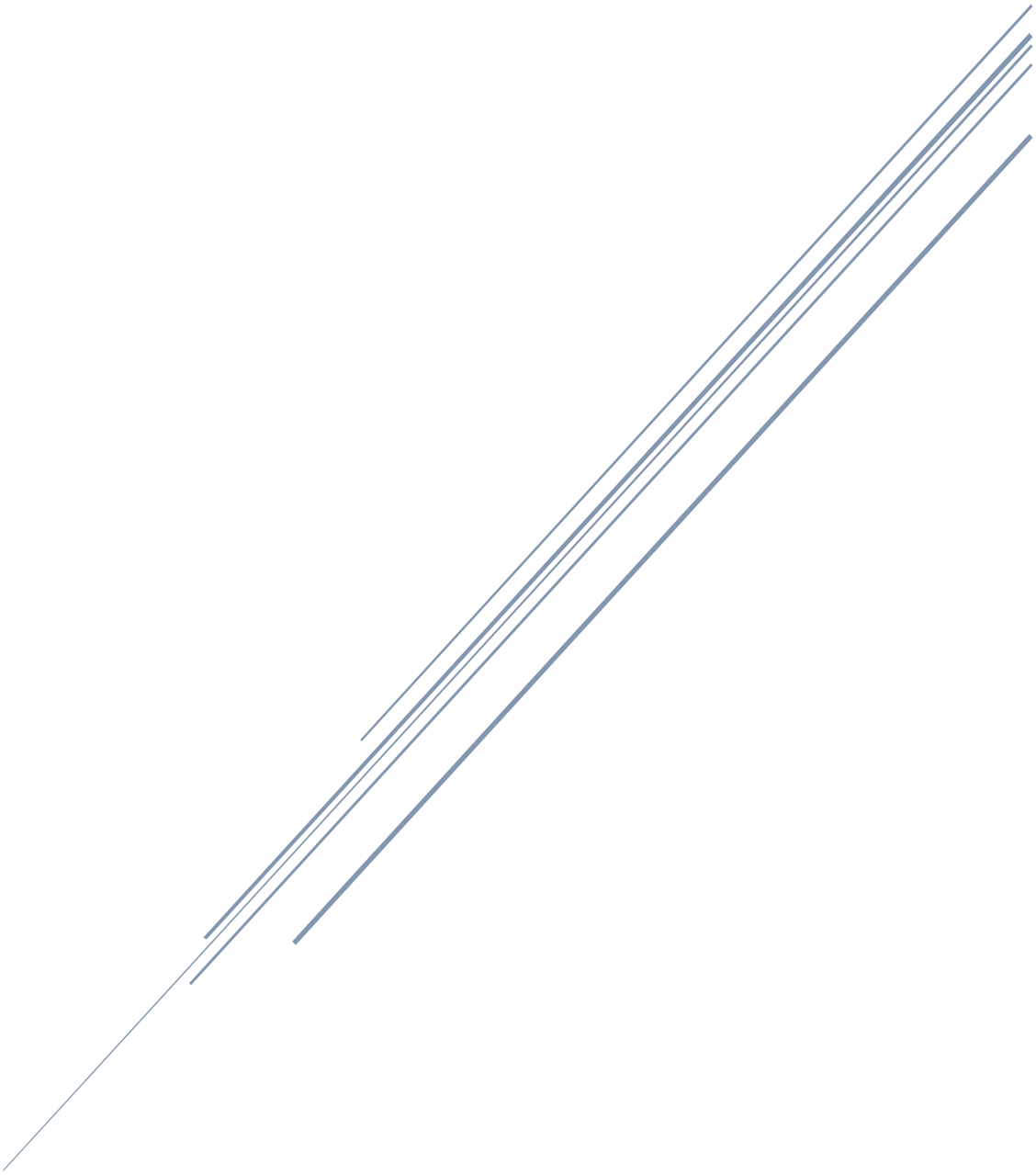
CARLOGGER

Johannes Ens, Willi Schäfer, Martin Ziel



FH Bielefeld

Mobile Applikationen WS 17/18

Inhaltsverzeichnis

Einleitung

Als Halter eines Kfz stellt man sich häufig die Frage welche Kosten das Fahrzeug verursacht. "Wann ist die nächste Inspektion oder der nächste Ölwechsel fällig?" Neue Fahrzeuge loggen viele Informationen. Doch ist man der Besitzer eines älteren Fahrzeuges, stehen oft wichtige Informationen wie etwa der genaue Kraftstoffverbrauch nicht zur Verfügung. Diese Informationen können helfen zu sparen und einen genauen Überblick über Kosten, fällige Werkstatt-Termine oder Reparaturen schaffen. Werden Reparaturen geloggt, kann es sich positiv auf den Verkaufswert des Fahrzeuges auswirken.

Die Android App Carlogger stellt ein nützliches Tool zur Verfügung mit dem das Loggen rund um das Auto ermöglicht wird.

Es können Einträge verfasst werden, diese werden online gesichert, wodurch die App auch auf mehreren Geräten vom selben User genutzt werden kann.

Außerdem werden Statistiken über die Kosten und den Spritverbrauch des eigenen Fahrzeuges sowie die Fahrzeuge der anderen Benutzer erstellt.

Um den Benutzer eine komfortable Nutzung der App zu ermöglichen gibt es die Möglichkeit einen Eintrag automatisiert eintragen zu lassen.

Möchte der Nutzer die Funktion der automatischen Einträge nutzen, kann er einen Eintrag erstellen und die Funktion „Automatischer Eintrag“ aktivieren.

Nach der Aktivierung erfolgt die Auswahl eines Intervalls, in dem der Eintrag automatisch hinzugefügt wird. Die Auswahl des Intervalls ist abhängig von der Kategorie, so kann ein Tanken-Eintrag täglich, wöchentlich oder monatlich automatisch hinzugefügt werden, ein Service-Eintrag aber auch zusätzlich jährlich und auch alle zwei Jahre.

Diese Funktion soll dem Nutzer erlauben einen ungefähren Überblick über seine Kosten zu haben, ohne gezwungen zu sein jeden Eintrag per Hand einzutragen.

Stand der Technik

Ziel dieses Projekts war es eine App für mobile Endgeräte zu entwickeln, welche dem Nutzer einen möglichst weiten Umfang an anwenderfreundlichen Funktionen bietet, ohne dass hierfür alle Bausteine, von Grund auf, ausprogrammiert werden müssen. Um die App einer möglichst großen Gruppe an Nutzern zugänglich zu machen, wird das API-Level 21 verwendet, was der Android-Version 5.0 entspricht. So haben 80% der Android Nutzer Zugang zu der App.

So werden, um dem Nutzer Zeitraumwähler und *Floating-Action-Button*s mit hoher Usability anzubieten, bereitgestellte Lösungen von anderen Android-Entwicklern implementiert.

Des Weiteren, wird die aktuelle Version der *Support*-Bibliothek benutzt, welche zum Zeitpunkt des Verfassens dieses Dokuments die Version 26.1.0 ist. Das Projekt implementiert die Bausteine *support, appcompat, design* und *constraint-layout*. Durch Verwendung dieser Bibliotheken können Funktionen neuerer Android-Versionen auch auf älteren Geräten verwendet werden.

Um Unit-Tests für das Projekt zu schreiben, wird die in Java üblicherweise verwendete Bibliothek JUnit in der Version 4.12 verwendet.

Da, wie in diesem Dokument schon beschrieben, der Nutzer die Möglichkeit haben soll, sich mit seinem Google-Account identifizieren zu können und Zugang zu einer Echtzeitdatenbank haben soll, werden verschiedene Projekte von Google implementiert. Dies betrifft *die play-services plus* und *auth*, sowie *firebase auth* und *database*. Die genannten Projekte werden wieder in der aktuellen Version in das Projekt implementiert, welche zum Zeitpunkt des Verfassens dieses Dokuments die Version 11.8.0 ist.

Anforderungen

Zielgruppe:

Als Zielgruppe sollen Besitzer eines mobilen Gerätes und Fahrer eines PKWs angesprochen werden. Die App ist zunächst auf mobile Geräte mit dem Android Betriebssystem beschränkt. Die Kerngruppe sind kostenbewusste Personen mit einem oder mehreren Fahrzeugen. Das Alter und die technische Affinität sollen nicht entscheidend für die Benutzung sein, um eine weite Benutzerbasis zu schaffen. Personen die oft ihr Fahrzeug verkaufen, können dieses, durch das Loggen der entstandenen Kosten, besser Bewerten. Die App soll geräteübergreifend funktionieren.

