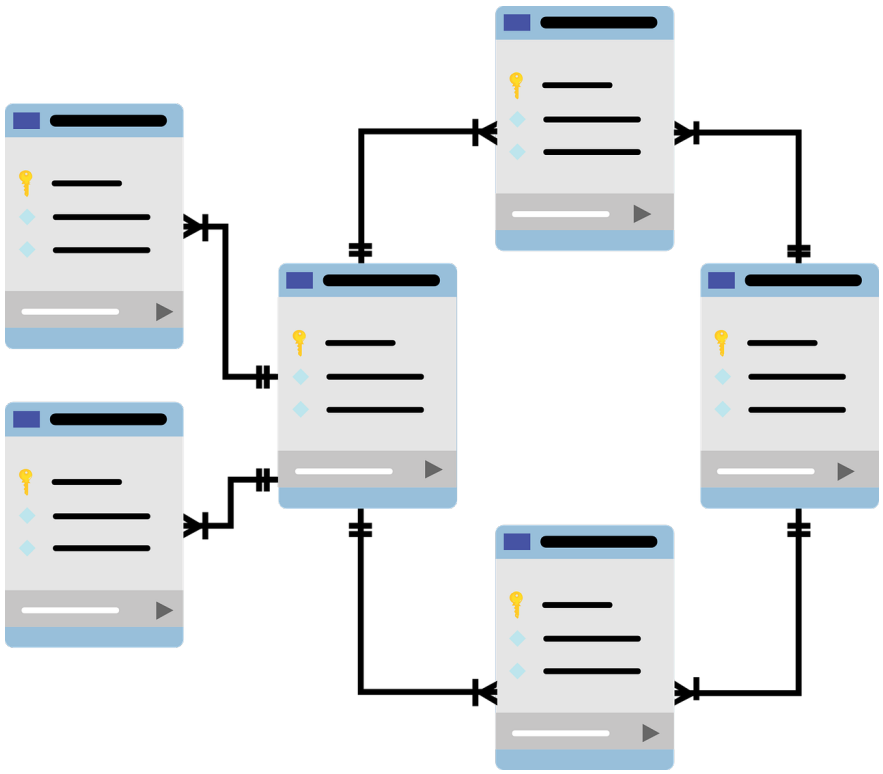
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Visualisering av statistikdata i en Androidapplikation

Databas, Grafisk Display, Kommentarhantering

Examensarbete inom Data- och Informationsteknik

JESPER SKOGLUND
NIKLAS CÔTÉ

Institutionen för Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Examensarbete 2019

Visualisering av statistikdata i en Androidapplikation

Databas, Grafisk Display, Kommentarhantering

JESPER SKOGLUND
NIKLAS CÔTÉ

CHALMERS

Institutionen för Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Visualisering av statistikdata i en Androidapplikation
Databas, Grafisk Display, Kommentarhantering
JESPER SKOGLUND
NIKLAS CÔTÉ

Handledare: Jacob Thorsell, Acobiaflux
Handledare: Uno Holmer, Institutionen för Data- och informationsteknik
Examinator: Jonas Duregård, Institutionen för Data- och informationsteknik

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
412 96 Göteborg
Telefon: 031-772 1000

Omslag: Visuell representation av data i en databas.

Institutionen för Data- och informationsteknik
Göteborg, Svergie 2019

Visualization of statistics data in an Android application
Database, Graphic Display, Persistent Comments
JESPER SKOGLUND, NIKLAS CÔTÉ
Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg

Abstract

Acobiaflux has a prototype of a mobile application which they have an interest of developing. The project's purpose is documenting and developing their application with a focus on the Android environment. The purpose of developing the application is to make it easier for the user to manage alarms with the help of a graphical view. The project turned out to be more structure oriented and trying to develop the application in parallel to the existing code because of outdated structure. The structure used was according to the MVVM design pattern which is a recommended framework for Android development. We also made our code independent from the existing code, as far as possible, to show how an Android application is supposed to be structured. The work has been done at Acobiafluxs office, while working we tried to work iteratively and dividing the projects different parts while communicating regularly. In the end, we deliverd an application that met the requirements Acobiaflux set for us which they are very content with. Finally, we encourage Acobiaflux to rebuild the application in a cross platform environment because the foundation of the Android application is built on deprecated libraries. The application exists for iOS as well, which means that there are two different projects which could be one, this is another reason Acobiaflux should rebuild the application in a cross platform environment.

Keywords: Android, MVVM, Kotlin, Dependency Injection, Fragments, LiveData.

Visualisering av statistikdata i en Androidapplikation
Databas, Grafisk Display, Kommentarhantering
JESPER SKOGLUND, NIKLAS CÔTÉ
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet

Sammanfattning

Acobiaflux har en prototyp av en mobilapplikation som de har intresse av att vidareutveckla, vårt projekt handlar om att dokumentera och utveckla deras applikation med fokus på Android-delen. Utvecklingen handlar om att göra det lättare för användare att hantera larm och skapa en grafisk vy så användare snabbt kan få en överblick. Projektet övergick till att ändra strukturen på applikationen och att skapa en del av applikationen som fungerar oberoende av den befintliga. Det som vi gjort annorlunda är att följa MVVM strukturen som är ett rekommenderat ramverk för Androidutveckling och gjort vår kod självstående från den befintliga i största möjliga utsträckning för att visa hur en Androidapplikation skall struktureras. Arbetet har pågått på Acobiafluxs kontor, under arbetetsgången har vi försökt arbeta iterativt med mycket kommunikation och fördelning av arbetet. I slutändan lyckades vi leverera en applikation som uppfyller de krav Acobiaflux ställt på projektet som de är nöjda med. Slutligen uppmanar vi Acobiaflux att bygga en ny applikation från grunden då den befintliga applikationen är baserad på föråldrade bibliotek. Applikationen finns även för iOS vilket innebär att det finns två olika projekt, detta är en till anledning varför vi uppmanar Acobiaflux att bygga applikationen i en multiplatformmiljö.

Keywords: Android, MVVM, Kotlin, Dependency Injection, Fragments, LiveData.

Förord

Vi vill tacka vår handledare på Acobiaflux Jacob Thorsell för sin rådgivning, sitt stöd och sitt tålamod. Vi vill också tacka Acobiaflux som öppnat upp sin kontorsmiljö för oss samt hjälpt till med datorer och annat material. Vidare tackar vi också Uno Holmer för sitt tålamod och för all hjälp med rapportskrivningen. Slutligen vill vi tacka Matej Rešetár för sin tutorial 'Forecast App - Android Kotlin MVVM Tutorial' som vi lärt oss väldigt mycket av.

Jesper Skoglund och Niklas Côté, Göteborg, Maj 2019

Innehåll

Ordlista

Activity Activity är i Android, skärmen eller fönstret där det läggs in komponenter (Menyer, knappar och listor). Motsvarande Frame i Java Swing.

