Smart City Parkhaus

Hasan, Akcay

7097245
hasan.akcay001@stud.fh-dortmund.de

1 Beschreibung der Projektidee

Es ist ein Problem, dass nicht weniger wird und den Menschen immer mehr Zeit stiehlt. Den Aspekt mit dem Schutz der Umwelt, sollte man auch nicht vernachlässigen. Die Suche nach einem Parkplatz kostet viel Zeit und es ist auch viel Kraftstoff- bzw. Energie (Elektroautos) verbrauch verbunden. Parkplätze sind in den Großstädten eines der großen Probleme und sind immer wichtiger bei den aktuellen Stadtplanungen.

Durch die Technologie, die uns heutzutage zur Verfügung steht, können wir zwar nicht mehr Platz auf der Welt schaffen, jedoch können wir die Zeit, die vorhandenen Parkplätze und den Kraftstoff verbrauch, durch Algorithmen und intelligente Software, effizienter nutzen.

Es gibt die Möglichkeit, alle Parkplätze in der Stadt mit Sensoren abzudecken und diese mit einem Server in Verbindung zu bringen, jedoch ist das ganze recht komplex, da eine Extreme Menge von Sensoren verbaut und eine ganze Infrastruktur aus dem Boden gestampft werden müsste, die natürlich eine Menge Kosten und Zeit mit sich bringen würde. Solche Projekte könnte man in neu entstehenden Städten oder Neubaugebiete einfacher realisieren, aber nicht schon bereits bestehende Parkplätze in der Stadt.

Jedoch kann man ein Smart City Park-Projekt, umso einfacher in Parkhäusern integrieren. Es gibt immer mehr und immer größere Parkhäuser beziehungsweise Parkgaragen. Damit der Fahrer, direkt zu einer Parklücke findet, wird ihm bei der Einfahret ins Parkhaus, ein Signal an das Navigationssystem gesendet. Das Navi im Auto zeigt dem Fahrer, den kürzesten Fahrweg zur nächsten Parklücke an.

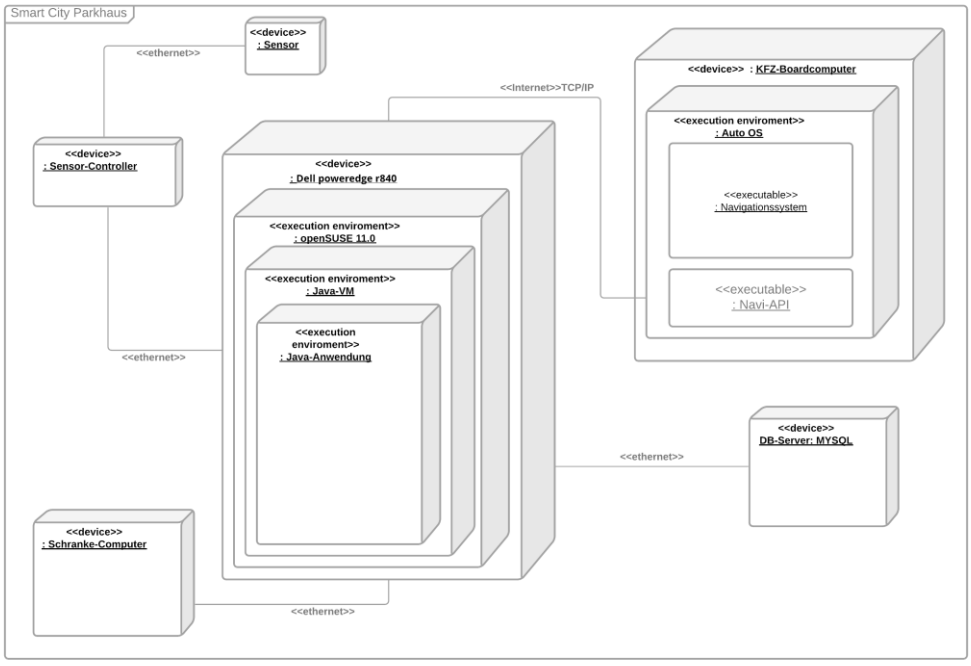
Die Parklücken sind mit Sensor bestückt, diese werden von dem Sensor-Controller gemanagt. Der Sensor-Controller ist mit einem Anwendungssever im Parkhaus verbunden. Sobald eine Parklücke belegt bzw. frei ist, wird eine Nachricht an den Server gesendet. In der Nachricht zum Server, wird die SensorId und der aktuelle Status und die Uhrzeit mit gesendet. Diese Informationen werden vom Anwendungsserver an das

