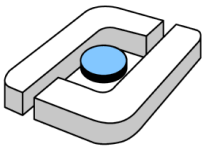

Hochschule Osnabrück

University of Applied Sciences

Fakultät
Ingenieurwissenschaften und Informatik

Schriftliche Projektarbeit zum Thema:

WetterApp

im Rahmen des Moduls
Programmierung 3 (MI),
des Studiengangs Informatik-Medieninformatik

Autor:	Laura Peter
Matr.-Nr.:	869876
E-Mail:	laura.peter@hs-osn-mabrueck.de
Dozent:	Prof. Dr. Rainer Roosmann

Abgabedatum: 13.02.2020

Inhaltsverzeichnis

Inhaltsverzeichnis 2
Abbildungsverzeichnis 3
Source-Code Verzeichnis 4
1 Einleitung 1
1.1 Vorstellung des Themas 1
1.2 Ziel der Ausarbeitung 1
1.3 Aufbau der Projektarbeit 2
2 Darstellung der Grundlagen 3
2.1 Verwendung der OpenWeather API und der Umgang mit JSON 3
2.2 AsyncTask 4
2.3 GestureDetector und GestureListener 4
2.4 Intents und seine Extras 5
3 Anwendung 6
3.1 Aufbau des Klassenschemas 6
3.2 Klassen *Weather* und *Standort* 7
3.3 Klasse Stadt 9
3.4 Klasse DayAndTime 10
3.5 Darstellung 11
3.6 MainActivity 12
3.7 CityList 15
4 Zusammenfassung und Fazit 16
5 Referenzen 17

Abbildungsverzeichnis

Abbildung 1: Ausschnitt eines JSON-Objekts ([@]https://samples.openweathermap.org/data/2.5/forecast?id=524901&appid=b6907d289e10d714a6e88b30761fae22) 3
Abbildung 2: UML-Diagramm 6
Abbildung 3: Code-Ausschnitt weather_five_three 8
Abbildung 4: Screenshot der MainActivity 12
Abbildung 5: Zusammenhänge und Aufgaben von DayUndTime in MainActivity 14
Abbildung 6: Screenshot von CityList 15

Source-Code Verzeichnis

Snippet 1: Definition von AsyncTask 4
Snippet 2: Code-Ausschnitt weather_five_three 8
Snippet 3: Verschiedene Verwendungen von *Weather* 9
Snippet 4: Code-Ausschnitt der Klasse *Stadt* 10
Snippet 5: Speicherung des CityPointers 13
Snippet 6: MyGestureListener mit den Methoden newStadtUp, newStadtDown,nextDay,prevDay (Toast nur zu Debuggingzwecken) 13
Snippet 7: *hinzufuegen* in *CityList* 15
Snippet 8: doSaveInFile() 15

1 Einleitung

Wetter-Informationen und Wetter-Apps werden alltäglich von vielen Menschen gebraucht. Oftmals entscheiden Benutzer solcher Apps danach, wie warm sie sich anziehen. Im Allgemeinen liefern Wetterdienste jedoch nur Informationen, wonach Menschen nur abwägen können, welche Kleidung die angemessenste für den Tag ist.

Deshalb soll in dieser Projektarbeit eine Android-App entwickelt werden, in deren Mittelpunkt eine Figur steht, die der Wettervorhersage entsprechende Kleidung trägt und damit zur Visualisierung des Wetters beiträgt. Dabei sollen herkömmliche Funktionen eines Wetterdienstes erhalten bleiben. Das heißt der Benutzer soll eine Liste ausgesuchter Städte anlegen können, zwischen denen er in der Hauptansicht wechseln kann, um das Wetter in verschiedenen Städten einfach betrachten zu können. Des Weiteren sollen zeitliche Parameter wie die Uhrzeit und der Wochentag verstellbar sein, um dem Benutzer die Wettervorhersage eines bestimmten Zeitpunktes vorzulegen.

1.1 Vorstellung des Themas

Das Thema dieser Projektarbeit ist die Entwicklung einer Android-Applikation mit der Programmiersprache Java, die Wettervorhersagen mit dem alltäglichen Kontext der Kleidung in Verbindung bringt. Dabei wird die Planung einer App und Konzepte der Android-Entwicklung thematisiert.

1.2 Ziel der Ausarbeitung

Die Wetter-App soll folgende Anforderungen erfüllen:

Die Wetter-App soll Wetterdaten und Vorhersagen mit Hilfe der OpenWeather-API [@OWA] zur bereitstellen. Der Benutzer hat eine Liste an Städten, die immer erweitert werden kann, um das Wetter in verschiedenen Städten sehen zu können (um eine neue Stadt zu betrachten kann man durch swipen nach oben und unten zwischen den Städten wechseln). Es soll also eine Möglichkeit gegeben sein, neue Städte einzugeben, diese auf Existenz zu überprüfen und diese dann persistent mit der bisherigen Liste der Städte zu speichern. Der Zustand der App soll zudem in dem Sinne gespeichert sein, dass die Stadt, die zuletzt betrachtet wurde, auch bei Neustart der App wieder angezeigt wird. Gewünscht ist außerdem eine stündliche Vorhersage. Aufgrund der verwendeten API kann jedoch nur das Wetter für alle drei Stunden innerhalb einer Spanne von fünf Tagen vorhergesagt werden. Je nach Temperatur, die von der ausgewählten Stadt und Zeit abhängt, trägt die Figur in der Mitte andere Kleidung und der Hintergrund verändert sich

visuell je nach Beschreibung des Wetters. Der Benutzer soll die zeitlichen und örtlichen Parameter einer Wettervorhersage durch einfache Gesten ändern können.