Grundfunktionen:

Die Grundfunktionen der Carlogger App sind Logging, Statistiken und Erinnerungen. Die Daten werden online gesichert, wodurch die App auch auf mehreren Geräten vom selben Benutzer genutzt werden kann. Folgend werden die vom Benutzer eingetragenen Daten auch Einträge genannt.

Logging:

Mit der App können Benutzer Daten für die Kategorien „Tanken“, „Reparaturen und Service“ und „Andere Kosten“ für ein Fahrzeug loggen. Ein Benutzer kann dabei mehrere Fahrzeuge haben und zwischen diesen wechseln. Die Einträge der Kategorie „Service und Reparaturen“ werden in vier Arten unterschieden. „Reparatur“, „Reparatur in der Werkstatt“, „TÜV“, „Inspektionen“ und „Andere“. Den „Service und Reparaturen“-Einträgen kann außerdem ein Bildbeleg der Rechnung angehängt werden, somit lassen sich die Kosten und Leistungen nachweisen und die Datenbank kann als digitales Checkheft genutzt werden. Für die Eintragskategorien kann der Benutzer ein automatisches Eintragen in bestimmten Zeitintervallen aktivieren. Diese Zeitintervalle werden von der App vorgegeben mit täglich, wöchentlich, monatlich, 2-monatlich, 3-monatlich, jährlich und 2-jährlich.

Statistiken:

Aus den Einträgen des Benutzers werden Kosten und Durchschnittswerte für ein Fahrzeug berechnet. Je mehr Benutzer eines Fahrzeugmodells ihre Daten loggen, umso genauer werden die Statistiken.

Die Gesamtkosten sind die Kosten aller Einträge und können als Durchschnittswerte für ein Jahr oder Monat angezeigt werden.

Die Reparaturkosten sind die Kosten aller Einträge aus der Kategorie „Reparaturen und Service“ der Arten „Reparatur“ und „Reparatur(Werkstatt)“. Können gesamt, als Monatsdurchschnitt und für ein bestimmtes Zeitintervall angezeigt werden.

Die Service und Verschleißteile-Kosten sind die Kosten aller Einträge aus der Kategorie „Reparaturen und Service“ der Arten „TÜV“, „Inspektion“ und „Andere“. Können Gesamt, als Monatsdurchschnitt und für ein bestimmtes Zeitintervall angezeigt werden.

Der Spritverbrauch sind die Kosten und Durchschnittswerte aller Einträge aus der Kategorie „Tanken“. Es können die Gesamtkosten, Monatsdurchschnitt, Durchschnitt eines bestimmten Zeitintervalls, Durchschnittswert L/100km oder Durchschnittswert €/100km angezeigt werden.

Zum Vergleich mit anderen Fahrern desselben Fahrzeugmodells werden folgende Durchschnittswerte berechnet:

Gesamtkosten Monatsdurchschnitt,

Reparatur und Service Monatsdurchschnitt,

Spritverbrauch Kosten Monatsdurchschnitt,

Spritverbrauch Durchschnittswert L/100km.

Erinnerung:

Als Erinnerung wird die Kalenderfunktion bezeichnet. Der Benutzer kann hier seine Termine mit einer Bezeichnung und Uhrzeit eintragen. Sie werden fahrzeugunabhängig eingestellt. Bei Bedarf wird eine Push-Notification, zum vom Benutzer eingestellten Zeitpunkt, ausgelöst.

Übersicht der Eintragskategorien und Attribute:

1. Tanken
    a. Menge in Liter
    b. Kosten/Liter
    c. Kilometerstand
    d. Ist Vollgetankt
    e. Automatischer Eintrag
2. Reparaturen und Service
    a. Art
        i. Reparatur (Werkstatt)
        ii. Reparatur
        iii. TÜV
        iv. Inspektion
        v. Andere
    b. Teilekosten
    c. Arbeitskosten
    d. Beschreibung/ersetzte Teile
    e. Automatischer Eintrag
    f. Bild der Rechnung
3. Andere Kosten
    a. Beschreibung
    b. Kosten
    c. Automatischer Eintrag
4. Erinnerungen
    a. Beschreibung/Name
    b. Datum
    c. Uhrzeit
    d. Push-Notification

Funktionen vertieft:

„Anmeldung“

Die Anmeldung erfolgt mittels Google Account. Für die Nutzung der App muss der Benutzer eingeloggt sein. Zum Wechseln des Benutzers an einem Gerät, gibt es eine abmelde Funktion. Nach dem Abmelden kann ein anderer Account zum Anmelden verwendet werden.

„Auto wählen“

Nach dem Login wird in der Datenbank nach vom Benutzer ausgewählten Fahrzeug geschaut. Ist ein Fahrzeug bereits ausgewählt wird das UseCase „Einträge anzeigen“ ausgeführt. Wenn kein Fahrzeug ausgewählt ist, wird der Benutzer aufgefordert eins auszuwählen. Aus einer Liste mit allen Fahrzeugen kann dann nach der HSN und TSN aus dem Fahrzeugschein oder der Bezeichnung ein Fahrzeug gewählt werden. Dieses wird dann in die Datenbank der ausgewählten Fahrzeuge des Benutzers, als aktives Fahrzeug für das Loggen gespeichert.

„Einträge anzeigen“

Ist ein Fahrzeug zum Loggen ausgewählt, werden alle Einträge aller Kategorien zu diesem Fahrzeug aus der Datenbank geladen und in einer Liste angezeigt.

„Eintrag hinzufügen“

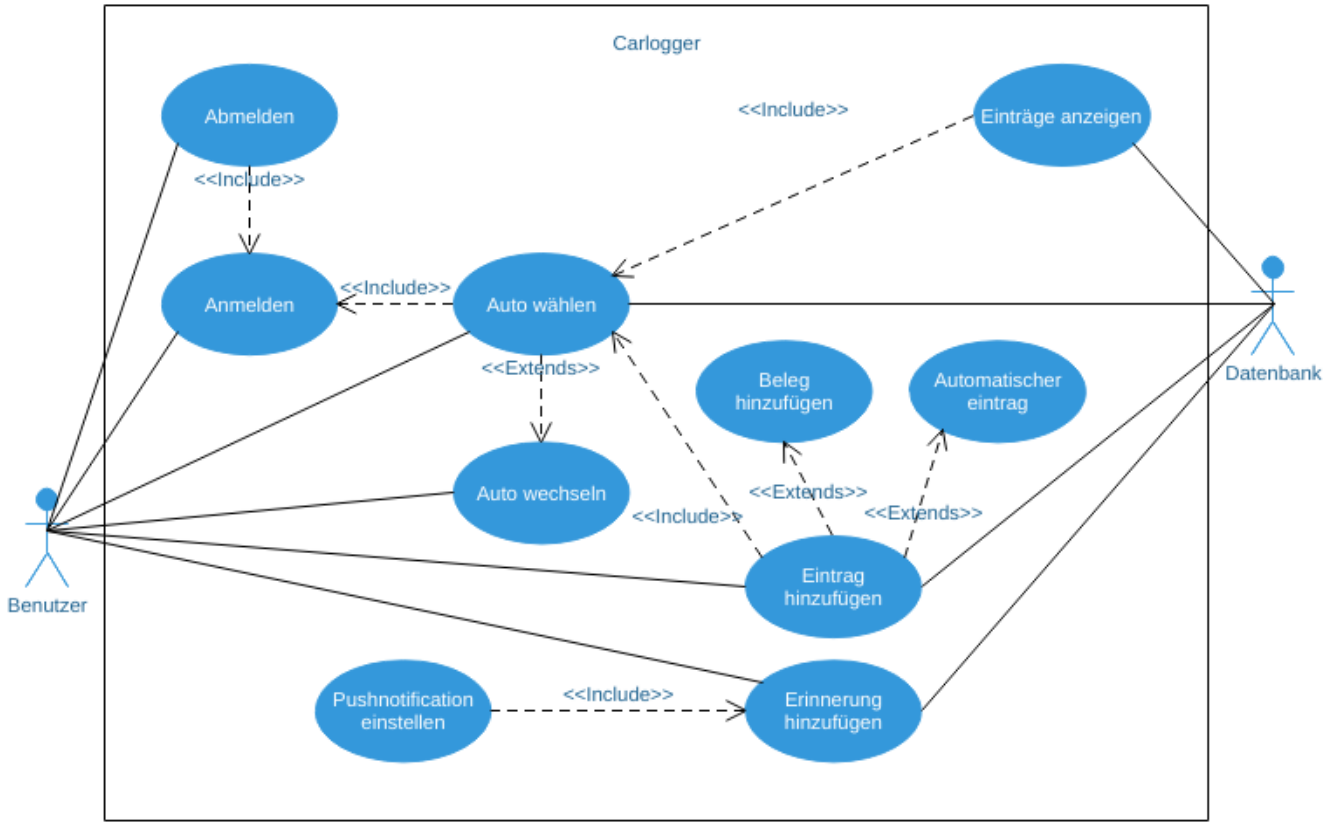
Der Benutzer kann neue Einträge zum gewählten Fahrzeug hinzufügen. Die Daten werden beim Eintragen in die Datenbank geschrieben. Die Einträge können mit zusätzlichen Informationen erweitert werden. Ein Fotobeleg der Rechnung kann mitgespeichert werden.

