**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.

Die Einträge werden in 4 verschiedene Kategorien unterteilt, welche durch folgende Attribute definiert werden:

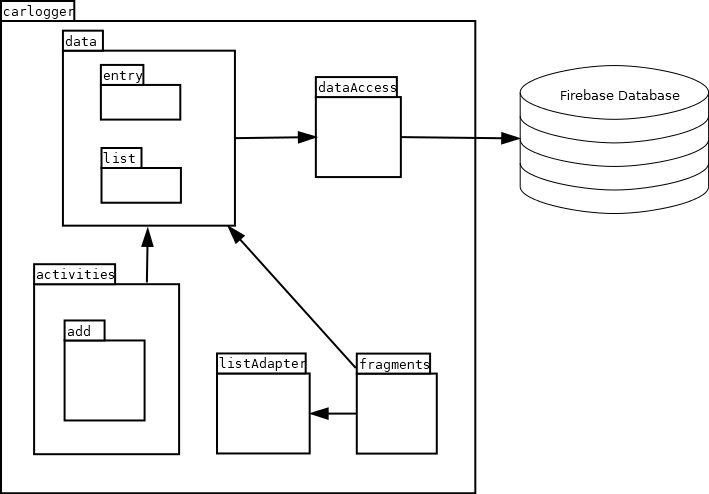
1. Tanken
   1. Menge in Liter
   2. Kosten/Liter
   3. Kilometerstand
   4. Vollgetankt?
   5. Automatischer Eintrag
2. Reparaturen und Service
   1. Art
      1. Reparatur (Werkstatt)
      2. Reparatur
      3. TÜV
      4. Inspektion
      5. Andere
   2. Teilekosten
   3. Arbeitskosten
   4. Beschreibung/ersetzte Teile
   5. Automatischer Eintrag
   6. Bild der Rechnung
3. Erinnerungen
   1. Beschreibung/Name
   2. Datum
   3. Uhrzeit
   4. Push-Notification
4. Andere Kosten
   1. Beschreibung
   2. Kosten
   3. Automatischer Eintrag

Ein Bild der Rechnung kann den Service und Reparaturen – Einträgen angehängt werden, somit lassen sich die Einträge nachweisen und die Datenbank kann als digitales Checkheft genutzt werden.

Die Anmeldung erfolgt mittels E-Mail und Passwort

**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.

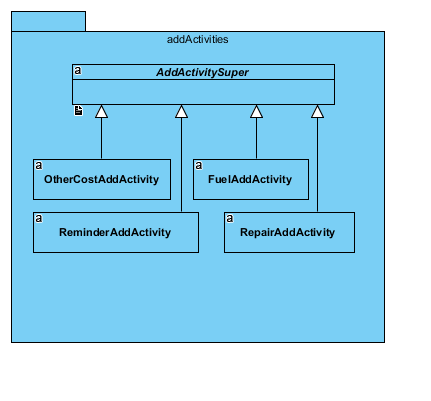
Die Anzeige der Durchschnitts-Werte und Gesamtkosten erfordert den Zugriff der *Fragments* auf die *Data*-Klassen.

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**



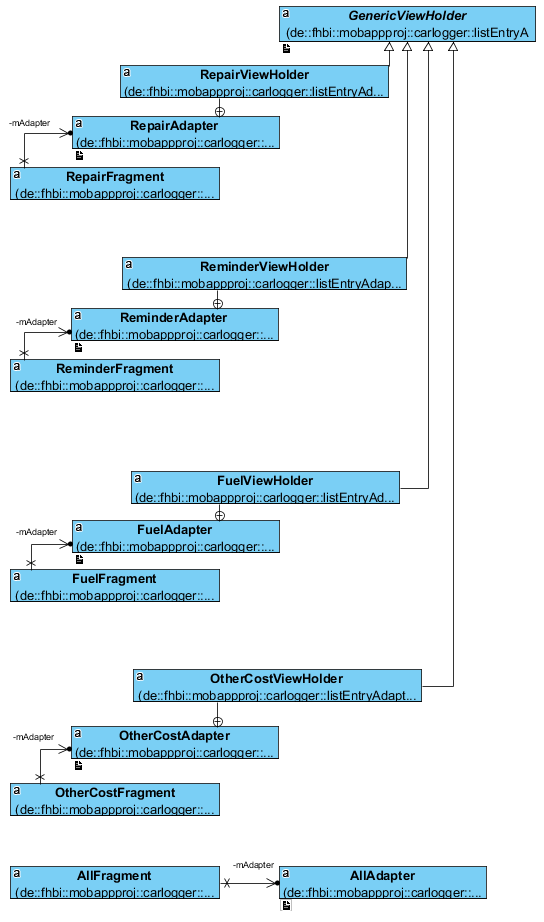
Die *AddActivities* ermöglichen die Eingabe von Datensätzen durch den Benutzer.