Datenbank-System weitergeleitet, somit haben wir die Informationen persistent gespeichert. Sobald die Information verarbeitet wird, wird der Server die nächstgelegene freie Parklücke in der Datenbank suchen und diese dann dem nächsten Fahrzeug zuspielen.

In dem Projekt wird davon ausgegangen, dass die Autoindustrie sich auf ein Kommunikationsprotokoll bzw. eine API geeinigt hat, um eine Verbindung zu jeweiligem Parkhaus aufzubauen. Über das GPS-Modul im Auto wird erkannt, wo und in welchem Parkhaus, das Auto sich aktuell befindet. Sobald die Schranke im Parkhaus betätigt wird, verbindet sich das Auto mit dem Onlineserver des Parkhauses und die nächstgelegene Parklücke wird ins Navigationssystem eingespeist. Da das Navigationssystem Zugriff auf die Fahrmeter hat, kann Meter genau bestimmt werden, wo sich das Auto aktuell befindet, und kann den Fahrer bzw. das Auto (autonomes Fahren) zu der reservierten Parklücke weisen.

2 Architektur des Projektes

2.1 Beschreibung der Systemarchitektur



Geräte:

- Sensor, Sensor-Controller
- Anwendungsserver
- Datenbankserver
- KFZ-Boardcomputer
- Schranke-Computer

Kommunikation:

- Anwendungsserver, Datenbankserver, Sensoren, Sensor-Controller, Schranke-Computer nutzen Ethernet
- KFZ-Boardcomputer und Anwendungsserver nutzen das Internet über TCP/IP

Anwendungsserver

- Hardware: Dell poweredge r840
- Betriebssystem: openSUSE 11.0
- Java VM: In der VM läuft die Anwendung für das Parkhaus Management System

Datenbankserver:

- Mit MYSQL-Server Installation

Sensor, Sensor-Controller:

- Der Sensor Controller ist mit den Sensoren verbunden und kann alle zustände der Sensoren lesen. Sobald eine Änderung am Sensor stattfindet, erhält der Sensor-Controller die Daten.

KFZ/ Automobil:

- Hat einen Boardcomputer mit einem eigenen Betriebssystem
    - Navigationsapplikation
    - Server-API / Schnittstelle

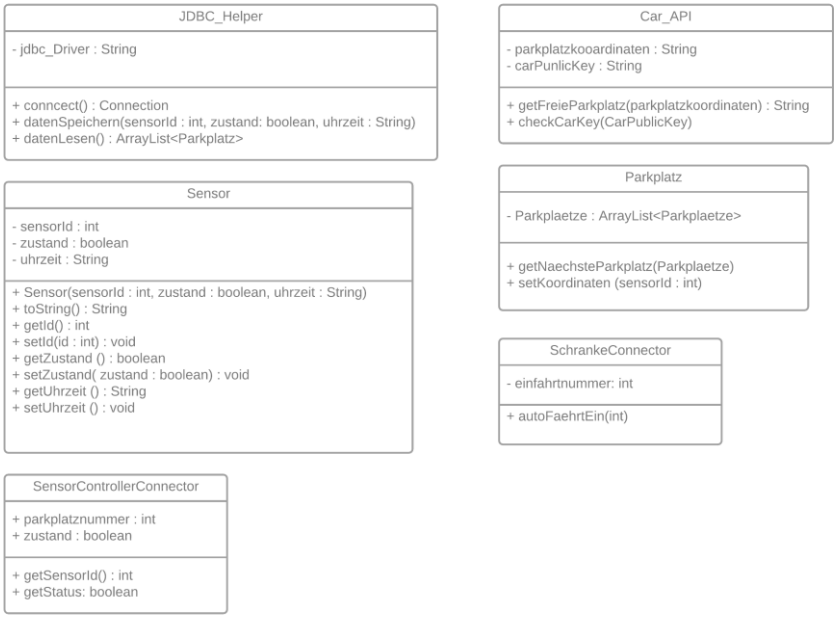
2.2 Einordnung der Architekturentscheidungen