Ein Eintrag kann in einem bestimmten Zeitintervall wiederholt automatisch eingetragen werden.

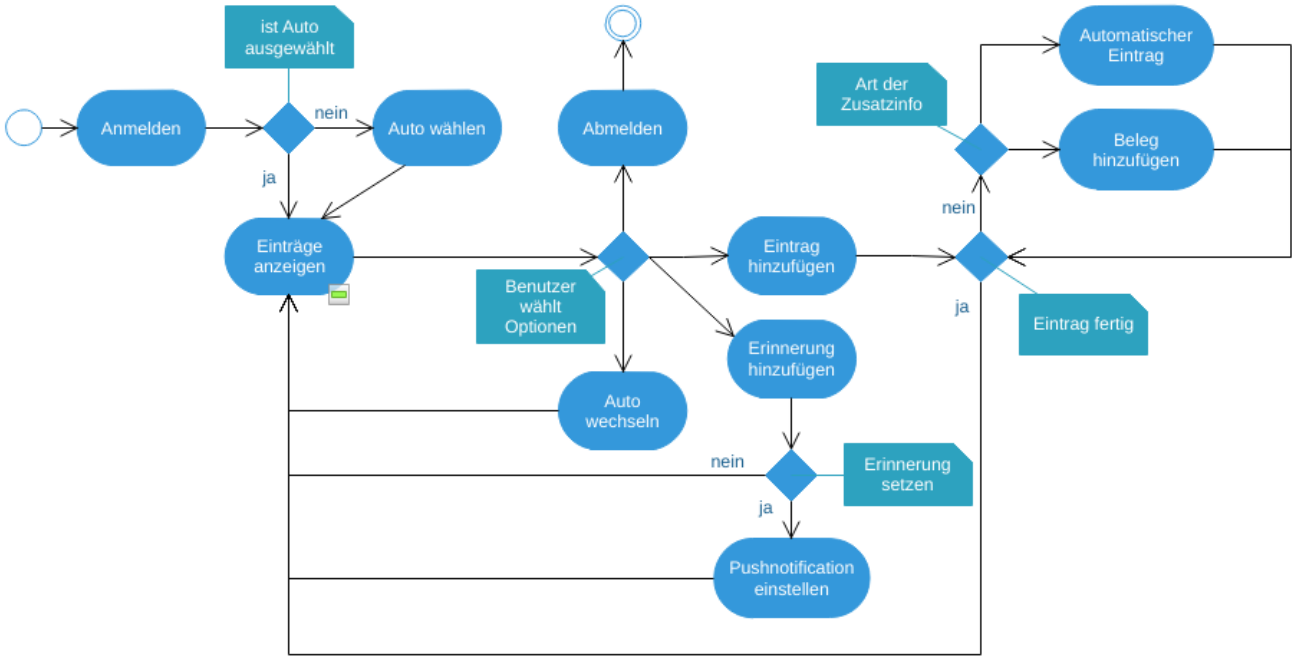
„Erinnerung hinzufügen“

Der Benutzer wählt aus dem Kalender ein Datum für seinen Termin aus und fügt Beschreibung und Uhrzeit hinzu. Optional kann eine Push-Notification hinzugefügt werden.

Use-Case Diagramm

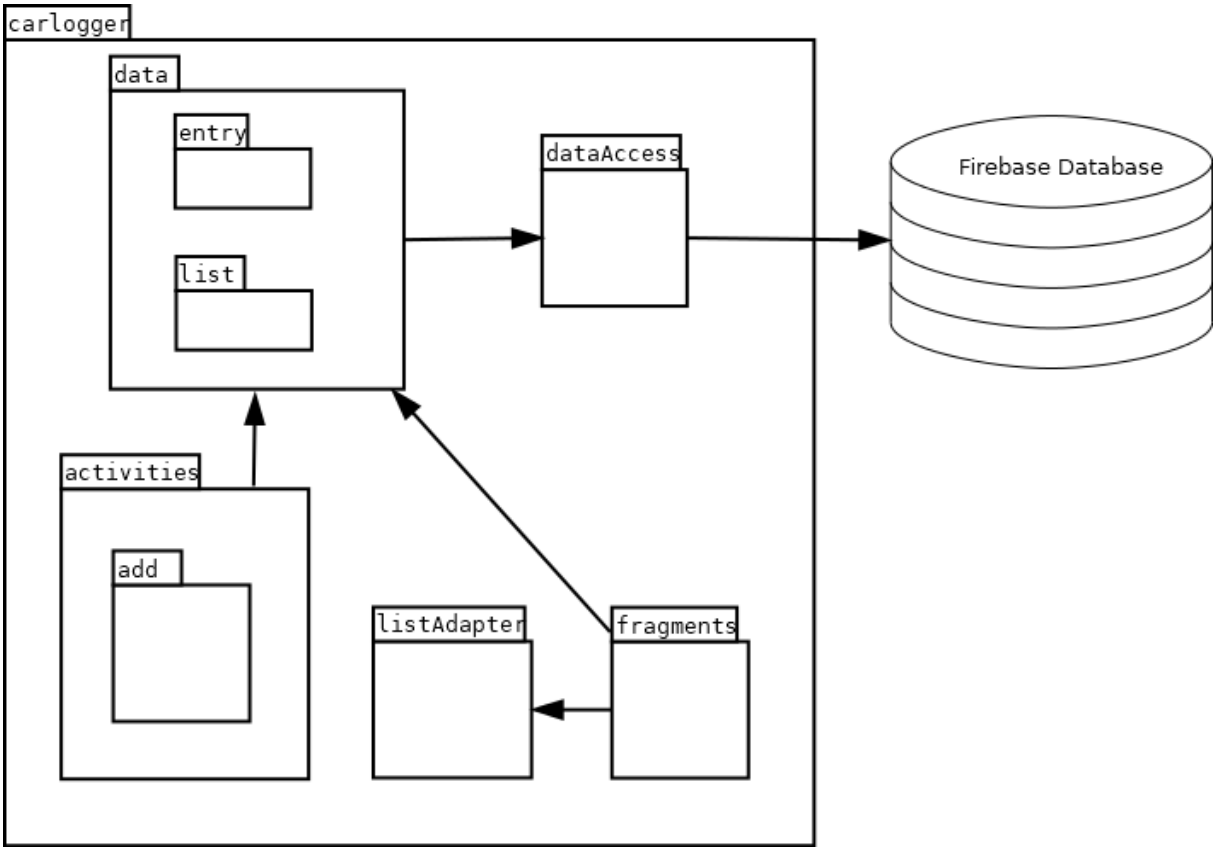


Aktivitätsdiagramm



Architektur

Zu sehen ist eine Übersicht über die Pakete, Verwendungen untereinander und die Anbindung der von *Firebase* zur Verfügung gestellten Real-Time Datenbank.



Activities werden genutzt um Einträge hinzuzufügen, *Fragments* stellen die Ansicht der Einträge sowie die errechneten Werte dar, wobei jeder Eintrag-Typ seinen eigenen Adapter für die Listen-Darstellung nutzt.

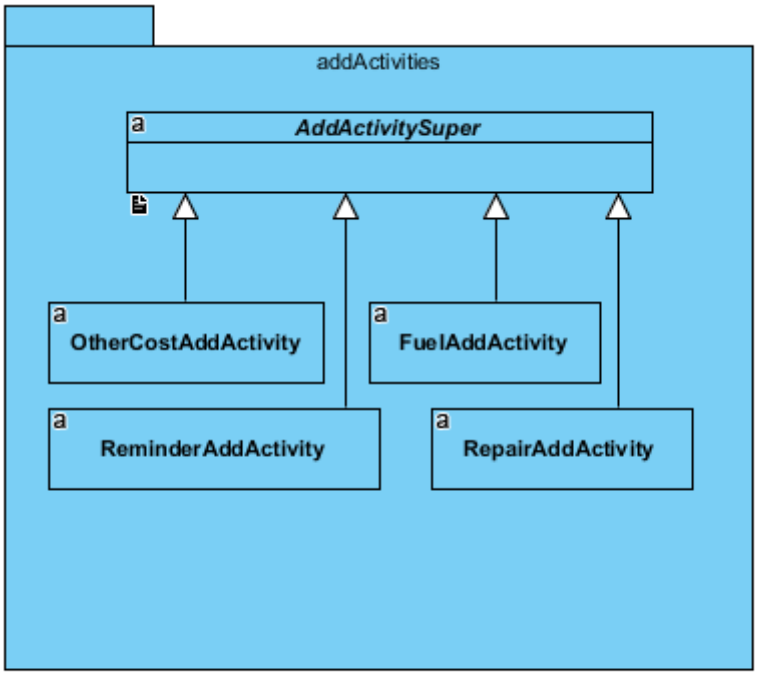
Die Anzeige der Durchschnitts-Werte und Gesamtkosten erfordert den Zugriff der *Fragments* auf die *Data*-Klassen.