DAO Data Access Object, entiteten förser repository med data.

Entity Entitet, är en datastruktur för lagring av data.

JSON-fil JavaScript Object Notation, kompakt textbaserat dataformat.

MVVM Model-View-ViewModel, är ett designmönster för Androidutveckling.

Multiplatform Multiplatformmiljö är en miljö där en applikation blir exekverbar i flera olika miljöer.

Observer Observerbar, objekt som övervakar när innehåll förändras eller något sker.

PLC Programming Logic Controller, ett kontrollerbart styrsystem som styrs genom programmering.

SCADA Supervisory Control And Data Acquisition, styr- och övervakningssystem för processer.

Repository Datakatalog, innehåller vilka metoder som ViewModel kan använda sig av för att ändra och få tillgång till data.

SQLite SQL database engine, används som databasspråk i smartphones.

UI User Interface, delen av en applikation som användaren ser och använder.

XML-fil Extensible Markup Language, XML förkortat är ett filformat. I Android används XML för deklarering av element när man designar en applikations olika vyer.

1
Introduktion

Detta examensarbete utfördes i samarbete med Acobiaflux under vårterminen 2019. Företaget arbetar med helhetslösningar till företag med PLC och SCADA system, allting från installation till underhåll av system.
Acobiaflux har en desktopapplikation för hantering av deras system, de vill föra över funktionalitet från denna till deras mobilapplikationer, som till exempel statistik över utlösta alarm. Ett alarm inträffar när något med reglersystemet gått snett och behöver korrigeras, ett exempel på detta är ifall ett reglersystem för temperatur skulle visa för hög eller för låg temperatur.
Nuvarande system består utav en mobilapplikation som via ett API kommunicerar med en databas. I databasen samlas data från Acobiafluxs system, och datan vill de presentera i en graf med funktionalitet för kommentering av larm.

1.1 Syfte

Syftet med projektet är att vidareutveckla Android-applikationen, undersöka hur man sparar historisk data och hur en graf ska implementeras.
Detta innefattar även undersökning och inhämtande av kunskap om hur man bygger upp och strukturerar en Android-applikation med en tillhörande databas. Dels för att man skall kunna lämna efter sig kod som är enkel att vidareutveckla och för att inhämta kunskap om industristandarden.

1.2 Problembeskrivning

Den existerande Android-applikationen saknar eller har väldigt begränsad dokumentation. Detta gör att en stor del av arbetet var att undersöka och förstå den ärvda koden. Utöver detta skaffades även information om hur industristandarden ser ut inom Android-utveckling, så projektets kod utgår från denna trots att den ärvda koden inte följer den moderna strukturen.
Efter detta var avklarat inleddes vidareutveckling av den existerande mobilapplikationen vilket genomfördes genom att använda riktlinjerna för arkitekturen.

1.3 Mål

När projektet är avslutat är målet att Android-applikationen skall kunna visa larmstatistik i en graf, förhoppningsvis likt desktopapplikationen.
Den skall vidare vara byggd på ett sätt som gör att den är enkel att vidareutveckla och följer industristandarden.

1.3.1 Kunskapsmål

1. **Bygga kod efter industristandarden**
   Kunna följa industristandarden och förstå dess uppbyggnad.

2. **Egenbyggd applikation med liknade arkitektur som Acobiafluxs applikation**
   Utöver att vidareutveckla redan existerande mjukvara, även kunna producera en liknade applikation från grunden.

1.3.2 Mål med Android-applikationen

1. **Dokumentation av applikationen**
   Skapa dokumentation av applikationens flöde och struktur.

2. **Refaktorera den ärvda kodens struktur**
   Anpassa den efter industristandard så att den är lätthanterlig för vidareutveckling.

3. **Spara kommentarer lokalt**
   Kommentarer på kvitterade larm ska vara oberoende av applikationens livscykel, vara bestående, och följa industristandardens arkitektur.

4. **Visa historik och statistik med text**
   Visa den sparade datan som statistik i textformat.

5. **Visa historik och statistik grafiskt**
   Översätta datan från databasen till en grafisk visning såsom stapeldiagram eller andra typer av diagram.

1.4 Avgränsningar

Detta projekt kommer endast ändra den ärvda koden vad det gäller struktur, och inte funktionaliteten, enligt industristandarden. Där det är möjligt, kommer projektets kodfunktionalitet följa standarden.
Vidare kommer projektet endast rikta in sig på utveckling av den mobilapplikation som är gjord till operativsystemet Android.

2
Teknisk Bakgrund

Denna tekniska bakgrund är gjord så att läsaren skall få en klar insikt i Googles Android Architecture Components, vilket är ett ramverk som Google rekommenderar för att bygga Android-appar[1]. Detta är en del av Googles mjukvaroprodukter de rekommenderar till Android-utveckling som kallas Android Jetpack[2]. Eftersom detta projekt inte kommer fokusera på annat än kodning, kommer de andra delarna av Android Jetpack inte tas upp.

2.1 Kotlin

Kotlin är ett språk som används inom Android-utveckling, en del av detta kan vara på grund av att man uppskattar en 40 procent minskning av skriven kod i jämförelse med Java. Det är ett språk man kan skriva objektorienterat eller funktionellt, men i Android-utveckling används det objektorienterat. Detta gör också att man kan använda det vid sidan om Java-kod väldigt enkelt då Kotlin även är fullständigt kompatibelt med Java. Man kan utan svårigheter anropa metoder och klasser mellan de båda språken.
Det är väldigt likt Java i hur man programmerar i de olika språken, men ytterligare fördelar som att det är typsäkert med support för icke nullbara typer. Detta gör att man undviker många s.k 'Null Pointer Exceptions' som kan uppkomma under Java-programmering.[3]
I figuren nedan ser man ett exempel på en Kotlin klass.

```
1  class CustomerId(name: String, sex: String?, val uniqueId: Int): Inventory(){
2
3
4      private var moneyOwed: Int = 0;
5
6
7      public fun addWaresToReceipt(wareId: Int){
8
9          //function cost() from Inventory class
10         moneyOwed = moneyOwed + cost(wareId)
11
12     }
13
14
15 }
```