Die Broker-Architektur passt zu diesem Projekt am besten. Da der **Client** bzw. das Fahrzeug, nur die Schnittstellendefinition kennt. In dem Fall verbindet sich der KFZ-Boardcomputer sich mit dem Server über das Internet.
Beim **Service** werden die Parameter zur nächsten Parklücke berechnet und bereitgestellt. Der **Broker** selber ist die Schnittstelle in dem Fall und führt die Kommunikation zwischen Fahrzeug und Server durch.
Das CORBA-Objekte (Broker), der einen Mechanismus bereitstellt, um Anfragen des Clients (KFZ-Boardcomputer) an die Implementierung weiterzuleiten, ohne Abhängigkeit des Betriebssystems oder des Standortes im Netzwerk zu haben, ist dies auch bei verschiedenen Autohersteller mit verschiedenen Betriebssystemen bei den Fahrzeugen passend. Durch den sogenannten „Stub“, wird dem Boardcomputer die Operationsschnittstelle zur Verfügung gestellt.

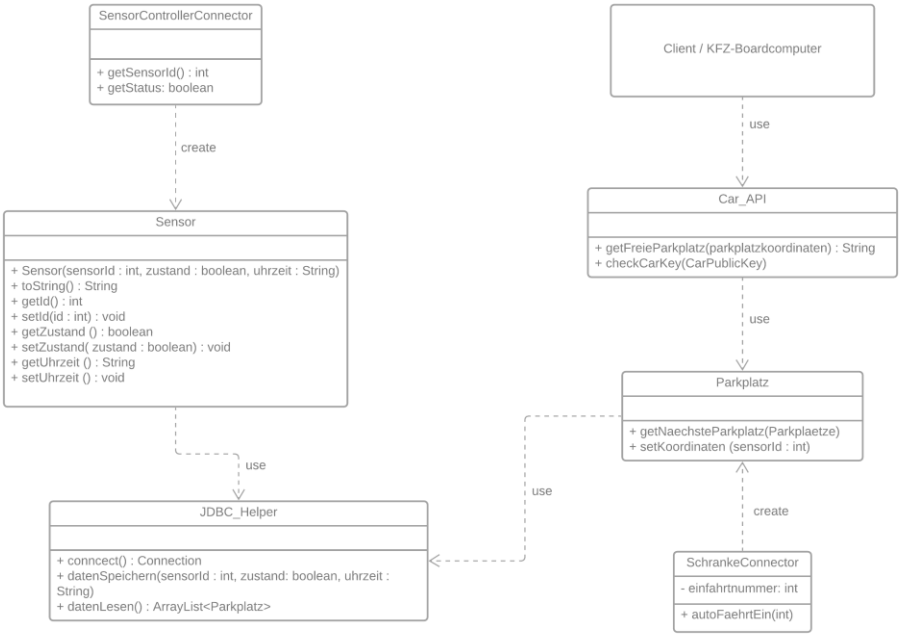
3 Entwurfsmuster

3.1 Darstellung der Entwurfsmuster als Klassendiagramm

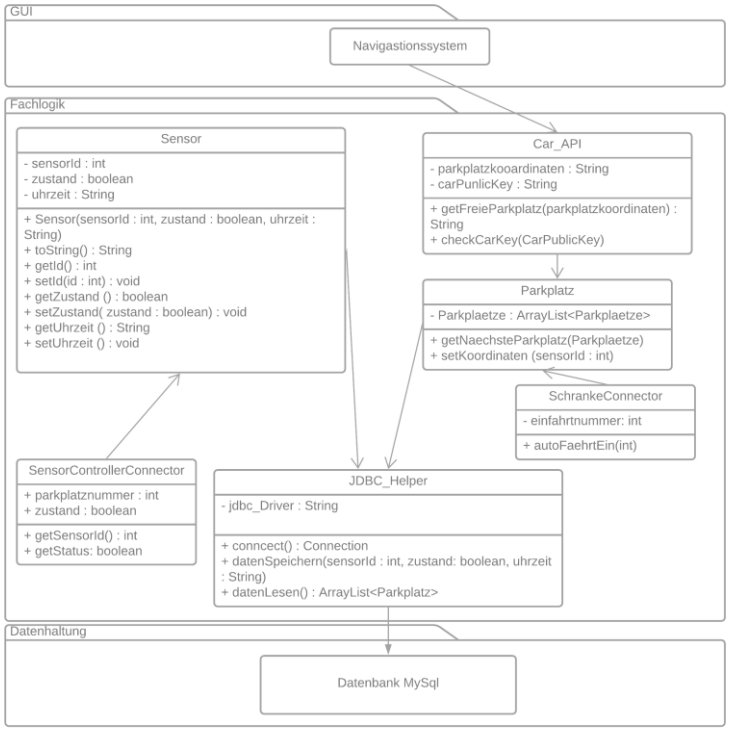
OOD Klassendiagramm



Entwurfsmuster – Fabrikmethode



Beobachter-Muster (objektbasiertes Verhaltensmuster)