Um Einträge in den *Fragments* anzeigen zu können werden Adapter genutzt, welche die Einträge in einer Listen-Ansicht darstellen. Die Adapter, welche für jede Eintragskategorie definiert sind, erhalten Zugriff auf die Datenstruktur über die einzelnen Fragmente.

Die Klassen im *Data*-Paket werden verwendet um die Eintrags-Daten mittels einer internen Datenstruktur abzubilden.

Das *dataAccess* Paket stellt eine Schnittstelle zu *Firebase*-Datenbank dar und ermöglicht die Kommunikation mit dieser. Alle Funktionen für das Holen der Daten und Speichern der Daten sind hier implementiert.

addActivities



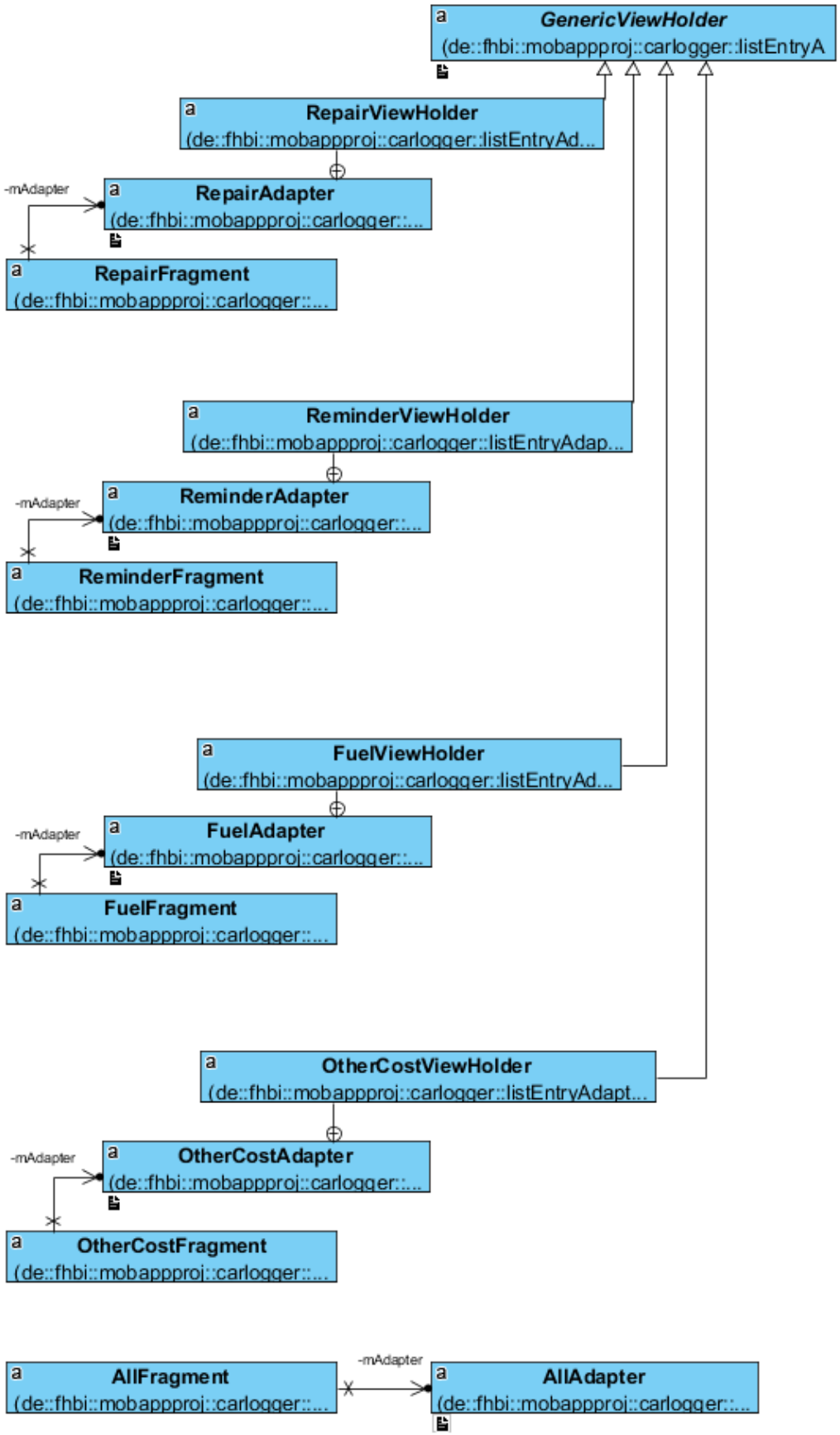
Die *AddActivities* ermöglichen die Eingabe von Datensätzen durch den Benutzer.

Die abstrakte Klasse *AddActivitySuper* wird als Vorlage der *AddActivity* Klassen genutzt. Hier werden abstrakte Methoden deklariert, welche von den Sub-Klassen definiert werden müssen, sowie gemeinsame Verhaltensweisen implementiert.

Beispiele für Verhaltensweisen die für alle *AddActivities* gelten sind:

- Zurück-*Button* in der *Action-Bar*, welcher die vorherige *Activity* öffnet
- Schließen der Tastatur, wenn der Fokus eines *EditText*-Elements verlassen wird

Fragments und ListAdapter



Um die Einträge und Statistiken anzuzeigen werden *Fragments* genutzt.

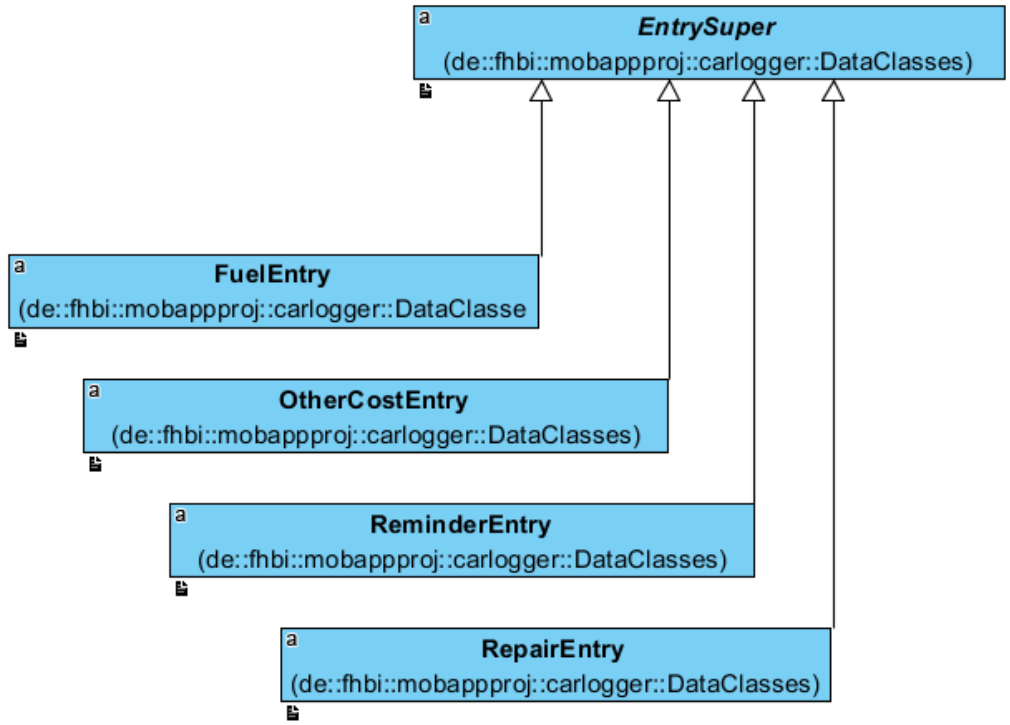
Jede Kategorie von Einträgen nutzt dafür ein eigenes *Fragment.*

Jedes Fragment hat eine *RecyclerView*, die über einen Spezifizierten Adapter eine Liste von Einträgen verwaltet. Ein Eintrag wird mit Hilfe eines *ViewHolders* in dem *Fragment* dargestellt. Der *ViewHolder* setzt dabei die einzelnen Text-Felder der Listen-Einträge und definiert die *onClick*-Methoden der *Buttons*, über die jeder Eintrag verfügt (Bearbeiten, Löschen). Außerdem wird hier geprüft, ob der Eintrag angeklickt wurde und dementsprechend mit Erweiterung oder Verminderung der Anzeige reagiert.