Figur 2.1: Kotlin exempelklass

Denna exempelklass heter 'CustomerId' och tar in tre parametrar, 'name', 'sex', 'uniqueId'. Variabeldefinitionerna fungerar som en konstruktor för klassen, när en klass av typen 'CustomerId' skapas kommer den innehålla dessa tre värden. Motsvarande i Java hade varit 'this.name = name' i konstruktormetoden. Om konstruktorn har annoteringar eller synlighetsmodifierare måste 'constructor' skrivas före parametrarna. För att programmera kod i konstruktorn i Kotlin-miljön används funktionen 'init{...}' i klassen.
Variabeln 'name' är av typen 'String', likaså 'sex', och 'uniqueId' är av typen 'Int' i exempelklassen. Variabeln 'name' saknar modifierare och är 'var' som default för Kotlins variabler och gör den till en muterbar-variabel. Variabeln 'sex' är av typen 'String?' vilket, till skillnad från 'name', gör att den kan ha ett null-värde. Som ovan nämnt är Kotlin typsäkert och om ett null-värde ges till en variabel som ej är av typen '<T>?' så ger kompilatorn ett null-variabel felmeddelande. Variabeln 'sex' saknar även modifierare och går att mutera.
Variablen 'uniqueId' har modifieraren 'val' vilket gör att den endast går att läsa av och inte modifieras. Detta kan jämföras med Javas 'final'.
Efter parametrarna återfinns klassen 'Inventory'. Som det är skrivet ärver 'CustomerId' från 'Inventory' och kan använda alla dess funktioner och synliga variabler.
I klasskroppen av koden finns en variabel, 'moneyOwed' som måste initieras direkt på grund av hur konstruktorn fungerar, metoden 'addWaresToReciept' används för att öka variabeln 'moneyOwed's värde. Modifierarna 'private' och 'public' fungerar som det gör i Java. Variabeln 'moneyOwed' är alltså endast synlig i klassen, har initialt värde 0, är en 'Int' och kan modifieras. Funktionen 'addWaresToReceipt' har parametern 'wareId' av typen 'Int', och funktionen är synlig för alla klasser som vill använda den.
I funktionskroppen finns en enkel addering av två värden som inte behöver förklaras närmare, utöver att funktionen 'cost()' är en ärvd funktion från 'Inventory'.

2.2 Dependency Injection med Dagger 2 biblioteket

Dependency Injection är något som brukar användas inom objektorienterad programmering, och det är till för att minska koppling mellan klasser samt öka kohesion. Nedan i figur 2.2 visar exempel (a) utan Dependency Injection, och (b) med Dependency Injection.

```
public class Bicycle  {

  private Frame frame;
  private Wheels wheels;

  public Bicycle (){

    frame = new Frame();
    wheels = new Wheels();

    frame.addWheels(wheels);

  }

}
```

(a) Utan di

```
public class Bicycle  {

  private Frame frame;
  private Wheels wheels;

  public Bicycle (Frame frame, Wheels wheels){

    this.frame = frame;
    this.wheels = wheels;

    frame.addWheels(wheels);

  }

}
```

(b) Med di

Figur 2.2: Dependency Injection Exemple

I (a) ses klassen 'Bicycle' gjord på ett standardiserat sätt. Dock har den hög koppling då den själv behöver skapa 'Frame' och 'Wheels' samt att den har låg kohesion då den gör mycket mer än att bara kombinera 'Frame' och 'Wheels', vilket är dess ansvar. Det som åstadkoms med Dependency Injection är att man ökar kohesionen och minskar kopplingen för klasser.
Studeras (b) finns inte längre skapandet av klasser som ansvarar för 'Bicycle', utan dessa tas in som parametrar istället. Detta är grunden för Dependency Injection, att man tar bort 'Dependency' från klasserna och använder 'Inject' så att klasserna huvudklassen behöver skapas någon annanstans. Detta ökar kohesionen, minskar kopplingen, och gör det mycket lättare att testa klasserna var för sig.
Dagger 2 gör att man behöver skriva mindre så kallad 'Boiler Plate'-kod, och gör detta genom Dependency Injection. När Dagger 2 biblioteket är importerat, krävs annonteringar så att Dagger 2 vet vilka klasser och metoder som skall importeras via Dependency Injection. Som exempel på detta finns tre klasser i figuren 2.3.

```
public class Wheels(){

  @Inject
  public Wheels(){

  }

}
```

(a) Wheels med annoteringar

```
public class Frame(){

private Wheels wheels;


@Inject
public Frame(){

}


public void addWheels(Wheels wheels){

  this.wheels = wheels;

}


}
```

(b) Frame med annoteringar

```
public class Bicycle(){

  private Wheels wheels;
  private Frame frame;

  @Inject
  public Bicycle(Wheels inputWheels, Frame inputFrame){

    wheels = inputWheels;
    frame = inputFrame;

    frame.addWheels(wheels);

  }


}
```

(c) Bicycle med annoteringar

Figur 2.3: Dagger 2 exempel på annoteringen "@Injectsom markerar klasserna för att Dependency Injection

Detta gör att Dagger 2 vet att dessa tre konstruktormetoder kommer behöva utföra 'Inject' i framtiden. För att binda ihop detta och generera Boiler Plate-kod behöver Dagger 2 en så kallad 'Component' så att den vet vad som skall genereras. En sådan Component kan ses i figuren 2.4 och kopplar samman våra klasser som visas i den tidigare figuren 2.3.

```
@Component
interface BicycleComponent{

  Bicycle getBicycle();

}
```

Figur 2.4: Dagger 2 Component exempel

Med hjälp av detta interface kan sedan Dagger 2 generera kod som då skapar alla Dependency's som behövs för att skapa ett Bicycle objekt. I tidigare figurer så har de klassernas konstruktormetoder annoterats med @Inject så då vet Dagger 2 att den skall använda dessa för att hämta Dependency's till att skapa ett Bicycle objekt. Nedan i figur 2.5 ses hur detta skulle kunna implemeteras.

```
Public class Main{


 public static void main(String[] args) {

  Wheels wheels = new Wheels();
  Frame frame = new Frame();
  Bicycle bicycle = new Bicycle(wheels, frame);

  System.out.println("Your bicycle is" + bicycle)


 }
}
```

(a) Main vanlig Dependency Injection

```
Public class Main{


 public static void main(String[] args) {

  BicycleComponent component = DaggerBicycleComponent.create();
  Bicycle bicycle = component.getBicycle();

  System.out.println("Your bicycle is" + bicycle)


 }
}
```

(b) Main med Dagger 2

Figur 2.5: Dagger 2 Main exempel

Figur (a) har traditionell Dependecy Injection förklarad tidigare, men i (b) ses hur Main kan se ut med Dagger 2. Med koden i figur 2.3 och 2.4 kommer Dagger 2 generera en klass som kallas DaggerBicycleComponent och via den skapas alla Dependencys, i detta fall Frame och Wheels. I ett så enkelt program som vårt exempel behövs inte Dagger 2 nödvändigtvis, men det är enkelt att se varför det skulle vara användbart i större projekt.

