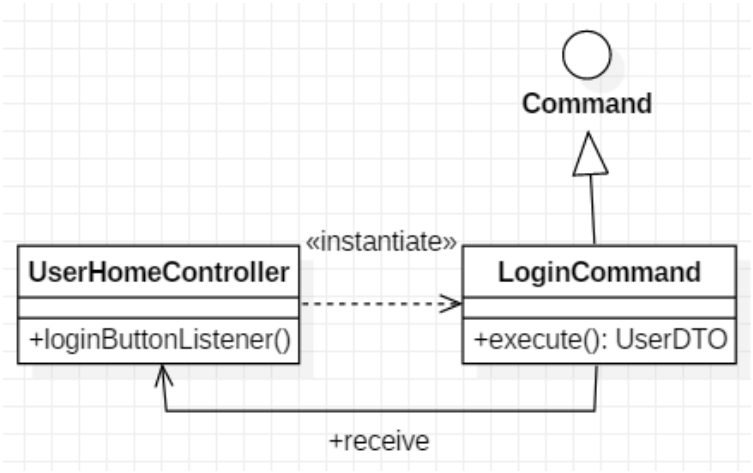
4. UML Sequence Diagrams



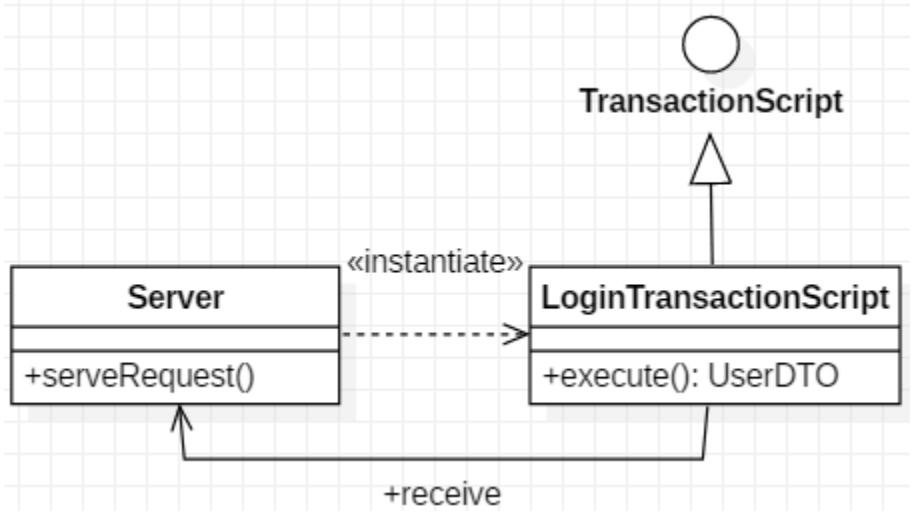
5. Class Design
5.1 Design Patterns Description

The **command** pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

A *command* object knows about receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The *receiver* object to execute these methods is also stored in the command object by aggregation. The *receiver* then does the work when the execute() method in command is called. An *invoker* object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about the command interface. Invoker object(s), command objects and receiver objects are held by a client object, the client decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