Der *GenericViewHolder* wird benötigt um dem Übersichts-*Fragment*, indem eine Liste mit allen Datensätzen zu sehen ist, eine Vorlage zu liefern mit der es möglich ist Einträge mit unterschiedlichen Layouts in einer einzigen Liste anzuzeigen. Aus diesem Grund erweitern alle *ViewHolder* den *GenericViewHolder*.

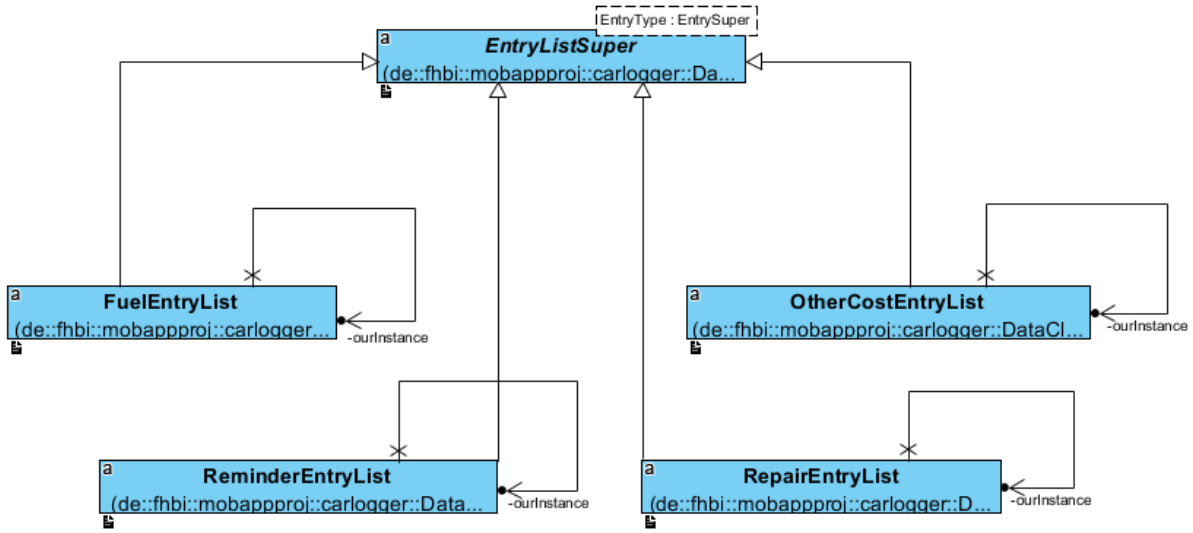
Die Adapter nutzen eine Eintrags-Liste und legen fest welches *Layout* für den Eintrags-Typen verwendet wird.

data.entry



Die verschiedenen Einträge verfügen über unterschiedliche Operationen und Attribute. Gemeinsame Verhaltensweisen und Daten-Felder werden in der abstrakten Klasse *EntrySuper* beschrieben.

data.list



Für jede Eintrags-Kategorie gibt es eine Liste in der die Einträge gehalten werden. Diese Eintrags-Listen benutzen das *Singleton-Design-Pattern* um in der gesamten App nur eine Instanz der jeweiligen Liste zu benutzen. Die generische *Super*-Klasse implementiert Operationen die beispielsweise die Gesamtkosten oder die Kosten pro Monat ermitteln.

Wird eine Instanz einer *Entry*-Klasse erzeugt, wird sie automatisch der zugehörigen *EntryList* hinzugefügt. Die Listen dienen lediglich der lokalen Verwaltung der Daten. Jede *Entry*-Klasse implementiert die Methoden:

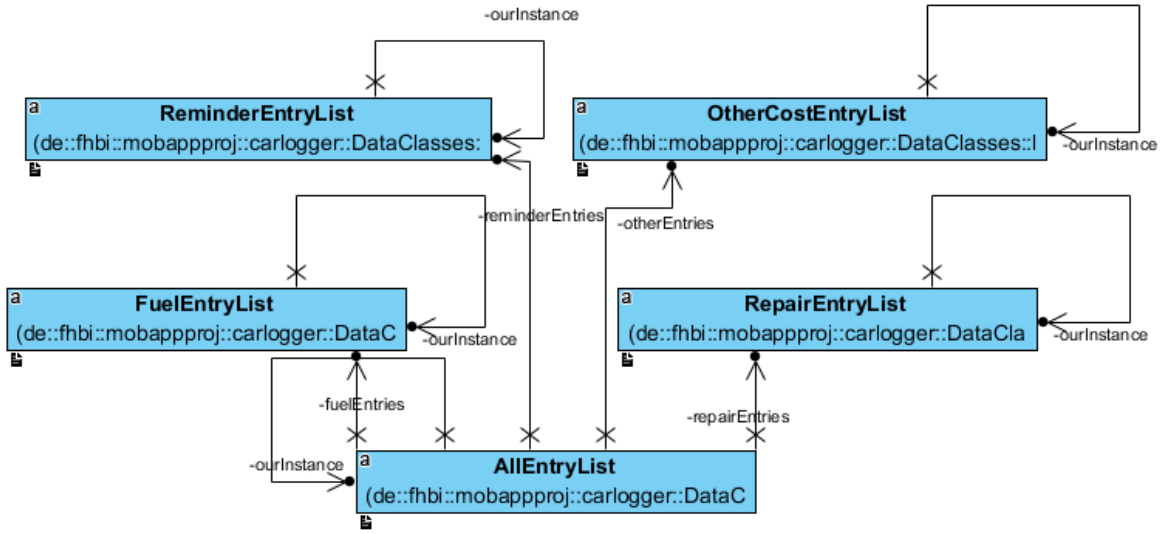
pushToFirebase()

removeFromFirebase()

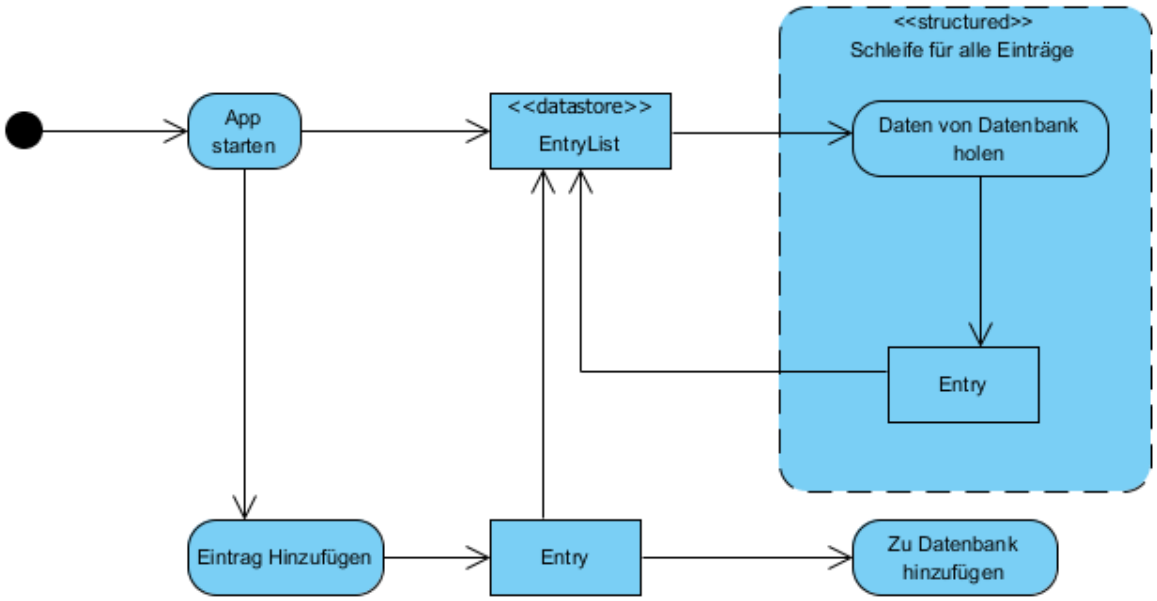
updateChangesOnFirebase()

mit diesen Methoden werden die Daten des einzelnen Eintrages in die Firebase-Datenbank eingetragen, geändert oder gelöscht.

Um die Daten bei App-Start von der *Firebase*-Datenbank zu holen werden in den List-Klassen Methoden definiert, welche die Daten von der Online-Datenbank holen und diese in die passenden *Entry*-Objekte parsen.