1.3 Aufbau der Projektarbeit

Im Folgenden Kapitel werden zunächst theoretische Grundlagen erläutert, die zur Realisierung zentraler Funktionen der App essenziell sind. Beispielsweise setzt die Verwendung der API das Verständnis des JSON-Formats und seine Verwendung. Des Weiteren sind durch die benötigten Wetterdaten Networking-Operationen nötig, die in einer Spezialisierung der Klasse *AsyncTask* ablaufen sollen. Zudem bedarf die Realisierung intuitiver Gestiken der Klasse *GestureDetector,* deren Verwendung ebenfalls erläutert werden soll.

Im dritten Kapitel wird die Anwendung der zuvor erklärten Theorie und die Umsetzung der Anforderungen beschrieben. Um einen Überblick über alle Klassen und ihre Beziehungen zu geben wird zunächst das Klassenschema mit Hilfe eines groben UML-Diagramms erklärt. Daraufhin werden die geschriebenen Klassen detaillierter betrachtet. Zunächst werden dabei Klassen betrachtet, die die Wettervorhersage und die Definition ihres Kontexts ermöglichen und damit die Grundlage und den logischen Hintergrund der App bilden. Darauf aufbauend können dann darstellende Methoden erklärt werden, die sich in einem separaten Package befinden und die Ergebnisse der logischen Klassen verwenden, um sie in den Activities darzustellen. Im folgenden Teilkapitel wird dann dargelegt, wie die Activities die logischen und darstellenden Helferklassen zusammenführen, um damit den Verlauf der App zu ermöglichen.

Abschließend für Kapitel 3 werden auf Konzepte der Usability in der App eingegangen, die die Nutzung der App möglichst intuitiv machen soll.

Ein zusammenfassendes Fazit wird die Lösung und die Umsetzung der Anforderung bewerten.

2 Darstellung der Grundlagen

In diesem Kapitel werden theoretische Grundlagen dargelegt, die für die Umsetzung der App unumgänglich sind und Kernfunktionen für die App ermöglichen. Darunter zählen der Umgang der verwendeten API und die Interpretation von JSON-Objekten, in denen Wetterdaten gespeichert sind. Weitergehend wird der Aufruf einer URL innerhalb einer von *AsyncTask* erbenden Klasse und der generelle Umgang mit Networking-Operationen in Android-Applikationen thematisiert. Zuletzt wird die Nutzung des *GestureDetectors* und *GestureListeners* und die damit erreichte intuitive Steuerung der App präsentiert.

2.1 Verwendung der OpenWeather API und der Umgang mit JSON

Um Wetterdaten für die App aus dem Internet zu holen, wird eine API verwendet, die von OpenWeather® zur Verfügung gestellt wird [@OWA]. In kostenloser Nutzung kann beispielsweise „Current weather data“ und ein „5 day / 3 hour forecast“ abgerufen werden. Dabei kann die API bis zu 60 Mal pro Minute aufgerufen werden.

Die Daten werden in JSON- oder XML-Format bereitgestellt, wobei für dieses Projekt JSON-Objekte ausgelesen werden. Ist der Stadtname für eine bestimmte Wettervorhersage bekannt, wird die API über eine URL folgender Form aufgerufen: api.openweathermap.org/data/2.5/forecast?q={city name}&appid={your api key}.

Das JSON (JavaScript Object Notation)-Format ist ein Textformat, das komplett unabhängig von Programmiersprachen ist, aber vielen Konventionen folgt, die aus der Familie der C-basierten Sprachen (darunter Java). Deshalb ist JSON ein verbreitetes Datenaustausch Format. Informationen in diesem Format werden in zwei zentralen Strukturen dargestellt. Ein JSON-Objekt wird als Key-Value-Paar getrennt von einem Doppelpunkt und umrandet von geschwungenen Klammern dargestellt. Ein JSON-Array wird hingegen mit eckigen Klammern umringt und enthält ein oder mehrere Values, die jeweils mit einem Komma getrennt werden. Dabei kann jeder Value beispielsweise wieder ein JSON-Objekt darstellen. [@ECM]


```
"cod": "200",
"message": 0.0036,
"cnt": 40,
"list": [
  {
    "dt": 1485799200,
    "main": {
      "temp": 261.45,
      "temp_min": 259.086,
      "temp_max": 261.45,
      "pressure": 1023.48,
      "sea_level": 1045.39,
      "grnd_level": 1023.48,
      "humidity": 79,
      "temp_kf": 2.37
    },
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "02n"
      }
    ],
```


Abbildung 1: Ausschnitt eines JSON-Objekts

Abbildung 1 zeigt einen Ausschnitt eines JSON-Objekts aus der in diesem Projekt verwendeten API.

2.2 AsyncTask

Um Wetterdaten abrufen zu können müssen an mindestens einer Stelle Network-Operationen durchgeführt werden. Da Network-Operationen eine potenziell lange Laufzeit haben und daher auf irgendeine Weise den UI-Thread blockieren können, können sie dazu führen, dass die Applikation nicht mehr antwortet („Application Not Responding“). Deshalb ist es ab der Honeycomb SDK nicht mehr möglich, Networking-Operationen im Main-Thread durchführen. Ist dies der Fall wird die *NetworkOnMainThreadExcepion* geworfen. Der effektivste Ansatz dies zu umgehen, ist einen Worker-Thread zu erstellen, der solche Operationen in einer Klasse erledigt, die von *AsyncTask* erbt und *doInBackground* implementiert, also dort die Network-Operationen durchführt [@ADR] [@ADE].

AsyncTask erlaubt es Operationen auf einem Hintergrund-Thread durchzuführen und sie dann im UI-Thread zu veröffentlichen. Ein „asynchronous task“ wir durch die 3 generische Typen *Params, Progress* und *Result* definiert:

```
public class Weather extends AsyncTask<Void,Void,Void> {
```

Snippet 1: Definition von AsyncTask

Params definiert den Typen der Parameter, die vor der Durchführung übergeben werden müssen. *Progress* definiert den Datentypen der Einheiten die während der Durchführung von *DoInBackground()* veröffentlicht werden. *DoInBackground()* kann einen Rückgabewert liefern, dessen Datentyp durch *Result* festgelegt wird. *OnPostExecute()* wird im Anschluss von *DoInBackground()* durchgeführt und erhält gegebenenfalls das Ergebnis von *DoInBackground()* als Parameter mit entsprechendem Datentypen übergeben.