Die Klasse „**JDBC_Helper**“ stellt eine Verbindung zum MYSQL Server her und stellt auch die Schnittstelle zu dem Ausführen der Datenbankabfragen und Übertragung der Daten für die Speicherung bereit. So sind die Daten Persistent in der Datenbank über diese Klasse gespeichert und abrufbar.
Der „**SensorControllerConnector**“ erhält die Parameter der Sensoren von dem Sensor-Controller und gibt die Daten, sobald ein Sensor eine Änderung durchführt, an die „**Sensor**“ Klasse weiter. In der „Sensor“ Klasse werden die Attribute mit dem Sensor Objekt verbunden und strukturiert, um mit den Objekten in der Java-Anwendung besser weiterzuarbeiten.
In der Klasse „**Parkplatz**“ wartet eine Methode auf den Event von der Klasse „SchrankeConnector“. Sobald aus der Klasse „**SchrankeConnenctor“** ein Ereignis ausgeht, wird in der „Parkplatz“ Klasse, nach der nächsten Parklücke gesucht. Hierfür wird die Methode ***getNaechsteParkplatz*** ausgeführt. In dieser Methode wird eine aus den Daten, der „Parkplatze“ Liste, der nächstgelegene Parkplatz gefunden. Anschließend wird dieser an die „**Car_API**“ Klasse übergeben, dort ist die Schnittstelle zwischen dem Anwendungsprogramm und des KFZ-Boardcomputers, über die dann das Navigationssystem des Autos, die Koordinaten zur nächsten Parklücke erhält.