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-*Elementsverlassen wird

**Fragments und ListAdapter**



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

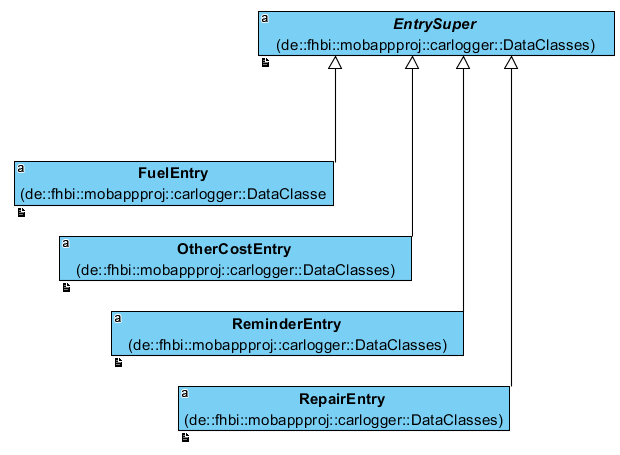
Jede Kategorie von Einträgen nutzt dafür ein eigenes *Fragment.*

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*.

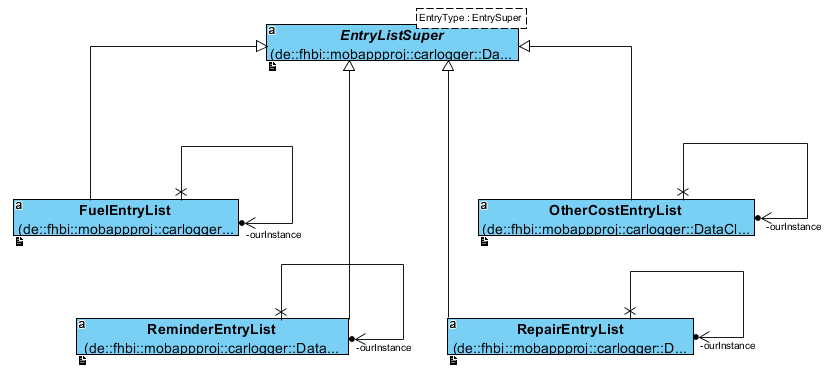
Die Adapter nutzen eine Eintrags-Liste und legen fest welches *Layout* für den Eintrags-Typen verwendet wird.

**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**



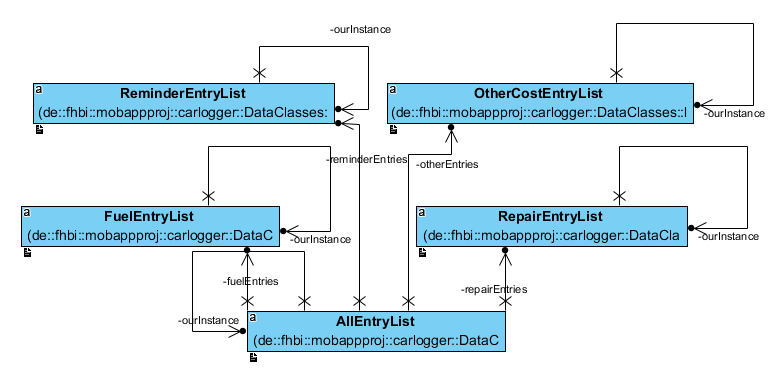
Für jede Eintrags-Kategorie gibt es eine Liste in der die Einträge gehalten werden. Diese Eintrags-Listen benutzen das *Singleton-Design-Pattern* um in der gesamten App nur eine Instanz der jeweiligen Liste zu benutzen. Die generische *Super*-Klasse implementiert Operationen die beispielsweise die Gesamtkosten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

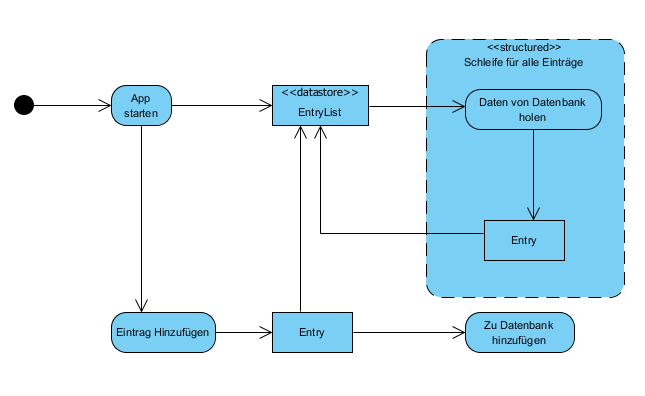
1. pushToFirebase()
2. removeFromFirebase()
3. updateChangesOnFirebase()

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.

**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.