In diesem Projekt wird *AsyncTask* dazu verwendet im Hintergrund die Abfrage der Wetter-API durchzuführen und die daraus folgenden Ergebnisse über *OnPostExecute()* im User Interface darzustellen.

2.3 GestureDetector und GestureListener

Um intuitive Nutzung der App zu ermöglichen, sollen zum Ändern der Parameter einer Wettervorhersage (Stadt, Zeit), Swipe-Gesten eingesetzt werden. Dazu wird ein Objekt der Klasse *GestureDetector* verwendet, die von Android zu Verfügung gestellt ist, um übliche Gestiken zu interpretieren. Wird ein *GestureDetector* instanziiert, ist ein Parameter, den es dazu benötigt eine Klasse, die *GestureDetector.OnGestureListener* implementiert. Wenn die überschriebene Callback-Methode *OnTouchEvent()* bei einer Geste

des Benutzer aufgerufen wird, werden dem *GestureDetector* alle beobachteten Events übergeben.

In der Definition von *GestureListener* können die Auswirkungen verschiedener Gesten implementiert werden. Für die Realisierung der benötigten Swipe-Geste wird die Methode *OnFling(MotionEvent event1, MotionEvent event2, float velocityX, float velocityY)* Implementiert. Dazu werden *event1* und *event2* Koordinaten entnommen, um sie miteinander zu vergleichen und daraus die Richtung des Swipes zu ermitteln [@ADG].

2.4 Intents und seine Extras

Intents werden hauptsächlich verwendet, um aus einer bestimmten Activity eine andere zu starten. Sie sind Nachrichten Objekte, die zusätzlich Key-Value-Paare mit zusätzlicher Information mit sich bringen können, die gebraucht werden, um bestimmten Aufgaben in der nächsten Activity erfüllen zu können. Mithilfe der *putExtra()* können dem Intent neue Extras hinzugefügt werden. Dabei kann auch ein *Bundle*-Objekt erzeugt werden, die alle benötigten Daten beinhalten und dieses dann zu den Extras des Intents hinzufügen [@ADI].

Einem Bundle können viele verschiedene Typen hinzugefügt werden. Im Gegensatz zum Intent, kann dem Bundle beispielsweise eine *ArrayList<String>* zugefügt werden. Da in diesem Projekt eine *ArrayList<String>* zwischen zwei Activities übergeben werden soll, muss ein Bundle verwendet werden, das die ArrayList enthält und im Intent als Extra übergeben wird.

3 Anwendung

In diesem Kapitel wird zunächst grob der Aufbau des Klassenschemas anhand des UML-Diagramms in Abbildung 2 erklärt, um daraufhin genauer auf die einzelnen Klassen einzugehen.