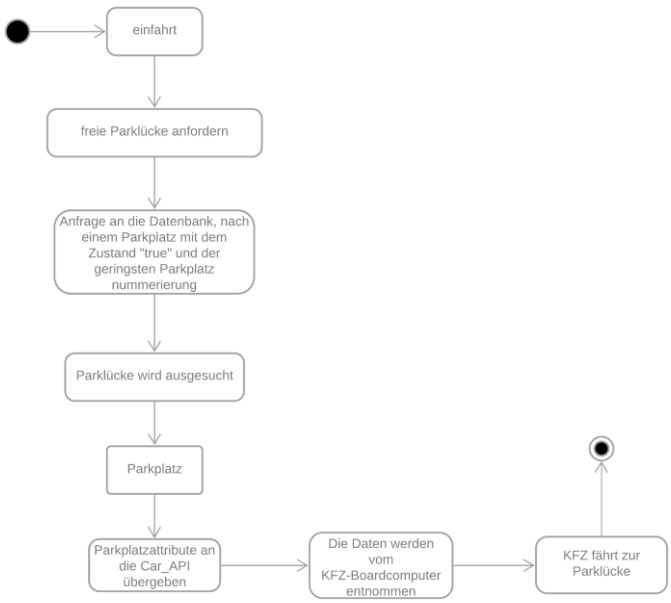
3.2 Begründung

Die Fabrikmethode verbildlicht den weg gut von der Verbindungsschnittstelle des KFZ-Boardcomputers zu den Datenabruf Klassen und die Beziehungen zwischen den Klassen. Die Fabrikmethode ist ein klassenbasiertes Erzeugungsmuster. In Smart-City-Parkhaus gibt es keine Vererbungen, also auch keine Abstrakte Klassen, daher ist eine Vererbung im Muster, wie im Originalen Entwurfsmuster nicht dargestellt. „Fabrikmethoden verhindern, dass man anwendungsspezifische Klassen in den Codes des Frameworks anbinden muss“[1].

Das Beobachter-Entwurfsmuster ist ein objektbasiertes Verhaltensmuster. Bei dem eine strenge „Drei-Sichten-Architektur“ verwendet wird. Die GUI Schicht wird in diesem Fall von dem Automobile-Boardcomputer erzeugt und kann nur auf die API-Schnittstelle der Anwendung zugreifen, somit hat er keine direkten Zugriffe auf die Daten bzw. die Datenbank. Die Fachkonzeptschicht wird in diesem Fall von der Java-Anwendung ausgeführt. Durch die JDBCHelper Klasse kann nur die Logik-Schicht auf die Datenbank zugreifen. Der Vorteil ist, dass der Client bzw. Navigationssystem unabhängig sein kann und somit eine Flexibilität herrscht.

4 Beschreibung der Kernfunktionalität

Aktivitätsdiagramm



[1] (https://epdf.pub/lehrbuch-der-objektmodellierung-analyse-und-entwurf.html, kein Datum)

Im Beispiel Quellcode wird die Einfahrt per Konsoleneingabe ausgelöst. Normalerweise sollte es durch betätigen der Schranke passieren.

Klasse SchrankeConnector

```
import java.util.Scanner;

public class SchrankeConnector {

	public static void main(String[] args){
		Parkplatz p = new Parkplatz();
		System.out.println("0: Auto fährt nicht ein\n"
				+ "1: Auto fährt ein \n"
				+ "--------------------------\n"
				+ "Wählen: ");

		Scanner sc = new Scanner(System.in);
		int einfahrt = sc.nextInt();

		switch(einfahrt) {
		case 0: System.out.println("Schranke unten");
		break;
		case 1: System.out.println("Schranke öffnet");
		p.getNaechsteParkplatz();
		break;
		}
	}
}
```

Klasse Parkplatz

```
public class Parkplatz {
	public void getNaechsteParkplatz() {
		System.out.println("Nächster Parkplatz wird abgefragt");
		Car_Api car_api = new Car_Api();
		car_api.getKoordinaten_Car_Api();
	}
}
```

Klasse JDBCHelper

```
import java.sql.*;

import com.mysql.cj.protocol.FullReadInputStream;

public class JDBCHelper {
	private static final String
url="jdbc:mysql://localhost:3306/smartparkhaus";

public static String sucheNaechsteParkplatz() {
String sqlString = "SELECT MIN(Parkplatznummer), Koordinaten,sensor_SensorId,
Zustand,Uhrzeit FROM smartparkhaus.parkplatz right Join sensor ON park-
platz.sensor_SensorId = sensor.SensorId where Zustand like 0;";
String freierParkplatzKoordinaten = null;
		try(Connection con = DriverManager.getConnection(url,
"root",""))
		{
			Class.forName("com.mysql.cj.jdbc.Driver");
			PreparedStatement stmt =
con.prepareStatement(sqlString);
			ResultSet rs = stmt.executeQuery();
			while(rs.next())
			{
				int parkplatznummer = rs.getInt(1);
				String koordinaten = rs.getString(2);
				int sensorId = rs.getInt(3);
				boolean zustand = rs.getBoolean(4);
				Timestamp uhrzeit = rs.getTimestamp(5);
				System.out.print("Nächster feier Parkplatz: ");
				System.out.println("Parkplatznummer: "+ park-
platznummer+" Koordinaten: "+koordinaten + " SensorId: "+sensorId +" Zustand:
"+zustand + " Uhrzeit: " + uhrzeit);
freierParkplatzKoordinaten = koordinaten;
			}
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}
		return freierParkplatzKoordinaten;
	}
}
```

Ein Freier Parkplatz wird in diesem Fall, durch den Boolean Wert „false“ angegeben.