2.3 Fragments

I Androidapplikationer arbetar man ofta med ett koncept som kallas Activitys. Activitys är, simpelt föklarat, varje skärm som visas. Till exempel kan inloggningsskärmen vara en Activity och själva appen en annan. Varje Activity har sin egna så kallade 'livscykel'. Denna startas med en 'Create' och avslutats med en 'Destroy'. I föregående exempel hade inloggningsskärmen gjort en 'Create' när applikationen startas och när man loggat in en 'Destroy'. Detta är en simpliferad förklaring och det finns många andra stadier i en Activitets livscykel som 'Idle' och 'Update', men Acobiaflux använder sig primärt av fragments i deras applikation.
Ett Fragment är en del av en Activity, den har sin egen livscykel, men har Activityns livscykel som överordnad. Alltså kan Fragments livscykel bete sig annorlunda från Activitys livscykel, så länge Activitys är aktiv.
Fragments är en del av en Activity, och en Activity kan ha flera Fragments för att sköta olika delar av en skärm. Den gör detta genom att skapa sin egna View som binder upp sig till UI:et via en .xml-fil.
Det finns många metoder i Fragments som utvecklare förväntas omdefinieras när man använder sig av Fragments. De som kommer behandlas i detta projekt är *onCreateView()* och *onCreateActivity()*. Den förstnämnda används för att binda samman en View med Fragmentet genom att först skapa en View, binda den till XML-filen som används av Fragment, och sen returnerar metoden som skapades i View. Den andra metoden är till för att modifiera UI:et i Fragmentets View. Som till exempel uppdatera en lista via LivaData eller fylla på data i en tabell. [4]
Specifikt för Acobiaflux applikation är att den använder sig av dependency injection för att samla alla fragments i en Activity, så alla Fragments är beroende av samma överordnade livscykel. Den kontrollerar dessa Fragments via en Fragmentmanager som implementeras och injiceras i en överordnad klass.

2.4 LiveData

LiveData är ett bibliotek som assisterar Observers genom att informera när data ändrats, då uppdateras Observers för att effektivt bara uppdatera applikationens UI när data ändrats. LiveData är en observerbar klass som skiljer sig från andra observerbara klasser genom att LiveData är livscykelmedveten vilket betyder att när en Activity försvinner då försvinner även LiveData-objektet för Activityn. LiveData används för att se till att data i applikationsens UI ska vara samma data som i databasen. Uppdatering av UI:et är beroende av livscykeln, uppdatering av UI:et sker automatiskt och eftersom LiveData är livscykelmedveten så hanteras minnet effektivt då objekten försvinner när Activityn gör det.[5]

2.5 Tabeller med biblioteket MPAndroidChart

MPAndroidChart är ett bibliotek som finns för många plattformar, inkluderat Android. För att skapa och integrera en tabell i en Android-applikation behöver man först,

utöver att importera biblioteket i önskat projekt, skapa en XML-fil innehållande ett så kallad Chart-objekt för att definiera hur stor plats av skärmen detta objekt skall ta och vilken typ det är. MPAndroidChart innehåller många typer av tabeller, bland annat 'BarChart', 'PieChart', 'LineChart' för att nämna några. Då i detta projekt endast 'BarChart' kommer användas fokuseras exemplet på detta. I figur 2.6 visas hur ett 'BarChart' objekt kan implementeras i en XML-fil och hur det implementeras i detta projekt.

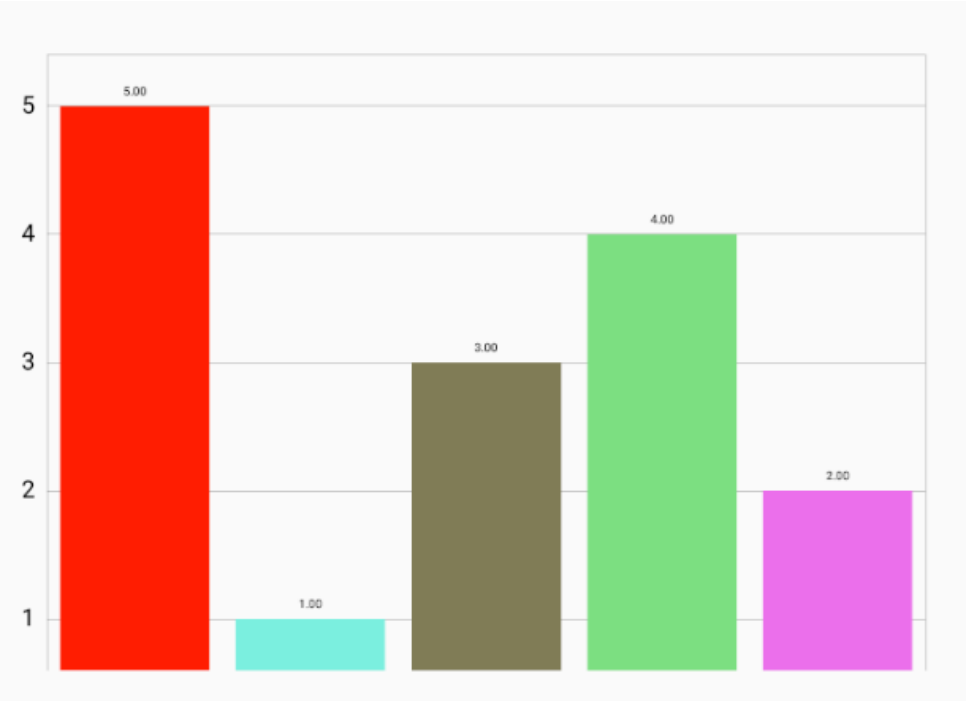
```
<com.github.mikephil.charting.charts.BarChart
            android:id="@+id/statistics_chart"
            android:layout_width="match_parent"
            android:layout_height="285dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:layout_marginEnd="8dp"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
```

Figur 2.6: BarChart i XML-fil med ConstraintLayout som överordnad layout

Efter detta skall objektet i XML-filen bindas ihop med en View som visas i en Activity. I detta projekt används Fragments, så själva bindandet sker i metoden *onCreateView()*. Alla inställningar för att fylla tabellen och hur datan visas, bestäms i metoden *onCreateActivity()*.
I detta projekt används LiveData och därför observeras databasens innehåll i denna metod och tabellen uppdateras därefter. Utöver att lägga in data i tabellen, bestäms bland annat inställningar som färger på staplarna i tabellen, hur animationer görs, vad x- och y-axel visar för information. I figur 2.7 visas hur en tabell skapad på ovan beskrivet sätt se ut.