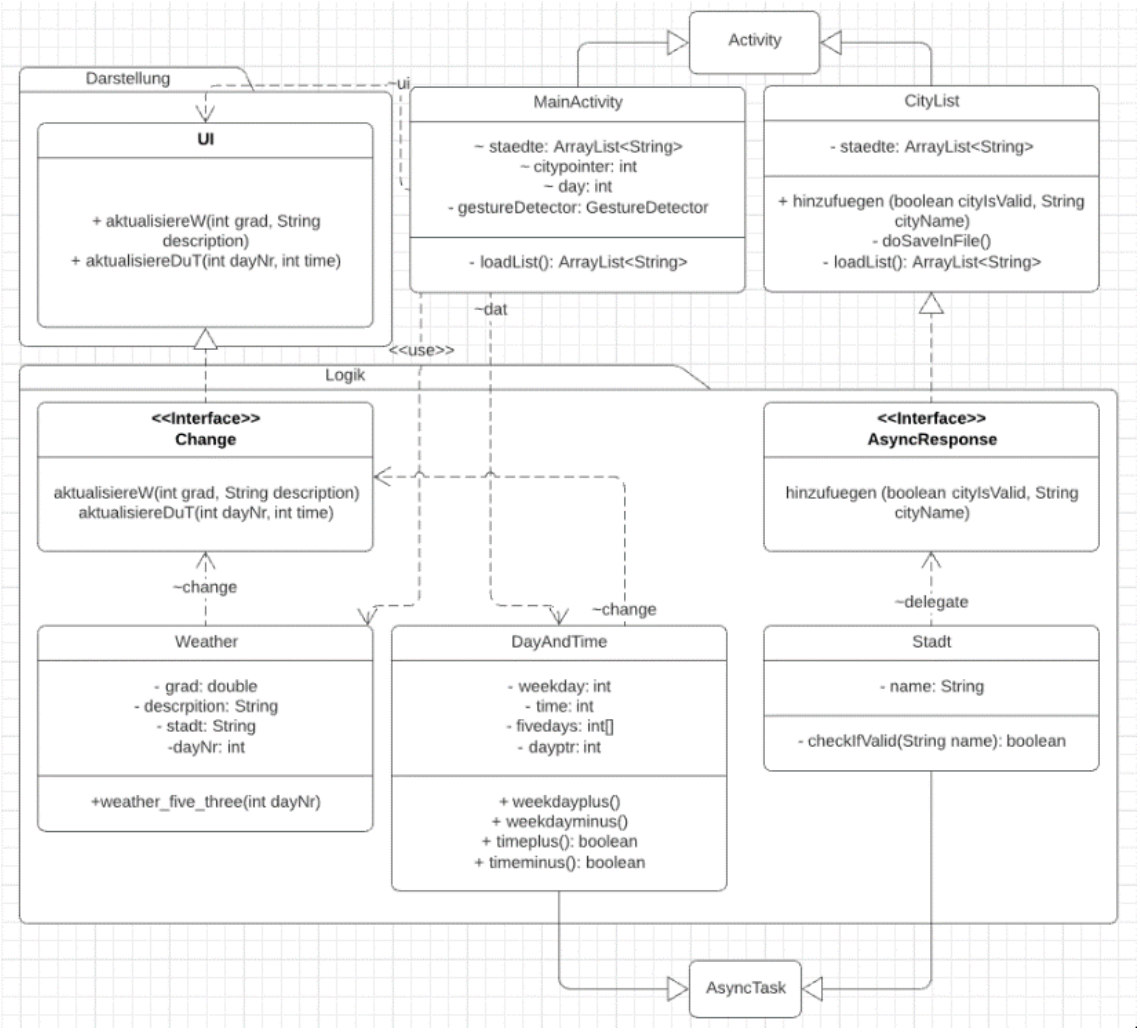

Abbildung 2: UML-Diagramm

3.1 Aufbau des Klassenschemas

Vergleiche Abbildung 2, spielt sich die App spielt hauptsächlich auf zwei Activities ab, die verschiedene Helferklassen aus dem Logik-Package mit jenen aus dem Darstellungs-Package zusammenführen.

Die *MainActivity* spiegelt den Hauptkontext und die Hauptaufgabe der App wider. In dieser Ansicht findet man die Ergebnisse der Wettervorhersage, die auf Basis zeitlicher und örtlicher Parameter entstanden sind und sich visuell im dynamisch veränderten Hintergrund und der Kleidung der im Zentrum stehenden Figur äußern. Des Weiteren kann der

Benutzer durch Swipe- und Click-Gesten beeinflussen zu welcher Zeit und an zu welchem Ort das Wetter vorhergesagt werden soll. Jedes Mal, wenn einer dieser Parameter verändert wird, wird ein neues Objekt vom Typ *Weather* angelegt, welches die zu den Parametern passenden Wetterdaten zu Verfügung stellt.

Der Logische Hintergrund der Wettervorhersage wird in der Klasse *Weather* verarbeitet. Nach abrufen und verarbeiten der Wetterdaten durch die API, wird die Darstellung der *MainActivity* angepasst, indem auf der Objektvariable *change* vom Typ *Change* eine Methode zum Aktualisieren der Darstellung aufgerufen wird.

Die logische Klasse *DayAndTime* dient zur Ermittlung der aktuellen Tageszeit, um die die zeitlichen Parameter der Wettervorhersage festzulegen, die beim Start der App erfolgt. Bei Veränderung der zeitlichen Parameter werden diese mit Rücksicht auf die verfügbare Tagesspanne von fünf Tagen verwaltet. Daraufhin wird Die Darstellung ebenfalls über das Interface *Change* aktualisiert, welches von der Klasse *UI* implementiert wird. Das Interface dient zur losen Kopplung der Packages *Logik* und *Darstellung*.

Über einen „+“-Button gelangt man in der *MainActivity* zur zweiten Activity namens *CityList*. Hier kann der Benutzer über ein Eingabefeld Städte suchen, um diese dann persistent in einer Liste vorliegen zu haben, durch die man per Swipe-Geste in der MainActivity blättern kann. Bei der Suche einer Stadt, muss geprüft werden, ob diese durch die API aufgerufen werden kann. Dies erfolgt durch die logische Klasse *Stadt*, die über das Interface *AsyncResponse* mit *CityList* gekoppelt ist.

Zu beachten ist, dass Abbildung 2 nicht alle Methoden und Klassenvariablen jeder Klasse aufzeigt. Abgebildet sind nur jene, die zum Verständnis der Zusammenhänge zwischen den Klassen und Packages und deren Hauptaufgabe nötig sind.

3.2 Klassen *Weather* und *Standort*

Die Klasse Weather bildet das Herzstück der App, da aus dem Ergebnis der Wettervorhersage alle Aktionen ausgehen, die zur zentralen Funktion der Anwendung beitragen.

Jedes Wetterobjekt zeichnet sich durch eine bestimmte Stadt und einen bestimmten Zeitpunkt (im Code als *dayNr* bezeichnet) aus und muss bei der Initialisierung ein Objekt übergeben bekommen, welches das Interface *Change* implementiert, sodass Updates in der Darstellung durch die Klasse UI durchgeführt werden können.

Für die Wettervorhersage wird hauptsächlich die Methode *weather_five_three(int dayNr)* verwendet, in der eine API aufgerufen wird, die das Wetter einer beliebigen Stadt in 3-Stunden Abständen innerhalb einer Spanne von fünf Tagen auflistet. Diese Information

wird in Form eines JSON-Objektes übergeben und wird dementsprechend in der Funktion ausgelesen.

```
public void weather_five_three(int daynr){
    JSONParser parser = new JSONParser();
    try {
        URL url = new URL( spec: "http://api.openweathermap.org/data/2.5/forecast?q=" + stadt +
                           "&units=metric&appid=a46b2315f08f5154c460af1ad9cb70ae");
        BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

        String input;

        while ((input = in.readLine()) != null) {

            JSONObject a = (JSONObject) parser.parse(input);
            JSONArray list = (JSONArray) a.get("list");
            JSONObject day = (JSONObject) list.get(daynr);
            JSONObject main = (JSONObject) day.get("main");
            this.grad = (double) main.get("temp");

            JSONArray weather = (JSONArray) day.get("weather");
            JSONObject zero = (JSONObject)weather.get(0);
            this.description = (String) zero.get("description");
        }
        in.close();
    } catch (MalformedURLException e) {
        e.printStackTrace();
```

Snippet 2: Code-Ausschnitt weather_five_three

Abbildung 3: Code-Ausschnitt weather_five_three

Um das JSON-Objekt einfach zu parsen, wird hier das Java Toolkit json-simple verwendet [@JSO].