Die Klasse *AllEntryList* verfügt über alle Eintrags-Listen und enthält eine eigene Liste in der alle Einträge enthalten sind. Diese Liste ermöglicht Operationen in denen alle Datensätze berücksichtigt werden müssen.



Die Abbildung veranschaulicht die Interaktion mit der Datenbank. Die Daten werden bei App-Start über die *EntryListen* von der Datenbank geholt, es werden Instanzen der *Entry*-Klassen erzeugt, welche automatisch den *EntryListen* hinzugefügt werden.

Wird ein Eintrag erstellt, so wird er direkt in die Datenbank eingepflegt und der lokalen *EntryList* hinzugefügt.

Implementierung

EntryListSuper

Die Super Klasse aller Eintrags-Klassen verfügt über drei Methoden die der Errechnung von Werten wie Gesamtkosten oder monatliche Kosten dienen. Die Methoden können von jeder Eintrags-Klasse verwendet werden, die Berechnung ist immer dieselbe.

getAllCosts(): ermittelt die Summe aller Kosten indem alle Einträge mittels einer Schleife durchlaufen werden.

In der Methode getCostPerMonth() wird der Eintrag mit dem niedrigsten Datum gewählt, dieser wird von dem heutigen Datum subtrahiert. Der resultierende Wert in Monaten ergibt die Zeitspanne, in der Einträge getätigt wurden. Die Summe aller Kosten wird durch die Anzahl der Monate geteilt und ergibt die Kosten pro Monat.

```
public double getCostPerMonth(Calendar today){
    double cost = 0;

    if(!allEntries.isEmpty()) {
        Calendar first = Collections.min(allEntries).getCreateTimeCalendar();


        int diffYear = today.get(Calendar.YEAR) - first.get(Calendar.YEAR);
        int diffMonth = diffYear * 12 + today.get(Calendar.MONTH) -
first.get(Calendar.MONTH);

        cost = getAllCosts() / ((diffMonth == 0) ? 1 : diffMonth);
    }
    return cost;
}
```

getCostTime() liefert die Kosten über einen, vom Benutzer, angegebenen Zeitraum. Hier werden die Erstellungs-Zeitpunkte miteinander verglichen. Liegt der Eintrag in dem angegebenen Zeitfenster, wird er in der Summe aufgenommen.

```
public double getCostTime(Calendar start, Calendar end) {
    double cost = 0;

    Collections.sort(allEntries);

    for(EntryType entry : allEntries){
        if(entry.getCreateTimeCalendar().compareTo(start) >= 0 &&
entry.getCreateTimeCalendar().compareTo(end) <= 0){
            cost += entry.getCost();
        }
    }

    return cost;
}
```

um die Einträge mittels Erstellungs-Datum miteinander vergleichen zu können wird die *compareTo()* Methode in den Eintrags-Klassen überschrieben. Hier wird anhand des Erstellungsdatums geprüft in welcher Reihenfolge die Einträge erstellt wurden.

```
@Override
public int compareTo(@NonNull EntrySuper entrySuper) {
    long thisTime = this.createTimeCalendar.getTimeInMillis();
    long anotherTime = entrySuper.createTimeCalendar.getTimeInMillis();
    return (thisTime<anotherTime ? -1 : (thisTime==anotherTime ? 0 : 1));
}
```

Die Klasse *FuelEntryList* erweitert die Klasse *EntryListSuper* um eine Methode die den Durchschnittsverbrauch des Fahrzeuges errechnet. Voraussetzung für eine realistische Berechnung ist, dass zwei aufeinanderfolgende Einträge als *Full* markiert wurden, denn nur durch das Volltanken zweimal hintereinander ist eine Berechnung des Durchschnittsverbrauchs möglich. Hierfür werden die gefahrenen Kilometer ermittelt indem die beiden Kilometerstände der Einträge subtrahiert werden und die verbrauchten Liter werden über die Menge des beim zweiten Tankvorgang hinzugefügten Kraftstoffes definiert.

Für einen präzisen Durchschnittswert wird der Durchschnitt aller berechneten Eintrags-Paare gebildet.

Andere Möglichkeiten der Durchschnittsverbrauch Berechnung erfordern eine größere Datenmenge für eine genaue Berechnung (siehe Usability)

```
public double getConsumtion(){
    double consumption = 0;
    int counter = 0;

    ArrayList<Double> conList = new ArrayList<Double>();

    Collections.sort(allEntries);

    int tempIndex = -1;

    for(FuelEntry entry : allEntries){
        if(entry.isFull()){
            //first Entry with Full
            tempIndex = allEntries.indexOf(entry);
            if(allEntries.size()-1 > tempIndex){
                //next Entry is Full
                if(allEntries.get(tempIndex+1).isFull()){
                    double diffKM = allEntries.get(tempIndex+1).getKm() -
allEntries.get(tempIndex).getKm();
                    double diffLitre = allEntries.get(tempIndex + 1).getAmount();

                    if(diffKM > 0) {
                        consumption += diffLitre * (100 / diffKM);
                        counter++;
                    }

                }
            }
        }
    }
    if(counter > 0){
        consumption = consumption / counter;
    }

    return consumption;
}
```

AutoEntry

Automatische Einträge werden realisiert indem jedes Eintrags-Objekt ein Attribut des *Enums* *AutoEntry* besitzt, welches null ist, wenn es kein automatischer Eintrag ist.

```
public enum AutoEntry {
    DAILY , WEEKLY, MONTHLY, YEARLY, EVERYTWOMONTH, EVERYTHREEMONTH, EVERYTWOYEAR;
}
```

Dieses *Enum* befindet sich in der Klasse *AutoEntryDates*.
Zusätzlich befinden sich Konstanten in der Klasse die den Enum-Feldern Werte zuordnen, welche jeweils dem Eintrags-Intervall in Tagen entsprechen.

```
private static final int DAILYINT = 1, WEEKLYINT = 7, MONTHLYINT = 30, YEARLYINT =
365, EVERYTWOMONTH = 60, EVERYTHREEMONTH = 90, EVERYTWOYEAR = 730;
```

getIntervallList() berechnet eine Liste vom Typ *Calendar*, welche alle Daten enthält an denen ein Eintrag hinzugefügt werden soll.

Als Parameter werden das Datum des Zuletzt hinzugefügten Eintrages und das Intervall in Tagen angegeben.

Die Differenz des aktuellen Zeitpunktes und des Zeitpunktes der Erstellung des letzten Eintrages wird berechnet und als Wert in Tagen genutzt.

Eine Schleife erzeugt Daten alle X Tage.

```
//intervall in days
private static ArrayList<Calendar> getIntervallList(Calendar lastEntryCalendar, int
intervall) {
    ArrayList<Calendar> list = new ArrayList<Calendar>();
    Calendar cur = Calendar.getInstance();

    long end = cur.getTimeInMillis();
    long start = lastEntryCalendar.getTimeInMillis();
    long days = TimeUnit.MILLISECONDS.toDays(Math.abs(end - start));

    for(int i = 1; i <=days; i+=intervall){
        Calendar newCal = (Calendar) lastEntryCalendar.clone();
        newCal.add(Calendar.DAY_OF_MONTH,i);
        list.add(newCal);
    }
    return list;
}
```

Die Methode *getIntervallList()* wird aufgerufen, wenn alle Einträge von der *Firebase*-Datenbank geholt werden. Es werden zu jedem von *getIntervallList()* gelieferten Datensatz jeweils ein Eintrag des Eintrags-Typen erstellt und in die *Firebase* hochgeladen.

```
public void setAutoEntries() {
    //list of Entries with attribute last == true
    ArrayList<EntrySuper> lastAutoEntries = new ArrayList<>();
    //fill list
    for (Object o : allEntries) {
        EntrySuper entry = (EntrySuper) o;
        if (entry.isLastEntry() && entry.getAutoEntry() != null) {
            lastAutoEntries.add(entry);
        }
    }
    //loop through all Entries with last == true
    for (EntrySuper entry : lastAutoEntries) {
        ArrayList<Calendar> calList =
AutoEntryDates.getList(entry.getCreateTimeCalendar(), entry.getAutoEntry());

        if(calList.size() > 0){
            // not anymore last
            entry.setLastEntry(false);
            entry.updateChangesOnFirebase();
        }
        //create Entry for each Date
        for (int i = 0; i < calList.size(); i++) {

            switch (entry.getEntryType()) {
                case FUELENTRY:
                    FuelEntry fe = (FuelEntry) entry;
                    FuelEntry newFe = new FuelEntry(fe);
                    newFe.setLastEntry(i == calList.size() - 1);
                    newFe.editCreateTimeCalendar(calList.get(i));
                    newFe.push();
                    break;

                case REPAIRENTRY:
                    RepairEntry re = (RepairEntry) entry;
                    RepairEntry newRe = new RepairEntry(re);
                    newRe.setLastEntry(i == calList.size() - 1);
                    newRe.editCreateTimeCalendar(calList.get(i));
                    newRe.push();
                    break;


                case OTHERCOSTENTRY:
                    OtherCostEntry oe = (OtherCostEntry) entry;
                    OtherCostEntry newOe = new OtherCostEntry(oe);
                    newOe.setLastEntry(i == calList.size() - 1);
                    newOe.editCreateTimeCalendar(calList.get(i));
                    newOe.push();
                    break;

            }

        }
    }
}
```

ReminderNotification

Die App bietet die Möglichkeit den Benutzer über eine *Notification* an einen anstehenden Termin zu erinnern.