2.6 Listvy med RecycleView biblioteket

Precis som i föregående avsnitt med MPAndroidchart importeras biblioteket till projektet och därefter behövs ett objekt av typen RecyclerView, i en XML-fil. Det som är annorlunda är att ytterligare en XML-fil skall skapas för varje rad i listan. Denna individuella rad kommer att fyllas via en RecyclerViewAdapter och sedan samordnas till en lista. Denna samordnade lista binds sedan ihop med den förstnämnda XML-filen för att visa listan. I figuren 2.8 ses hur en rad i listan är gjord i detta projekt.
På samma sätt som med MPAndroidChart binds RecycleView ihop med XML-filen i *onCreateView()* och fylls på med data i *onCreateActivity()*, som uppdateras med LiveData på liknade sätt som MPAndroidChart. Det som skiljer de åt är att det krävs en RecycleViewAdapter för att binda ihop de ovan nämnda raderna med listan. Den-



Figur 2.7: BarChart, exempel på utseende för ett diagram

na funktionalitet inträffar i den sistnämnda metoden och en ny RecycleViewAdapter skapas för att sammanfatta datan från LiveData till rader i en lista, som sedan binds till ovannämnda RecycleView. Övrig funktionalitet som till exempel ActionEvents och animationer implementeras via *onCreateView()*.
I figur 2.9 visas hur listvyn med RecycleView ser ut i projektets applikation.

2.7 Databasarkitektur

Här beskrivs databasarkitekturen för vad Android-applikationen innehåller, hur 'Room Persistence' används för att skapa en lokal databas i applikationen, fördelar med att använda MVVM strukturen och beskrivning av MVVMs delar och hur de används.

2.7.1 Databas med 'Room Persistence'

'Room Persistence' använder sig av följande komponenter vid skapandet av en lokal databas.

- Entitet, en entitet är ett databasobjekt som används för att lagra data. I Android med biblioteket 'Room Persistence' är en entitet en dataklass, när klassen läggs till i databasen skapas en tabell med motsvarande innehåll.
- DAO, ett DAO är ett Data Access Object som används för att anropa funktioner som gör operationer på databasen utan att avslöja innehållet i databasen. Ett DAO förser Repository med data från databasen.
- Repository, ett repository hämtar data från databasen via DAOet och förser

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <ImageView
        android:id="@+id/statistics_recyclerview_color"
        android:layout_width="62dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:foreground= "@drawable/rounded_coners_imageview"
        android:scaleType="fitStart" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="2dp"
        android:layout_marginStart="4dp"
        android:layout_weight="1"
        android:orientation="vertical">
```

```
        <TextView
            android:id="@+id/statistics_recyclerview_alarm"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/statistics_recyclerview_comment"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textStyle="italic"/>

        <TextView
            android:id="@+id/statistics_recyclerview_frequency"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>

</LinearLayout>
```

Figur 2.8: Exempel på en XML-filen för en rad i RecycleView

ViewModel med datan så att innehållet i UIt är samma som databasen.

Acobiaflux använder sig av Microsofts tjänst 'Azure' för att ha en molnbaserad databas där all data lagras. För att använda datan i olika applikationer använder Acobiaflux sig av ett API[6]. API:et gör om databasens data till JSON-filer, Android applikationen hanterar JSON-filerna genom att automatiskt skapa entity, DAO och repository. 'Room Persistence' används för att spara data lokalt i applikationen samtidigt som den utnyttjar SQLite effektivt[7].

Alla entiteter och DAO kopplas ihop i en databasklass med hjälp av biblioteket 'Room Persistence' som är en del av 'Android Architecture Components'.

I figur 2.10 visas databasen som kopplar ihop entiteterna och lägger till metoder för användning av DAO.

Figur 2.9: Listvy skapad med RecycleView

```
@Database(
        entities = {Tag.class, Building.class, Floor.class, Rooms.class,
                Alarm.class, TagTime.class, StatisticsComment.class},
        version = 1,
        exportSchema = false)
@TypeConverters(Converters.class)
public abstract class AXBuildingManagmentDB extends RoomDatabase{

    abstract public TagDAO tagDAO();
    abstract public BuildingDAO buildingDAO();
    abstract public FloorDAO floorDAO();
    abstract public RoomDAO roomDAO();
    abstract public TagTimeDAO tagTimeDAO();
    abstract public AlarmDAO alarmDAO();