Um beim Aufruf der URL der `android.os.NetworkOnMainThreadException` zu entgehen, erbt *Weather* von *AsyncTask* und muss demnach die Funktionen *doInBackground* und *onPostExecute* implementieren. In *doInBackground* wird der URL-Aufruf mit *weather_five_three* getätigt und in *onPostExecute* wird dessen Ergebnis genutzt, um die Darstellung des User Interfaces zu ändern. Es wird also die update-Methode auf dem *Change*-Objekt aufgerufen.

Die API bietet weitere Funktionen an, um das Wetter in anderen Umständen zu betrachten. Darunter kann das aktuelle Wetter betrachtet werden. Diese Funktion wird in *Weather* mit Hilfe der Methode *current_weather()* eingesetzt, wenn *dayNr* auf 0 gesetzt ist (vgl. Snippet 3, Zeile 154), d.h. der aktuelle Zeitpunkt wird betrachtet. In diesem Fall ist es sinnvoller *current_weather()* als *weather_five_three* zu verwenden, da der entsprechende API-Aufruf zeitlich genauere Wetterdaten für die aktuelle Zeit liefert. *weather_five_three* wird also nur, wenn zeitliche Parameter in der *MainActivity* verändern werden, also eine andere Zeit, als die aktuelle betrachtet wird.

```
147     protected Void doInBackground(Void... voids) {
148
149         if (coordinateFlag) {
150             weather_coordinates(lat, lon);
151         } else if (dayNr == 0) {
152             current_weather();
153         } else {
154             weather_five_three(this.dayNr);
155         }
156         return null;
157     }
158
159     @Override
160     protected void onPostExecute(Void aVoid) {
161         super.onPostExecute(aVoid);
162         change.aktualisiereW((int) this.grad, this.description, this.stadt);
163     }
164 }
```

Snippet 3: Verschiedene Verwendungen von *Weather*

Für den Fall, dass noch keine Stadt zu Verfügung steht, dessen Wetterdaten man über die URL mit eingesetztem Städtenamen beziehen kann, kann mit Hilfe der Klasse *Standort* der Standort des Benutzers ermittelt werden und über die Koordinaten als Parameter *weather_coordinates* aufgerufen werden. Diese Methode ruft ebenfalls das aktuelle Wetter einer Stadt auf, jedoch hierbei über ihre Koordinaten.

In der *MainActivity* wird ein Objekt vom Typ *Standort* instanziiert, falls beim Start der App noch keine Stadt in der gespeicherten Liste vorliegt. Der Konstruktor von *Standort* ruft dabei die Methode *requestLocation* auf, die bei Erststart der App zunächst per Dialog-Fenster nach der Permission zur Standortermittlung über die *ACCESS_FINE_LOCATION* fragt. Gibt der Benutzer die Erlaubnis zur Standortermittlung. Die Klasse implementiert *LocationListener,* wodurch die Methode *onLocationChanged(Location location)* überschrieben werden kann. Voraussetzung dafür ist, dass *Standort* auf einem Objekt vomTyp *LocationManager* die Methode *requestLocationUpdates* ausführt. In *onLocationChanged(Location location)* werden die Koordinaten des aktuellen Standorts ermittelt und damit ein neues Wetterobjekt erzeugt, das durch das auf *true* gesetzte *coordinateFlag* im Konstruktor bei *execute()* die Methode *weather_coordinates* aufgerufen wird (vgl. Snippet 3, Zeile 150) [@JPL].

3.3 Klasse Stadt

Die Klasse *Stadt* ist ähnlich wie *Weather* aufgebaut, da hier auch die URL mit dem gesuchten Stadtnamen der API aufgerufen werden muss, um zu prüfen, ob man mit der Stadt, die man sucht auch die API aufrufen kann. Deshalb erbt Stadt ebenfalls AsyncTask und ruft zur Aktualisierung der Darstellung über *execute()* die Methode *hinzufuegen* auf der Objektvariable *delegate* vom Typ *AsyncResponse* auf, welches die

Activity *CityList* implementiert. In *checkIfValid(String name)* wird geprüft, ob die URL, die mit eingesetztem Stadtnamen zu der API führen kann und somit als anerkannte und existierende Stadt gewertet kann, um später in der *CityList*-Activity in die permanent gespeicherte Liste eingetragen zu werden (vgl. Snippet 4).

```
public boolean checkIfValid(String name){
    JSONParser parser = new JSONParser();

    try {
        URL url = new URL( spec: "http://api.openweathermap.org/data/2.5/forecast?q=" + name +
                                  "&units=metric&appid=a46b2315f08f5154c460af1ad9cb70ae");
        BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()));

        String input;
        while ((input = in.readLine()) != null) {

            JSONObject a = (JSONObject) parser.parse(input);
            String message = (String) ""+ a.get("message");
            if (message.equals("city not found")){

                return false;
            }

        }
        in.close();
    }catch (Exception e){
        return false;
    }

    return true;

}

@Override
protected Boolean doInBackground(Void... voids) {
    return checkIfValid(this.name);

}
@Override
protected void onPostExecute(Boolean aVoid) {
    delegate.hinzufuegen(aVoid,this.name);
    super.onPostExecute(aVoid);
```

Snippet 4: Code-Ausschnitt der Klasse *Stadt*

3.4 Klasse DayAndTime

Wie alle anderen Logik-Klassen hat diese Klasse ein Objekt vom Typ *Change,* um die Ergebnisse der einzelnen Methoden im User Interface darzustellen. Beim Start der App werden zunächst der aktuelle Wochentag und die aktuelle Zeit mit Hilfe der Klasse *Calender* ermittelt, um damit das *DayAndTime*-Objekt zu initialisieren. Da aufgrund der verwendeten API nur eine Fünf-Tages-Spanne für die Wettervorhersagen zur Verfügung steht, werden die darstellbaren fünf Wochentage, ausgehend vom aktuellen Wochentag

in der Methode *setFiveDays()* ermittelt und in Form von Integern in einem Array hinterlegt, durch das zur Laufzeit durchiteriert wird, wenn sich der darzustellende Wochentag ändert.