Car_Api Klasse

```
public class Car_Api {
	public String koordinaten_Car_Api;


	public void getKoordinaten_Car_Api() {
		koordinaten_Car_Api = JDBCHel-
per.sucheNaechsteParkplatz();
	}
}
```

```
Ausgabe:
0: Auto fährt nicht ein
1: Auto fährt ein
---------------------------
Wählen:
1
Schranke öffnet
Nächster Parkplatz wird abgefragt
Nächster feier Parkplatz: Parkplatznummer: 3 Koordinaten:
51.49609460591684, 7.4170059474480883 SensorId: 3 Zustand:
false Uhrzeit: 2021-05-08 08:50:50.0
```

5 Persistente Datenhaltung

5.1 Persistierungansatz

Die Daten werden auf einem MYSQL Datenbank System verwaltet. Es handelt sich um eine Relationale Datenbank, die Daten werden in Form von Tabellen gespeichert. Eine Zeile der Tabelle repräsentiert ein Objekt der Klasse, in diesem Fall ist das Objekt der Parkplatz. Da die Parkplätze durchnummeriert sind, wird die Parkplatznummer als Primärschlüssel genutzt. Somit sind die Datensätze eindeutig zu identifizieren. Die SensorId Fremdschlüssel werden in den Parkplätzen Datensätzen sein, da es eine Liste mit Sensoren und eine Liste mit Parkplätzen gibt und diese durch eine Fremdschlüssel Assoziation in einer Beziehung stehen müssen, damit eine zu Ordnung der Sensoren und Parklätzen realisiert werden kann. Diese stehen in einer 1 zu 1 Beziehung, denn jedem Parkplatz ist ein Sensor zugeordnet.
Die Verbindung zwischen der Datenbank und der Fachlogik wird mit dem JDBC (Java Database Connectivity) gelöst, dieser „bietet auf Basis von SQL eine einheitli-

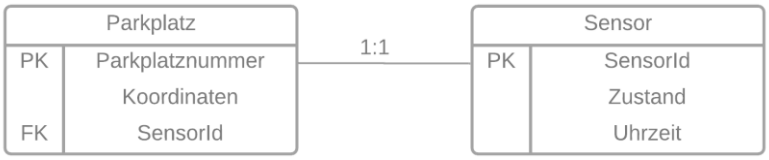
che und herstellerunabhängige Schnittstelle zwischen Java-Anwendungen und relationalen Datenbanken“[2] an.

Tabelle Parkplatz:

- **Parkplatznummer:** Speichert die Parkplatznummerierung und dient als Primärschlüssel der Tabelle
- **Koordinaten:** Hier sind die Koordinaten der Parklücken hinterlegt, diese werden an das Navigationssystem weitergeleitet.
- **SensorId:** Dient Fremdschlüssel der Sensor Tabelle und verbindet so die Datensätze miteinander.

Tabelle Sensor:

- **SensorId:** Die Sensoren haben eine eindeutige Identifizierungsnummer
- **Zustand:** Gibt an, ob die Parklücken besetzt oder frei ist
- **Uhrzeit:** Speichert die Uhrzeit der zuletzt geführten Änderung ein.



5.2 Begründung

Das von mir gewählte Relationale Datenbankmodell ist in der Praxis weitverbreitet und der meist genutzte auf dem Markt. Dieser ist auch einfach zu realisieren und zuwarten. Einer der Hauptvorteile ist die Entkopplung des Datenmanagements von der Java-Anwendung selber. Der Nachteil ist, dass die Daten in der Datenbank zusätzlich gemanagt werden müssen.