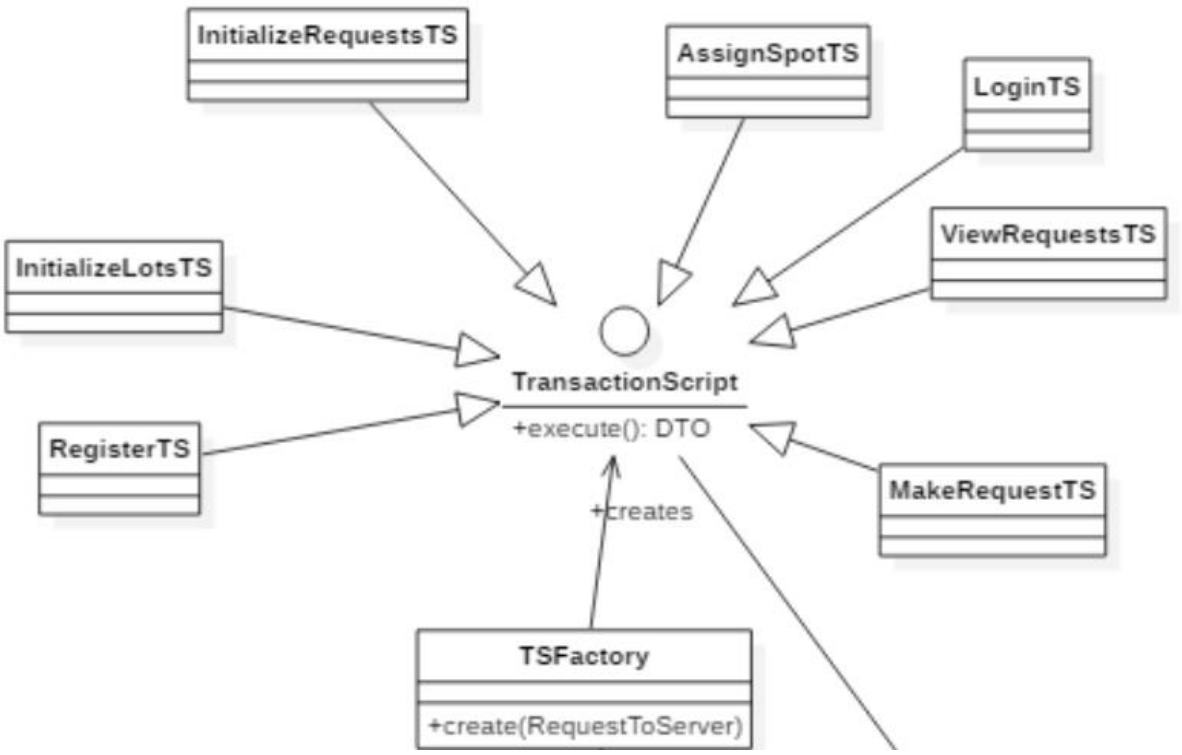


The **Transaction Script** pattern organizes business logic by procedures where each procedure handles a single request from the presentation.