In der *MainActivity* kann durch Swipe-Gesten nach links und nach rechts der Wochentag gewechselt werden, dessen Wetter betrachtet werden soll. Dabei werden die Methoden *weekdayplus()* und *weekdayminus()* aufgerufen. Um durch das Fünf-Tage-Array durchzuiterieren wird ein Pointer als Objektvariable gespeichert, der initial 0 ist und bei Änderung des Wochentages jeweils in- oder dekrementiert wird. Entspricht der Pointer dem Anfangs- oder End-Index des Arrays kann entweder *weekdayminus()* oder *weekdayplus()* keine Änderung des Pointers und kein Update der Darstellung der UI bewirken.

Die Uhrzeit wird in der MainActivity als Uhr dargestellt und kann durch zwei Pfeilbuttons ober- und unterhalb verändert werden. Die API nur Wetterdaten für alle drei Stunden zur Verfügung, weshalb die Uhrzeit auch hier nur eine sein kann, die durch drei teilbar ist. Initial wird *DayAndTime* deshalb mit einer Uhrzeit definiert, die durch die Methode *timeOfDay(Calender c)* ermittelt wird. Hier wird zunächst die Zeit mit Hilfe von *Calender* ermittelt, um daraufhin die Stunde zu erschließen, die in der API zuerst aufgelistet wird. Dadurch wird beim Start der die Wettervorhersage mit möglichst aktueller Zeit dargestellt.

Wird die Uhrzeit verändert, werden ähnlich wie bei den Wochentagen dabei die Methoden *timeplus()* und *timeminus()* aufgerufen, die mit Rücksicht auf den aktuell betrachteten Wochentag die Zeit entweder um drei erhöhen oder verringern. Werden dabei Tagesgrenzen überschritten, können auch wieder *weekdayminus()* oder *weekdayplus() aufgerufen werden.*

3.5 Darstellung

Das Package *Darstellung* enthält nur die Klasse *UI.* Hier werden alle Methoden gekapselt, die Änderungen jeglicher Views in der *MainActivity* bewirken. Dazu werden die Views der *MainActivity* im Konstruktor der *UI* lokalen Objektvariablen zugewiesen, um den späteren Zugriff darauf zu erleichtern.

Da die Klasse *UI* das Interface *Change* implementiert, werden hier die Update-Methoden definiert*,* die durch logische Klassen aufgerufen werden. Eine der zwei Update-Methoden (*aktualisiereW(int grad, String description)*) wird in *Weather* aufgerufen und aktualisiert Views, die direkt mit dem Ergebnis der Wettervorhersage zusammenhängen, also den Wetter-beschreibenden Hintergrund und die im Vordergrund stehende Figur.

Die zweite Update-Methode (*aktualisiereDuT(int day, int time)*) aktualisiert hingegen Views, die mit der Darstellung der Zeit zusammenhängen. Also dem aktuell betrachteten Wochentag am unteren Rand des Displays und der Uhr am rechten Rand.

3.6 MainActivity

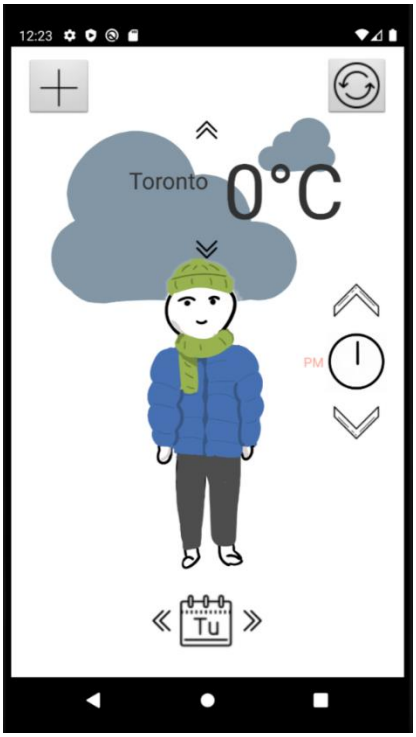

Abbildung 4: Screenshot der MainActivity

In der *MainActivity* werden Darstellung und Logik zusammengeführt. Demnach verwaltet die *MainActivity* Objekte der Klassen *Weather, DayAndTime* und *UI.* Zudem wird die Variable *day* gespeichert, die einem Index in der API entspricht und den aktuell betrachteten Tag oder Zeitpunkt symbolisieren soll. Des Weiteren hat die *MainActivity* eine Liste mit Städtenamen (im Code *staedte* genannt), durch die während der Laufzeit durchiteriert werden kann und die dynamisch erweitert werden kann, indem auf den „+“-Button geklickt wird, der zu einer weiteren Activity führt. Die angezeigte Stadt wird durch eine Pointer-Variable (im Code *int citypointer* genannt), die einem bestimmten Index in *staedte* entspricht.

Die Änderung der zu betrachtenden Stadt, sowie die Änderung des zu betrachtenden Wochentages, werden durch Swipe-Gesten gesteuert, die durch ein Objekt der Klasse *GestureDetector* realisiert werden. Dafür wird *gestureDetector* in der *OnCreate*-Methode der Activity mit einem Context, der der MainActivity entspricht, und einer implementierten Version des *GestureDetector.OnGestureListener* konstruiert. *GestureDetector.OnGestureListener* wird mittels der inneren Klasse *MyGestureListener* implementiert. Um Methoden durch Swipe-Gestik auszulösen wird die Methode *OnFling* verwendet, wobei durch Vergleich von y-Koordinaten der *MotionEvents* vertikales Swipen erkannt wird. Je nachdem, ob nach unten oder nach oben geswipet wird, ändert sich die angezeigte Stadt. Es werden die entsprechenden Methoden *newStadtUp()* oder *newStadtDown()* aufgerufen (vgl. Snippet 6), die *citypointer* in- oder dekrementieren und den Stadtnamen mit dem entsprechenden Index in *staedte* anzeigen lassen. Um das Wetter für die neue Stadt darstellen zu können wird ein neues Wetterobjekt initialisiert und dabei der neue Stadtname übergeben. Auf diesem Wetterobjekt wird dann *execute()* aufgerufen, um die Wetterdaten abzurufen.