    //Thesis workers
    abstract public StatisticsDAO statisticsDAO();
}
```

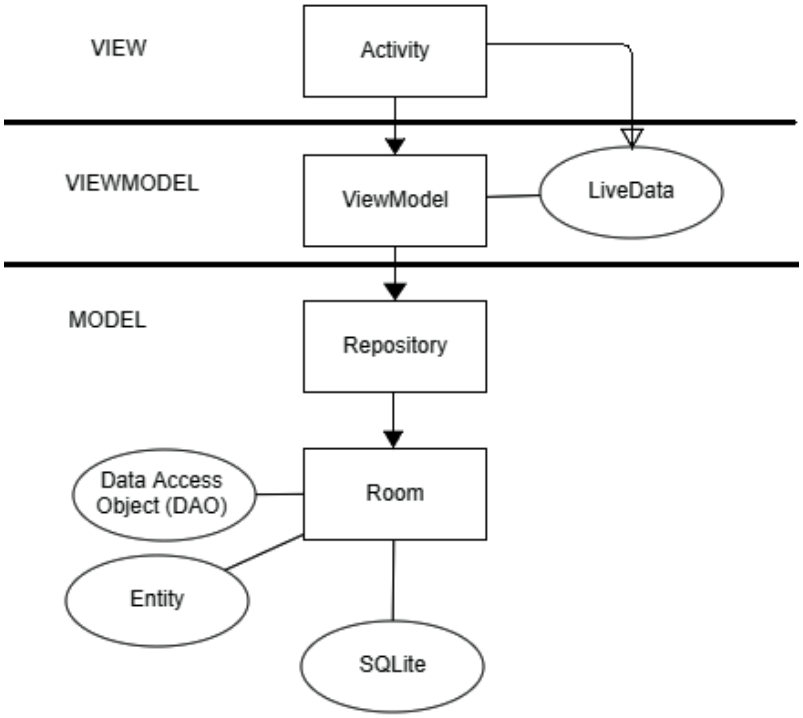
Figur 2.10: Databasen, sätts upp i en klass som utnyttjar biblioteket 'Room Persistence' för att snabbt sätta upp en databas.

2.7.2 Designmönstret - MVVM

MVVM står för Model-View-ViewModel och är ett designmönster Android rekommenderar och som projektet utgått ifrån. Designmönstret innebär att man använder tre lager (Model-View-ViewModel) som hanterar en uppgift vardera och ska fungera oberoende av de andra lagren, fördelar med detta är att ändringar i ett lager inte innebär ändringar i de andra lagren.

- Model - Grundlagret är Model vilket hanterar databasentiteterna (Entity), vilka metoder som kan användas mot datan (DAO) och ett skyddande lager för metoderna mot datan (Repository).
- View - Applikationens UI använder ViewModel för att få tillgång till datan som finns i databasen. Datan som ändras av användaren i View-delen lagras i ViewModel-delen som skickar sedan uppdateringen av datan till Model-delen[5].
- ViewModel - Lagret som hämtar datan från Model kallas ViewModel och används för att visa data för View-delen. Datan som är lagrad i ViewModel är livscykel baserad för att datan alltid ska vara relevant och beständig även utan uppkoppling mot molndatabasen. LiveData används i ViewModel för att få notifikationer när databasens innehåll ändras vilket gör att ViewModel uppdaterar datan den tillhandahåller för View-delen[8].

I figuren 2.11 visas strukturen för MVVM.



Figur 2.11: MVVM strukturen, Model står för data hanteringen och innehåller databasen. View-Model fungerar som lager mellan Model och View och innehåller LiveData som observerar när data ändras och uppdaterar både Model och View. View är gränssnittet som användaren använder.

3
Metod

Projektet inleddes av en planeringsfas där det planerades hur projektets mål ska uppnås effektivt. En tidsplan gjordes i form av ett Gantt-schema för att enkelt kunna följa och strukturera upp arbetsgången.

Projektet började med en undersökningsfas för att få en uppfattning hur systemet fungerar samt hur systemets delar påverkar varandra. Delarna som undersöktes var en applikationsdel och en API-del. Under arbetets gång fokuserades det mest på applikationsdelen. Undersökningsfasen fortsatte med att se tutorials om Android-utveckling enligt industristandard för att skapa en djupare förståelse för hur applikationen bör struktureras.

3.1 Planering

Gantt-schemat består i huvudsak av de tre delarna Undersökning, Utveckling och Rapportskrivning. Se Gantt-schemat i Bilaga A.1.

Projektet inleddes med en undersökningsperiod för att skapa djupare förståelse av projektets alla delar och identifiera vilka problem som kan dyka upp och hur de ska lösas för att hitta den bästa vägen mot målet. Parallellt med att kunskapen ökar under undersökningsperioden skrevs innehållet ner i rapporten, detta är rapportskrivningsperioden som börjar i slutet på undersökningsperioden och i början av utvecklingsperioden. Utvecklingsperioden pågick parallellt med rapportskrivningsperioden efter att undersökningsperioden avslutades.

3.2 Programmeringsmiljö och Versionshantering

Programmeringsmiljön var Android Studio, utvecklingsspråket var en blandning mellan Java och Kotlin och även SQLite. Versionshantering behandlades i Git.

3.3 Tutorials

Tutorialen vi gjorde var implementation av en väderapplikation i Android som fokuserar på programmeringsspråket Kotlin samt ger mycket information om hur en modern applikation skall struktureras [9].

Tutorials användes för att förstå strukturen i moderna applikationer och för att förstå de olika koncept projektets ärvda applikation använt sig av samt för att få en bredare inblick i hur en generell modern Android applikation skall struktureras. Tutorialen 'Forecast App - Android Kotlin MVVM Tutorial Course' gav en djupare förståelse för vad projektets applikation skulle behöva uppfylla [9].

3.4 Implementering

Implementeringen av projektet började med att strukturera upp en lättarbetad mapp-struktur liknande den Tutorialen använde sig av. Sedan implementerades en lokal databas för att spara kommentarer på larmkvitteringar. Nästa steg var att skapa en statistisk vy på gamla larm för analys av orsaker till att larmen uppstår.

3.4.1 Implementering av Mapp-strukturen

Mapp-strukturen implementerades enligt MVVM designmönstret som vi lärde oss genom tutorialen. Mapp-strukturen ändrades genom att flytta klasserna till mapparna de hörde till. Data-mappen innehåller bara databasklassen som sätts upp med Room Persistence, entitetklasserna, DAO:erna och Repositorys för varje entitet. DI-mappen innehåller bara Depenency Injection moduler och komponenter. UI-mappen innehåller bara Fragments och ViewModels. Detta behövdes för att kunna bygga vidare på projektet och även göra det lättare att bygga vidare på i framtiden.

3.4.2 Implementering av Databasen

Databasen implementerades genom att följa Android Developers guide för Room Persistence. Först skapade vi en entitet för statistik som innehåller id, kommentar och antal gånger larmet inträffat. Därefter implementerade vi ett DAO för att hämta och sätta datan som ska finnas i entiteten med anrop som sätta alla kommentarer, hämta alla kommentarer och hämta antal larm som inträffat. Till sist sätts entiteterna och DAO:erna upp i en databasklass som är applikationens lokala databas. [10]

3.4.3 Implementering av den Statistiska vyn

Det finns många delar som behöver behandlas för att kunna implementera den statistiska vyn.

3.4.3.1 Navigering till vyn

I Acobiaflux applikation finns en menu-bar som ska användas för navigering. Applikationens menu-bar behövde uppdateras med en ny knapp som vid tryck öppnar ett fragment till den statistiska vyn.

3.4.3.2 Model

Ett repository som kommunicerar med databasen ska implementeras. Detta repository bör innehålla metoder som visar observerbar data och kan uppdatera datan också. Vyn kommer ha tillgång till dessa via View-Model.

3.4.3.3 ViewModel

View-Model kommunicerar direkt med Model och ViewModelns metoder kommer användas av vyn. Denna klass kommer inte vara speciellt stor, men har en viktig funktion för att minska kopplingen och öka kohesionen i MVVM strukturen. Den kommer använda samma Dependency Injection moduler som finns i koden, så dessa kommer behövas uppdateras.

3.4.3.4 View

View kommer representera det man faktiskt ser och UI:et. Viewn är även klassen som behöver ärva från Fragments så att det går att navigera till den. Det är även den klass som anropas vid tryck på knappen beskriven i stycket 'Navigering till vyn'. Klassen kommer kommunicera mycket med den existerande koden och behöver finnas som referens i många moduler för Dependency Injection.
Denna klass kommer behöva sin egna XML-fil som beskriver hur den skall se ut. XML-filen kommer refereras i vy-klassen så de tillsammans kan skapa det användaren ser på skärmen. Vad det gäller den visuella delen av Statistiken kommer MPAndroidChart användas. Det kommer ha på sin y-axel, heltal som representerar hur ofta ett alarm har utlösts. Varje stapel kommer ha unika färger för att koppla samman det visuella och textbaserade.
Den textbaserade delen kommer visas med en RecycleView. Varje listobjekt i RecycleViewn kommer innehålla tre rader text, alarmets namn, alarmets kommentar och hur många gånger larmet har utlösts. Det kommer även finnas en cirkel med en unik färg för att enkelt kunna koppla samman en stapel i BarCharten till listobjektet. När man trycker på ett listobjekt kommer en textruta upp där man kan uppdatera alarmets kommentar.

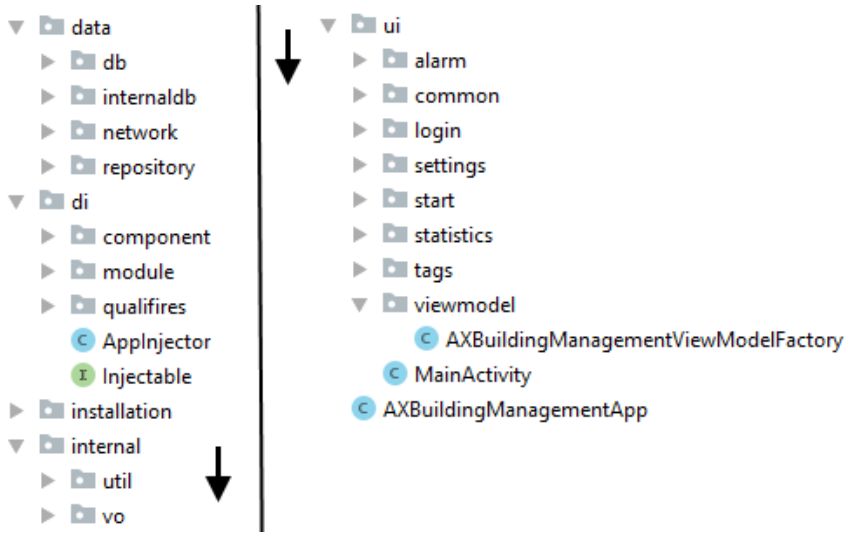
3.5 Utvärdering

Utvärdering av projektet skall göras genom kommunikation med Acobiaflux efter deras förväntningar. Arbetet skall utföras på Acobiafluxs kontor där målen följs upp iterativt under projektets gång.
Utvärderingen görs på dokumentationen, applikationens funktionalitet och den statistiska vyn. Acobiaflux skall gå genom vad som utförs tillsammans med oss och ser till att de mål som satts upp följs upp i slutet av projektet. Mål som utvärderas är kunskapsmål, dokumentationen, funktionaliteten av kommentering av larm och den statistiska vyn.

4

Systemkonstruktion

4.1 Refaktorering av kod



Figur 4.1: Den refaktoriserade mappstrukturen i sin helhet

Den ärvda koden led av dålig mappstruktur. Många klasser var blandade med varandra på ett sätt som inte var logiskt enligt någon kommersiell arkitektur. Innan implementeringensarbetet inleddes en refaktorering av koden. Denna refaktorering fokuserade främst på att styra upp mapp-strukturen så att den följde MVVM på samma sätt som Google gjort i sina egna kurser [11].
Detta innebär en uppdeling av Model/Repository, View/ViewModel, och interna hjälpklasser som används av hela appen. Så projektet utgick från dessa tre grundmappar, **ui**(View/ViewModel), **internal**(hjälpklasser för hela projektet), och **data**(Model/Repository). Dock blev denna struktur inte tillräcklig och ytterligare två mappar adderades till grundstrukturen, **installation**(installations klasser), och **di**(dependency injection).
Undermappar till **data** är *db*(Database) som innehåller allt relevent för databasens uppbyggnad. *internaldb*(Internal Database), som är databashantering för applikationen och tas upp i avsnitt 4.2. Mappen *network* innehåller all kod som hanterar nätverkskommunikationen. *repository* innehåller alla repositories som är en del av MVVM, exempel i figur 2.11.
Mappen **di** har undermappar *component*, *module*, och *qualifiers*. Vad dessa innehåller och ansvarar för tas upp i avsnitt 2.2.

Vidare har mappen **internal** undermappar *util* med allmänna hjälpklasser som till exempel trådhantering, och *vo*(Value Objects) som är enkla klasser som endast sparar data. **ui** har undermappar för varje View, och en mapp, *common*, med klasser som används av alla views, inklusive NavigationController.

4.2 Tillägg av en ny databastabell

Projektets databas, är en vidareutveckling av den befintliga genom ett tillägg av en databastabell som ska hantera kommentarer på larm.

För att skapa en ny databastabell behövs en entitet för vad tabellen ska innehålla, ett DAO och ett repository samt behöver entitetens klass och en funktion för DAO:et läggas till i databasen. Uppbyggnaden består av fyra element för att implementera en ny tabell i en applikations lokala databas i Andriod-miljön. Tabellen ska vara backenddelen i projektet som hanterar utdelning och hantering av datan som applikationen ska uppvisa.

De fyra elementen för tillägg av en tabell i en lokal databas:

1. En dataklass för innehållet av entiteten, hur datan ska sparas och hur datan ska hanteras. En entitet ska innehålla:
   - Variabler, vad tabellen innehåller.
   - Nycklar, primärnycklar och främmande nycklar som hämtas från andra entiteter.

   Nedan i Figur 4.2 visas en entitet över hur en dataklass kan se ut.