Eine weitere Möglichkeit, wäre die Java Persistence API (JPA), dieser ist ein objektrealtionaler Mapper für Java-basierte Anwendungen, dieser nutz aber auch über eine einzelne Sitzung hinaus, eine relationale Datenbank. Die zu speichernde Daten werden in einer Klasse mit dem „javax.presistence-*“ Import realisiert. Die einzelnen Attribute werden mit Getter und Setter Methoden bestückt. Die Annotation wird mit „@Entity“ und der Primärschlüssel mit „@Id“ markiert, der mit „@GeneratedValue“ eine Automatische Generierung der ID erhält.
Um das Ganze zu nutzen, muss auch ein Entity Manager implementiert werden, der die Schnittstelle zur Datenbank, für Objekte finden, persistieren und entfernen anbietet.

[2] (Sachweh)

6 Kommunikation

6.1 Kommunikationsansatz

Die Parkplatzdaten werden durch einen Online-Server bereitgestellt. Die Modernen Autos haben die „Cellular V2X[3]“ Lösung eingebaut, dieser ermöglicht den Fahrzeugen über das Mobilfunknetz zu kommunizieren. Das KFZ-Boardcomputer verbindet sich ganz normal als Client zu dem Server und fragt nach den Daten des nächstfreien Parkplatzes an. Anschließend antwortet der Server dem Boardcomputer diese gibt die Daten, dann an das Navigationssystem weiter. Die Kommunikation ähnelt einer Kommunikation zwischen Smartphone und einem Onlinedienst.
Die Kommunikation wird auf einem Verteiltem System realisiert, bei dem die TCP/IP Protokolle genutzt werden. Das Fahrzeug (Client) verbindet sich mit dem Parkhausserver, sobald die Schranke betätigt wird, startet die Java Anwendung. Anschließend wird eine Antwortnachricht vom Server zum Fahrzeug (Client) gesendet. Heißt das ganze wird über eine Synchrone Kommunikation gewährleistet, da der Client anfragt und auf die Rückantwort des Servers wartet.
Da die Datenbank selber im selben Netzwerk, wie der Anwendungsserver ist, wird die Kommunikation über Ethernet realisiert. Das Fahrzeug selber verbindet sich über das Internet mit dem Parkhausserver. Die Schnittstelle in der Java-Anwendung selber wird über den Schnittstellendefiniton und Dokumentation Swagger realisiert.
Die Kommunikation von dem Schranken-Computer und des Sensor-controllers findet auch über das Ethernet statt, da die Systeme sich auch im Parkhaus befinden.

6.2 Begründung

Sollte ein Fahrzeug die Kommunikation abbrechen, müsste eine neue Anfrage gestartet werden, da das Fahrzeug in Verbindung mit dem Server sein muss.
Zu der Synchronen Kommunikation würde auch die Asynchrone Kommunikation stehen, dort würde das Fahrzeug eine Nachricht an das Serversystem senden, der Server würde daraufhin die Nachricht in eine Warteschlage tun und nimmt es entgegen, sobald er empfangsbereit ist. Ein Vorteil wäre eine Entkopplung von Sender und Empfänger, dies würde über eine Point to Point Verbindung realisiert werden. Somit werden auch Laufzeitabhängigkeiten minimiert, darüber hinaus ist die Kommunikation weniger anfällig.
Die aktuelle Kommunikationsart würde bei einem Parkhaus mit einer Einfahrt in Ordnung sein, da die Autos nacheinander einfahren. Bei mehreren Einfahrten und Ausfahrten würde eine Entkopplung der Fahrzeuge vom Server mit einer Asynchronen Kommunikation deutlich sinnvoller sein, da die Nachrichten dort Nacheinander vom Server abgearbeitet werden würde. Somit ist gewährleistet, dass das erste Fahrzeug was einfährt, auch als erstes eine Antwort erhält.

[3] (Wikipedia, 2021)

Literaturverzeichnis

https://epdf.pub/lehrbuch-der-objektmodellierung-analyse-und-entwurf.html. .
Sachweh, P. D. *11_SWT2_Persistente_Datenhaltung_I, Folie 9*.
Wikipedia. (2021). Von https://de.wikipedia.org/wiki/Cellular_V2X abgerufen
Sachweh, SWT2-Vorlesungen