Einer der Anforderungen für die App war es außerdem, dass die Stadt, die zuletzt betrachtet wurde, auch beim Neustart der App wiedererscheint. Dies wurde realisiert, indem der cityPointer in den *SharedPreferences* der App gespeichert wurde. Jedes Mal, wenn sich der Wert des Pointers ändert wird er in *pointerSave(int ptr)* in den *SharedPreferences* abgespeichert (vgl. Snippet 5, Zeile 145-148). Zum Neustart der App wird der Pointer über *loadPointer()* initialisiert. Falls der Pointer noch nie gespeichert wurde, wird ihm der Wert 0 zugewiesen (vgl. Snippet 5, Zeile 153).

```
144     void pointerSave(int ptr) {
145         SharedPreferences sharedPreferences = this.getSharedPreferences( name: "Pointer", Context.MODE_PRIVATE);
146         SharedPreferences.Editor editor = sharedPreferences.edit();
147         editor.putInt("cityPointer", ptr);
148         editor.apply();
149     }
150
151     int loadPointer() {
152         SharedPreferences sharedPreferences = this.getSharedPreferences( name: "Pointer", Context.MODE_PRIVATE);
153         int ptr = sharedPreferences.getInt( key: "cityPointer",  defValue: 0);
154         return ptr;
155     }
```

Snippet 5: Speicherung des CityPointers

Der Wochentag wird geändert, indem nach links oder rechts geswipet wird. In *OnFling* wird dazu die x-Koordinate der beiden übergebenen *MotionEvents* verglichen. Bei horizontalem Swipe wird dann entweder *nextDay()* oder *prevDay()* ausgeführt (vgl. Snippet 6), welche daraufhin auf dem *DayAndTime*-Objekt *weekdayPlus()* oder *weekdayMinus()* ausführen, um zu überprüfen, ob es möglich ist den Wochentag zu wechseln Die Variable *day* wird dann um 8 inkrementiert, da ein Tag in der API aus 8 Zeitabschnitten besteht. Auch hier werden wieder Wetterdaten über ein neues Wetterobjekt mit dem neuen Wert von *day* ermittelt und dann dargestellt (vgl. Abbildung 5).

```
221 @   public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
222         if (e1.getY() - e2.getY() > 75) {
223             newStadtUp();
224             return true;
225         }
226         if (e2.getY() - e1.getY() > 75) {
227             newStadtDown();
228             return true;
229         }
230         if (e1.getX() - e2.getX() > 75) {
231             nextDay();
232             return true;
233         }
234         if (e2.getX() - e1.getX() > 75) {
235             prevDay();
236             return true;
237         }
238         return false;
239
240     }
```

Snippet 6: MyGestureListener mit den Methoden newStadtUp, newStadtDown,nextDay,prevDay (Toast nur zu Debuggingzwecken)

Das gleiche Prinzip vertreten auch die OnClick-Methoden *nextTime(View view)* und *prevTime(View view),* die Wetterdaten in 3-Stunden-Abschnitten darstellen lassen. Dabei wird *day* jedoch nur um eins inkrementiert oder dekrementiert, da sich der betrachtete Zeitabschnitt in der API nur um 1 verändert (vgl. Abbildung 5).

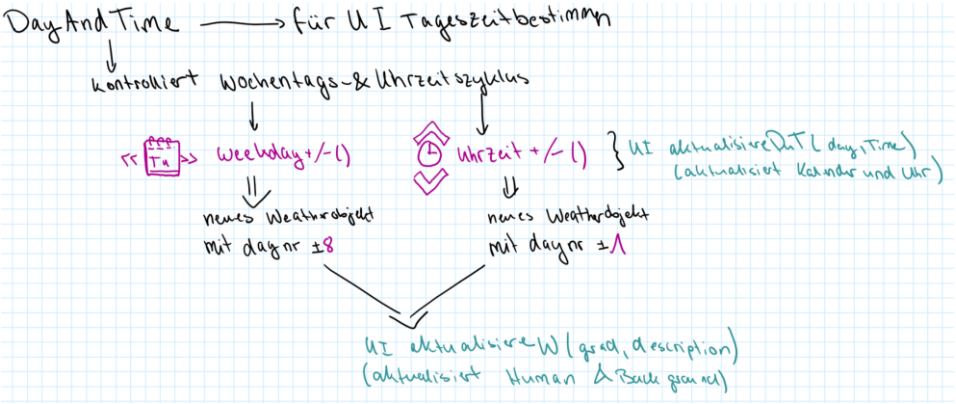

Abbildung 5: Zusammenhänge und Aufgaben von DayUndTime in MainActivity

Mit der OnClick-Methode *toCityList* des „+“-Buttons gelangt man zur CityList-Activity. Im Intent wird durch sein Extra die String-ArrayList innerhalb eines Bundles übergeben, die in CityList zur Darstellung benötigt wird.

3.7 CityList

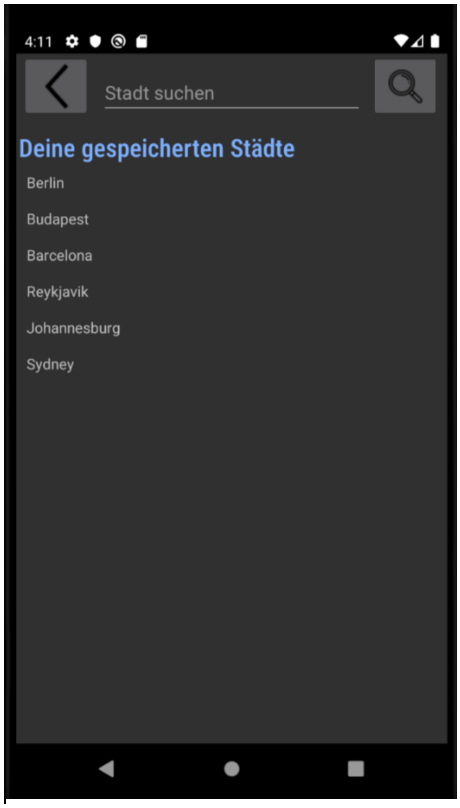

Abbildung 6: Screenshot von CityList

Die Activity *CityList* dient zum Suchen neuer Städte und der persistenten Speicherung einer Liste von Städten, die die neu hinzugefügten beinhaltet.