```
package com.acobiaflux.axbuildingmanagement.data.internaldb

import android.arch.persistence.room.Entity

@Entity(primaryKeys = arrayOf("commentId"))
data class StatisticsComment(val commentId: String, val commentText: String)
```

Figur 4.2: Entitet, ska vara en tabell för lagring av data, i detta fallet är det entiteten för hantering av kommentarer på larm.

2. Ett Data Access Object är ett interface som agerar som ett skyddande lager mellan användare och databastabellen, i interfacet implementeras endast de metoder som är nödvändiga.
   Nedan i Figur 4.3 visas hur projektets Data Access Object ser ut för entiteten projektet skapat.

```
import android.arch.lifecycle.LiveData
import android.arch.persistence.room.*


 @Dao
interface StatisticsDAO{

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertComment(commentText: List<StatisticsComment> )

    @Query( value: "SELECT * FROM statisticscomment")
    fun getComments(): LiveData<List<StatisticsComment>>

    @Delete
    fun delete(commentText: StatisticsComment)
}
```

Figur 4.3: Data Access Object, skapar ett interface för vilka operationer en användare ska kunna använda på datan i entiteten.

3. Ett Repository använder funktionaliteten som implementeras i DAO:et och injicerar operationerna till de existerande instanserna av klasserna StatisticsDao och AppExecutors.

   I figur 4.4 visas hur entitetens Repository ser ut.