Diese Funktion wird durch den System-Dienst *ALARM_SERVICE* ermöglicht.

Ein *AlarmManager* sorgt dafür, dass zu der eingestellten Zeit ein *PendingIntent* ausgeführt wird. Dieser *Intent* wird von einem *AlarmReceiver* empfangen, welcher die *Notification* auslöst.

Durch die Nutzung eines *PendingIntents* ist es möglich die *Notification* auch auszulösen, wenn die App nicht gestartet wurde.

Da eingestellte *AlarmManager* nach System-Neustart nicht mehr existieren, ist es notwendig den *BOOT_COMPLETED Intent* abzufangen.

Nach dem vollständigen Boot-Vorgang werden erneut *AlarmManager* für alle Erinnerungen mit dem Attribut *PushNotification* gesetzt.

Autoauswahl

Ein elementarer Bestandteil der Anwendung ist die Autoauswahl. Ein Nutzer soll dort das Modell seines Fahrzeugs wiederfinden können, um seine Log-Einträge eindeutig diesem zuordnen zu können.

Um dem Nutzer eine Listenansicht dieser Fahrzeuge anbieten zu können, welche sich unter verschiedenen Bedingungen ändern kann wurden unterschiedliche Vorgehensweisen getestet.

Es wurde eine Textdatei im JSON Format im res/raw Ordner der Applikation bereitgestellt, welche etwa 27.000 verschiedene Autoeinträge enthält. Aus dieser kann mit der GSON-Bibliothek eine Liste von Java-Objekten erzeugt werden, welche wiederrum durch die Implementierung eines ArrayAdapters diese auf einer ListView darstellen kann. Um die Einträge dem Nutzer möglichst Übersichtlich darstellen zu können, wurde hierfür ein neues Layout im XML-Format erzeugt, welches die einzelnen Views und Viewgroups beschreibt, in denen die Informationen der Fahrzeuge dargestellt werden.

Nach zunächst erfolgreichem Testen wurde, trotz vollständiger und korrekter Darstellung der Liste, diese Vorgehensweise wieder verworfen, da mehrere Probleme auftraten. Beim erstmaligen erzeugen der Liste erfordert dies einen recht hohen Rechenaufwand, wodurch die App für etwa zwei Sekunden nicht antwortet. Dies geschah auch bei Ausführung der Aufgabe in einem zweiten Thread, in einem Handler oder in einem AsyncTask. Weiterhin sollte die Menge der angezeigten Fahrzeuge von dem Nutzer über eine Suchfunktion zu filtern sein. Um dies mit einem ArrayAdapter zu ermöglichen, muss bei jedem Tastendruck des Nutzers eine neue Liste erzeugt werden, ausschließlich mit den anzuzeigenden Fahrzeugen. Da hierfür jeder Eintrag der Liste auf den vom Nutzer eingegebenen Filter untersucht werden muss, stellte sich dies als zusätzlich inperformant heraus.

Um diesen Problemen entgegenzutreten wird in der App nun auf ArrayAdapter und GSON einschließlich der json-Datei verzichtet. Stattdessen befindet sich nun in dem res/raw eine sqlite Datenbank, welche bei Start der App in den für die Anwendung bereitgestellten databases Ordner kopiert wird. Dieser befindet sich in /data und ist von einem normalen Nutzer ohne root-Rechte nicht einsehbar und durch ständiges Überschreiben bei Appstart wird gewährleistet, dass auch bei Änderung der bereitgestellten Datenbank bei Aktualisierung der App durch den PlayStore die Datenbank auf dem Gerät auf dem aktuellen Stand ist.

Es muss kein weiterer SQLite Client implementiert werden, da dieser schon in den Standardbibliotheken von Android enthalten ist, im package *android.database.sqlite.* So kann mit einer einfachen Erweiterung der Klasse SQLiteOpenHelper der Zugriff auf die Datenbank ermöglicht werden. Bei Ausführen einer Anfrage auf dem vom Helper erzeugten SQLiteDatabase Objekt liefert dieses uns ein Cursorobjekt zurück, welches das Ergebnis dieser Anfrage repräsentiert. Die wichtigsten Anfragen die wir für unsere Liste benötigen, ist die Abfrage nach allen Daten (*SELECT **

FROM cars), sowie zusätzlich die Möglichkeit nach einem Feld zu filtern (*WHERE field LIKE* Statement).

Wie bereits erwähnt wird kein ArrayAdapter verwendet, welcher auf die Verwendung einer Liste von Objekten im Speicher besteht, sondern der Cursor wird mit einem Cursoradapter an die ListView gebunden. Hierfür wurde ein eine Klasse geschrieben, welche den Cursoradapter erweitert und beschreibt wie jedes Ergebnis der Datenbankabfrage, welches sich im Cursor befindet, auf einem View dargestellt wird.

Hierfür wird wieder das vorher erwähnte Layout von dem Cursoradapter verwendet, um einen View zu erzeugen und den Inhalt der Spalten der Datenbanktabelle den TextViews zuzuordnen und zu befüllen.

Durch das Wegfallen der aufwändigen Listen können so nun ohne große Verzögerungen auf Anfragen vom Nutzer reagiert werden. So wird in der QueryTextChange()-Methode der implementierten Suche, welche aufgerufen wird wenn der Nutzer den Text in der Sucheingabe ändert, eine neue Anfrage an die Datenbank gestellt, der Cursor des Adapters ausgetauscht und die ListView mit dem Adapter aktualisiert.

LoginActivity

Um dem Nutzer eine einfache Möglichkeit zu bieten seine Daten über mehrere Geräte zu synchronisieren, kann dieser sich durch sein Google Konto authentifizieren. Hierfür wurde eine weitere Activity erzeugt, welche in der Manifestdatei mit dem theme *AppCompat.Light.Dialog* versehen wurde. So erhält eine vollwertige Activity die Vorteile eines Dialogs, wie etwa die Möglichkeit die Activity abzubrechen durch tippen auf den Hintergrund.

Die Activity baut das Layout mit einem Knopf auf, welcher mit der Funktion versehen ist, einen mit den Google Bibliotheken erzeugten SignIn-Intent abzuschicken. Wenn der Nutzer einen Account gewählt oder den Vorgang abgebrochen hat, wird die Funktion onActivityResult mit dem Ergebnis aufgerufen. Diese wiederum informiert die aufrufende Aktivität und beendet sich selber. In der Zukunft ist es sehr einfach in diese Aktivität weitere Authentifizierungsmöglichkeiten einzufügen.

MainActivity

Die Aktivität welche bei Start der Anwendung gestartet wird, deckt viele wichtige Aufgabenbereich ab. Die Navigationsleiste auf der linken Seite wird erzeugt und ein Listener implementiert, welcher bei Auswahl eines Eintrags aufgerufen wird. Dieser unterscheidet zwischen den Menüs und dem Login/Logout Eintrag. Wenn der Nutzer das Menü wechseln möchte, wird der Inhalt der Hauptaktivität durch ein passendes Fragment ersetzt, entsprechend der Auswahl. Der Login/Logout Button öffnet die bereits beschriebene LoginActivity, sollte der Nutzer nicht eingeloggt sein oder loggt den Nutzer aus. Da auch die Nutzung der App ohne eine Authentifizierung mit Google zugelassen werden sollte, wird der Nutzer nicht nur ausgeloggt, sondern authentifiziert sich mit einer einmaligen User-ID, welche beim Einloggen in der App an den Google-Account gebunden wird. Die anonyme Authentifizierung findet auch beim Start der App statt, sollte kein Nutzer eingeloggt sein, anonym oder über Google.

Test

Methoden, die logische Berechnungen enthalten, wurden mithilfe von JUnit Testklassen getestet. Hierfür wurden Einträge in den *setUp()* Methoden definiert und die zu Testenden Operationen getestet.

EntryListTest.java

Die Entry-Listen werden getestet indem Einträge hinzugefügt werden, diese werden bei der Erzeugung durch ihren Konstruktor-Aufruf der jeweiligen Liste hinzugefügt.