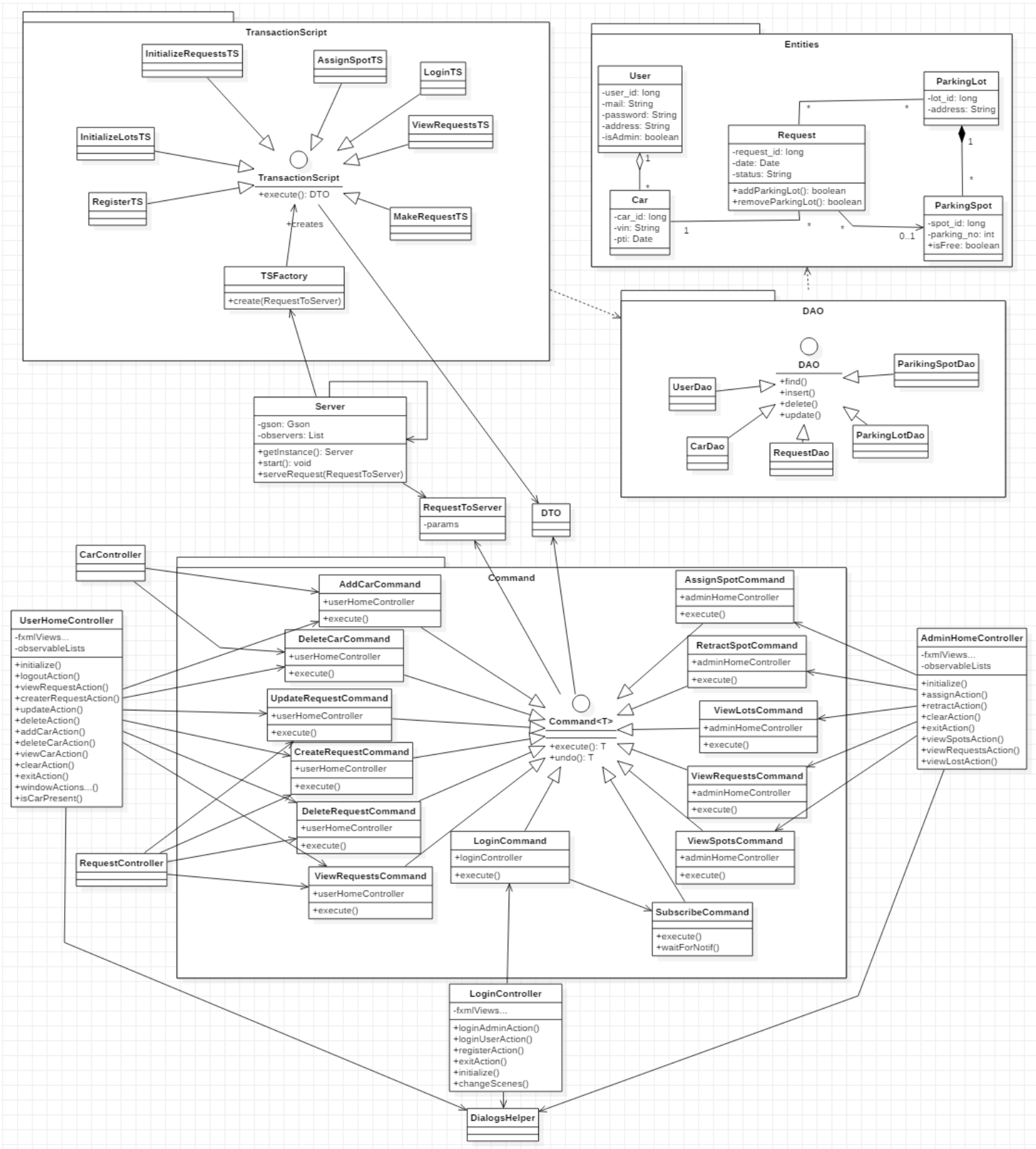


The **Factory** pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

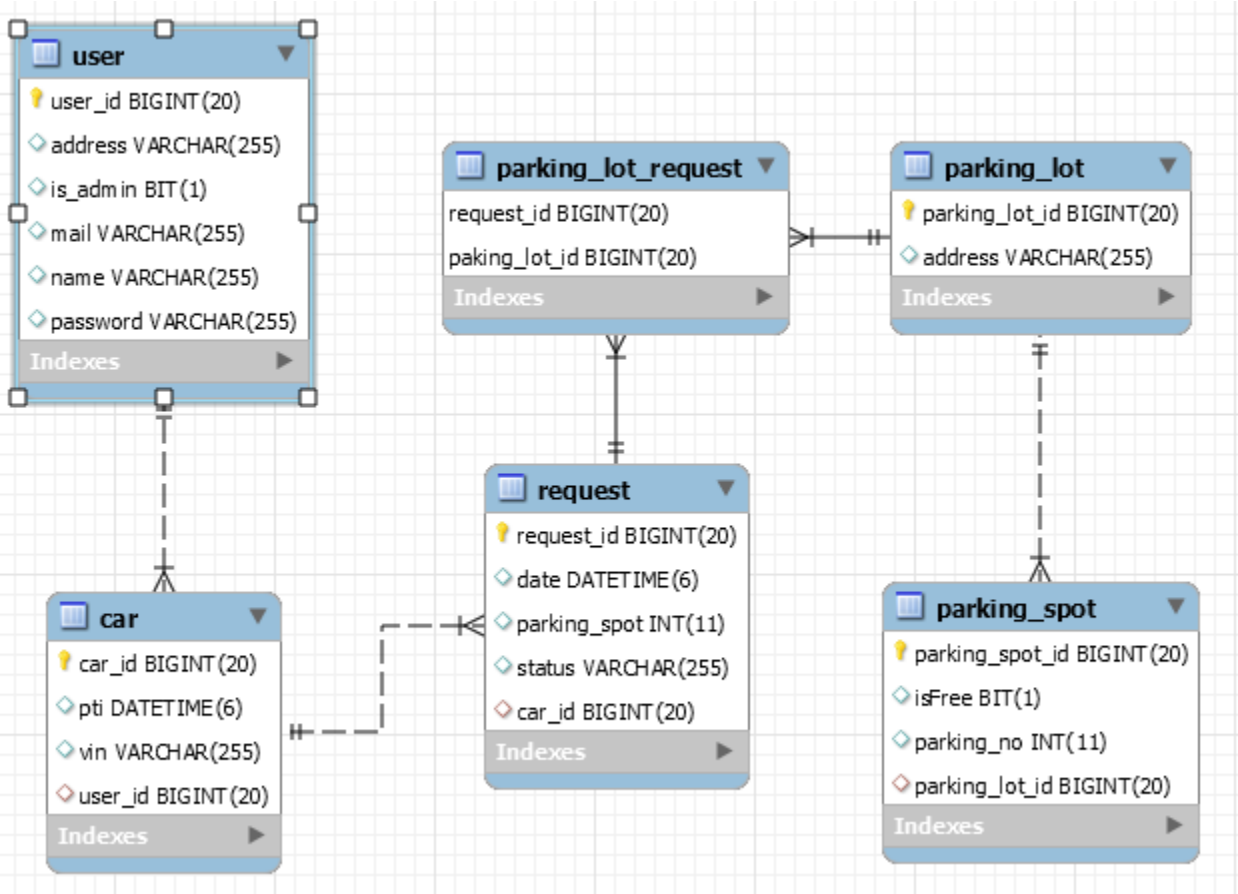
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



5.2 UML Class Diagram



6. Data Model



7. System Testing

The system testing was performed using unit testing for the Data Mappers, integration testing for the integration of the client with the server, and validation testing for the finished system. The unit testing was performed using jUnit in the TestCases class. Integration testing was performed using the graphical interface by testing each functionality in turn for main success and fail scenarios (data-flow strategy). Each Use Case has been tested using the data-flow strategy.

8. Bibliography

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel
https://sourcemaking.com/design_patterns/command
https://github.com/buzea/SoftwareDesignLabResources