Die von der MainActivity übermittelte ArrayList wird in der *OnCreate*-Methode dieser Activity zunächst durch die Funktion *loadList()* abgerufen und der lokalen Objektvariable *staedte* zugewiesen. Mit der Methode *staedteAuflisten()* werden im LinearLayout *ll* die Städte aufgelistet, die bereits in der übermittelten Liste gespeichert waren.

Gibt der Benutzer einen String in das Edit-Textfeld und löst die Suche dann mit dem Such-Button *btn_search* aus, wird, ähnlich wie in *Weather*, ein neues *Stadt*-Objekt erstellt, das die URL-Anfrage zur Prüfung des eingegebenen Strings durchführt und bei positivem Ergebnis *hinzufuegen* auf *CityList* aufruft.

```
public void hinzufuegen(boolean cityIsValid, String cityName){
    if(cityIsValid){
        if(!isDuplikat(cityName)){
        staedte.add(cityName);
        TextView neu=new TextView( context: this);
        neu.setText(cityName);
        neu.setPadding( left: 20, top: 20, right: 20, bottom: 20);

        ll.addView(neu);
        doSaveInFile(); //persistent eintragen
        }

    }
    else {
        Toast.makeText( context: CityList.this, "Stadt nicht gefunden"
    }
```

```
private void doSaveInFile(){
    try(OutputStreamWriter out = new OutputStreamWriter(
            this.openFileOutput(CITYLIST,Context.MODE_PRIVATE))){
        staedte.stream().forEach(s->{
            try{
                out.write( str: s+"\n");
            }catch (IOException e) {
                e.printStackTrace();
            }
        });
    }catch (IOException e){
```

Snippet 8: doSaveInFile()

Innerhalb der Methode wird der Stadtname als TextView im LinearLayout unterhalb des Eingabefeldes hinzugefügt, um dem Benutzer visuell klarzumachen, dass die gewünschte Stadt in seine Liste aufgenommen wurde.

doSaveinFile() sorgt für die persistente Speicherung der Liste in einer Datei, die im internen Speicher der App liegt.

4 Zusammenfassung und Fazit

Der Benutzer kann in der *MainActivity* jederzeit Wettervorhersagen für alle 3 Stunden über 5 Tage betrachten und die Städte speichern, für die er das Wetter gerne betrachten würde. Dies wurde durch den Aufruf der OpenWeather-API erreicht. Dabei konnte keine stündliche Vorhersage realisiert werden, so wie es ursprünglich gewünscht war, da die API dies in kostenloser Nutzung nicht zulässt. Des Weiteren bedarf es für den Aufruf der URL zur API einer Klasse, die von *AsyncTask* erbt. Dies führt dazu, dass die Methode mit dem URL-Aufruf nur einmal durch *execute()* ausgeführt werden kann. Folglich muss jedes Mal, wenn einer der Parameter für das angezeigte Wetter verändert wird, ein neues *Weather*-Objekt instanziiert werden, um die Darstellung entsprechend der neuen Wetterdaten zu ändern. Dabei war zunächst geplant, dass es ein Wetterobjekt während der gesamten Laufzeit gibt, dessen Objektvariablen sich je nach Wetterkontext aktualisieren.

In der MainActivity gibt es die Möglichkeit die Wettervorhersage über den „Refresh“-Button zu aktualisieren. Da sich die Daten der 5 days/3 hour-forecast-API höchstens alle zwei Stunden aktualisieren, ist es fragwürdig, ob die Funktion dieses Buttons Sinn macht, es sei denn die App ist über Stunden im Hintergrund geöffnet. Der Button ist jedoch sinnvoll, wenn das aktuelle Wetter betrachtet wird, da die dazu verwendete API öfter aktualisiert wird.

Hinzufügen lässt sich auch, dass die Usability der App noch verbessert werden könnte, indem mehr Animationen eingefügt werden würden, die die Swipe-Gesten unterstützen und dem Benutzer noch ein besseres visuelles Feedback über das geben, was sich auf in der App ändert.

Zusammenfassend lässt sich jedoch sagen, dass die gewünschten Anforderungen als Android-Applikationen erfüllt werden konnten und der Benutzer die angebotenen Funktionen mit einfacher Steuerung benutzen kann.

5 Referenzen

[@JSO] https://code.google.com/archive/p/json-simple/downloads

[@OWA] https://openweathermap.org/api

[@ECM] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf Seite 1-3

[@ADR] https://developer.android.com/training/articles/perf-anr?hl=de Keeping your app responsive

[@ADE] https://developer.android.com/reference/android/os/NetworkOnMainThreadException

[@ADG] https://developer.android.com/training/gestures/detector

[@ADI] https://developer.android.com/guide/components/intents-filters

[@JPL] https://javapapers.com/android/get-current-location-in-android/

Alle Seiten wurden am 13.02.2020 das letzte Mal aufgerufen.

Eidesstattliche Erklärung

Hiermit erkläre ich/ Hiermit erklären wir an Eides statt, dass ich/ wir die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe/ haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Urheberrechtliche Einwilligungserklärung

Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

....
Ort, Datum Unterschrift

(bei Gruppenarbeit die Unterschriften sämtlicher Gruppenmitglieder)

Hinweis: Die urheberrechtliche Einwilligungserklärung ist freiwillig.