Wir testen, ob alle Einträge hinzugefügt wurden und ob die Zuordnung der Einträge richtig funktioniert.

EntryListSuperTest.java

Die Super-Klasse aller Listen enthält Operationen über die Listen, wie ermitteln der Gesamtkosten etc.

Hier werden ebenfalls Einträge aller Kategorien mit verschiedenen Werten erzeugt.

Um die *costPerMonth()* Methode testen zu können werden die *createTimeCalendar,* welche das Datum und die Uhrzeit der Erzeugung halten, überschrieben.

Es werden die Methoden zur Berechnung der Gesamtkosten, monatlichen Kosten und Kosten für einen Zeitraum in allen Listen getestet.

AutoEntryDatesTest.java

Um die statische Methode *getList()* der *AutoEntryDates*-Klasse zu testen wird ein *Calendar*-Objekt erzeugt und das Datum überschrieben.

Die Methode soll eine Liste von Daten erzeugen indem man Ihr ein Datum und ein Typ des *Enum´s* AutoEntry übergibt.

Diese Liste soll verwendet werden um Einträge automatisch zu erstellen.

Es werden die verschiedenen Intervalle getestet und geprüft, ob die Anzahl der Daten korrekt ist und die Daten richtig gesetzt wurden.

AutoEntryTest.java

In dieser Test-Klasse wird *setAutoEntries()* getestet. Die Methode wird genutzt um Einträge anhand der Liste von Daten von *getList()* zu erzeugen. Diese Test-Klasse baut auf die Test-Klasse *AutoEntryDates* auf und prüft, ob die Einträge korrekt erzeugt und die Daten fehlerfrei übernommene wurden.

Graphische Elemente wurden ausführlich manuell per Hand getestet, wobei versucht wurde möglichst alle möglichen Benutzer-Eingaben und Verhalten zu simulieren. Zusätzlich wurde die App an einen kleinen Kreis bestehend aus Familienmitgliedern und Freunden weitergegeben mit dem Auftrag alle Funktionen auf Fehlverhalten zu testen.

Usability

Um eine gute Benutzbarkeit zu gewährleisten wurde versucht gängige GUI-Elemente zu verwenden die dem Benutzer einen angenehmen und vertrauten Workflow zu ermöglichen. Zudem wurde die App in Ihrem Funktionsumfang so einfach wie möglich gehalten. Ziel war es dem Benutzer eine App zur Verfügung zu stellen die ein einfaches Loggen der Kosten rund um das Fahrzeug ermöglicht.

Das Login/Anmelden gestaltet sich sehr einfach da der Nutzer sein Vorhandenes Gmail-Konto verwendet und nur noch ein Passwort vergeben muss. Dieses Passwort bleibt erhalten und muss nicht mehr erneut eingegeben werden. Ist der User nicht registriert, wird ihm ein anonymes Konto zu Verfügung gestellt, womit seine Daten in der Datenbank identifiziert werden können. Eine Verwendung der Daten auf einem anderen Gerät ist erst nach Registrierung des Gmail Accounts möglich. Wird zuerst ein anonymes Konto genutzt, werden die Nutzer-Daten bei Registrierung mit in den neuen Account übernommen.

Das *Navigation-Drawer-Panel* wurde gewählt da dieses einen hohen Wiedererkennungswert besitzt und es einfach zu bedienen ist.

Mit dem Panel kann man zu jeder Zeit in die anderen Kategorien wechseln oder sich das Start-Fragment mit einer Übersicht über alle Kosten anzeigen lassen.

In jeder Kategorie werden die zu dieser Kategorie gehörigen statistischen Werte ausgegeben. Das Feld *Kosten für Zeitraum* besitzt einen Button mit dem sich ein *Date-Picker* in einem *Alert-Dialog* öffnet. Dieser *Date-Picker* enthält zwei *Tabs: Von, bis* um den Zeitraum zu wählen für dem die Kosten berechnet werden sollen.

Bei der Auswahl des Start- und End-Datums ist es unerheblich welches zuerst gewählt wird, da der kleinere Wert als Start-Datum übergeben wird, End-Datum kann also auch unter dem Tab: Von gewählt werden.

Einträge werden hinzugefügt indem der *Floating-Action-Button* mit dem Plus Symbol in der unteren rechten Ecke gedrückt wird. Hier öffnet sich eine Leiste zur Auswahl der Eintrags-Kategorie.

Jede Eintrags-Kategorie verfügt über eine eigene *Activity* mit einer Eingabemaske für relevante Informationen. Die eingegebenen Daten werden auf Gültigkeit geprüft.

Für die Auswahl der Zeit-Intervalle der automatischen Einträge öffnet sich ein *Alert-Dialog* um ein komplexes Erscheinungsbild der Eingabemaske zu verhindern.

Die Einträge werden in Listen angezeigt. Sichtbar sind nur die wichtigsten Informationen wie Datum der Erstellung und Preis. Zusätzlich wird jeder Eintrag mit einem Symbol gekennzeichnet um ihn einer Kategorie zuzuordnen. Dies ist vor allem in dem *Übersicht-Fragment* vonnöten, da hier alle Einträge in einer Liste sind.

Erst bei Berührung des Eintrages werden die restlichen Informationen, sowie ein Button für Änderungen und ein Button zur Löschung des Eintrages, sichtbar.

Änderungen erfolgen über dieselbe *Activity* wie schon das Hinzufügen, die Eingabemaske ist hier schon mit den Werten des Eintrages gefüllt.

Diese Vorgehensweise lässt die App einfacher wirken.

Zur Berechnung des Durchschnittsverbrauches sind verschiedene Strategien möglich. Zum einen kann der Durchschnitt ermittelt werden indem die gefahrenen KM mit der Menge in Liter aller Tankvorgänge dividiert werden und anschließend mit dem Faktor 100 versehen (L/100KM). Diese Vorgehensweise erfordert jedoch ein kontinuierliches und präzises Loggen für eine hohe Genauigkeit. Mit dieser Methode werden außerdem automatische Einträgen im Tanken Bereich nicht optimal erfasst und die Genauigkeit des Wertes ist abhängig von der Menge und Konsistenz der Daten.

Um hier eine Möglichst einfache Handhabung zu gewährleisten wird der Algorithmus wie er in *Implementierung* beschrieben wird verwendet. Dieser Algorithmus erfordert das Eintragen von mindestens zwei Tankvorgängen mit vollem Tank hintereinander um einen präzisen Wert zu ermitteln. Auf diesen Umstand wird der Benutzer in der Anzeige des Durchschnittsverbrauches hingewiesen.

Die *AutoEntry*-Funktion soll für eine bessere Benutzbarkeit sorgen. Einige Einträge werden sich voraussichtlich regelmäßig wiederholen. Beispiele für solche Einträge währen TÜV-Inspektion oder Service-Inspektion in der Werkstatt. Diese Einträge haben oft in etwa dieselben Kosten und eignen sich hiermit für einen *AutoEntry*.

Durch diese Funktion ist der Benutzter nicht gezwungen jedes Ereignis von Hand einzutragen. Die Tank-Einträge können auch automatisiert eingetragen werden, jedoch lässt hier die Genauigkeit der Statistiken nach, wenn der Nutzer ein Unregelmäßiges Fahrverhalten hat oder die Kraftstoffpreise stark schwanken.

Zusammenfassung

Über den Zeitraum des Projekts wurden die Mitglieder mit zahlreichen unterschiedlichen Aufgaben und Problemen konfrontiert, welche sie bewältigen mussten. Hierfür wurden die Aufgabenbereiche auf die Teammitglieder verteilt, jedoch stellte sich heraus, dass alle Mitglieder ein gutes Verständnis über die wichtigsten Teile der Android-SDK benötigten. So mussten unter anderem die für die Oberfläche und Logik zuständigen Personen häufig gemeinsam an einem Problem arbeiten, wenn dies eine gute Kenntnis über verschiedene Komponenten der gegebenen Bibliotheken und der entwickelten Struktur erforderte. Dazu zählen unter anderem die Performanceprobleme, welche sich des Öfteren in unerwarteten Situationen offenbarten. Zusätzlich musste die zu Beginn abgesprochene Struktur der Daten wiederholt überarbeitet werden, sobald festgestellt wurde, dass diese nicht oder nicht optimal in der App implementiert werden konnte.

Durch die in diesem Projekt erlangten Fähigkeiten sind die Mitglieder in der Lage in Zukunft weitere mobile Applikationen mit einer modernen grafischen Benutzeroberfläche für das Android System zu entwickeln. Dies betrifft jedoch nicht die Entwicklung von umfassenden Spielen oder aufwändigen grafischen Animationen, welche im Zeitraum dieses Projekts keine Aufmerksamkeit erhalten haben.