```
import android.arch.lifecycle.LiveData
import com.acobiaflux.axbuildingmanagement.internal.util.AppExecutors
import javax.inject.Inject

class StatisticsRepository @Inject constructor(
        val statisticsDAO: StatisticsDAO,
        val appExecutors: AppExecutors
) {

    fun getComments():LiveData<List<StatisticsComment>>{
        return statisticsDAO.getComments()
    }

    fun insert(statisticsComment: List<StatisticsComment>){
        appExecutors.diskIO().execute{
            statisticsDAO.insertComment(statisticsComment)
        }
    }

    fun delete(statisticsComment: StatisticsComment){
        appExecutors.diskIO().execute{
            statisticsDAO.delete(statisticsComment)
        }
    }

}
```

Figur 4.4: Repository, den använder metoderna i StatisticsDAO för att ändra datan som är sparad i databasen.

4. Databasen sätter upp funktionalitet så att entiteten blir en tabell i databasen och skapar en metod för DAO för att hämta data från klasserna.
   Nedan i Figur 4.5 visas databasen som innehåller entiteterna med tillägget StatisticComment och metoder för DAO för entiteterna.

```
@Database(
        entities = {Tag.class, Building.class, Floor.class, Rooms.class,
                Alarm.class, TagTime.class, StatisticsComment.class},
        version = 1,
        exportSchema = false)
@TypeConverters(Converters.class)
public abstract class AXBuildingManagmentDB extends RoomDatabase{

    abstract public TagDAO tagDAO();
    abstract public BuildingDAO buildingDAO();
    abstract public FloorDAO floorDAO();
    abstract public RoomDAO roomDAO();
    abstract public TagTimeDAO tagTimeDAO();
    abstract public AlarmDAO alarmDAO();

    //Thesis workers
    abstract public StatisticsDAO statisticsDAO();
}
```

Figur 4.5: Databasen, sätts upp i en klass som utnyttjar biblioteket Room för att snabbt sätta upp en databas.

4.3 Skapandet av View och ViewModel

I detta avsnitt beskrivs de olika delarna som behövdes implementeras för att grafiskt kunna visa och navigera till en vy med statistik.

4.3.1 Navigation till Statistics

Längst ned i Acobiafluxs applikation finns en menyrad med fyra olika knappar. Med dessa navigerar användaren till applikationens fyra huvudmenyer; **Home**, **Alarm**, **Statistics**, och **Settings**. I detta projekt har endast utveckling gällande **Statistics** behandlats.
Först ändrades den existerande XML-filen(*bottom_navigation_menu*) som behandlade menyraden för att inkludera ytterligare en knapp. Detta inkluderar en referens till en vectorbild som finns i Androids standardbibliotek, en id-tag så att denna knapp kan refereras i resterande applikation och övriga kosmetiska inställningar.
För att denna knapp skall vara användbar modifierades koden som behandlar menyraden. Först ändrades metoden *onTabSelected()*, som innehåller en switch-sats där de olika fragmenten startar beroende på knapp tryckt på menyraden, så att den innehöll Statistik-knappen. Efter detta adderades metoden *navigateToStatistics()* till ovannämnda klass och denna metod, baserad på existerande kod i applikationen för liknande metoder, stänger ned aktuell fragment och startar Statistik-fragmentet. Efter detta modifierades metoden *onNavigationItemSelected()* i klassen *MainActivity* för att använda metoden *navigateToStatistics()* genom att länka metoden till den grafiska knappen i XML-filen *bottom_navigation_menu*.

4.3.2 StatisticsViewHolder

En ViewHolder har ansvar för att koppla samman Repository och View, i denna applikation är detta klasserna **StatisticsRepository** och **StatisticsFragment**. Statistikfragmentets ViewHolder, **StatisticsViewHolder**, gör detta genom att använda metoderna från Repository, som injicerar denna med Dagger 2, och ViewHoldern gör de metoderna tillgängliga för View.
Alla ViewHolders ärver från klassen *ViewHolder()*, och det gör även **StatisticsViewHolder**. Klassen används av *ViewModelModule* som via Dagger 2 binder ihop alla ViewModels i applikationen med den generella klassen *AXBuildingManagementViewModelFactory* som är ViewModelFactory för hela applikationen.
Två metoder finns i ViewHolder och dessa är *getComments()* och *updateSingleComment()*. Den första metoden hämtar en lista av typen LiveData som innehåller entiteter av typen StatisticsComment. Den andra metoden uppdaterar en StatisticsComment, istället för att uppdatera hela listan. Behov av andra metoder fanns inte i detta projekt.

4.3.3 StatisticsFragment

StatisticsFragment ärver från **Fragments**, som beskrivs närmare i avsnittet 2.3. Den implementerar även ett gränssnitt som markerar för Dagger 2 att man vill injicera den i en annan del av applikationen. All funktionalitet och implementation med Dagger 2 sker i mappen **di** som nämns i 4.1.
StatisticsFragment innehåller flera privata instanser av klasser och variabler som används av många olika metoder i klassen. Dessa visas i figuren 4.6 nedan och förklaras när metoderna beskrivs i underrubrikerna 4.3.3.1 och 4.3.3.2.

```
@Inject
    ViewModelProvider.Factory viewModelFactory;

    private StatisticsViewModel statisticsViewModel;

    private BarChart chart;
    private BarData BARDATA;
    private RecyclerView statisticsRecycleView;
    private StatisticsRecyclerViewAdapter statisticsRecyclerViewAdapter;
    private RandomColorGenerator randomColorGenerator;
    private int[] randomColors;

```

Figur 4.6: Privata klasser och variabler

4.3.3.1 *onCreateView()*

Hur denna metod fungerar nämns närmare avsnittet 2.3. Det första metoden gör är att skapa en **RandomColorGenerator**. Detta är en hjälpklass skapad under projektens gång som tar in ett positivt heltal och returnerar ett fält av heltal som motsvarar färger. Om man anropar metoden *randomColors()* i klassen med in parametern 5 får man ett fält innehållandes 5 slumpmässiga färger tillbaka. Efter detta skapar metoden en **View** och binder ihop denna med relevant XML-fil.
BarChart och **RecycleView** initieras efter detta och binds ihop med relevanta XML-objekt som finns i filen som bundits ihop med **View**. **RecycleView** binds ihop med en Layoutmanager och andra saker som hur rader skall presenteras och om animation skall aktiveras bestäms här också.
Metoden returnerar en **View** som används av **MainActivity** och övriga metoder i klassen **StatisticsFragment**.

4.3.3.2 *onActivityCreated()*

Som nämnt i avsnittet 2.3, används *onActivityCreated()* för att fylla och uppdatera data i en **View**. Det första som händer är att man hämtar **StatisticsViewModel** via Dependency Injection så att man kan använda dess metoder för att kommunicera med databasen. Eftersom metoden *getComments()* är en lista av typen LiveData behöver man observera klassen **StatisticsViewModel** efter förändringar. Metoden *onChanged()* skrivs över för att observera förändringar i listan över StatisticsComments så att den fyller **BarChart** och **RecycleView** med data. I figur 4.7 visas hur detta görs utan lambda uttryck.

```
statisticsViewModel.getComments().observe(this, new Observer<List<StatisticsComment>>() {
            @Override
            public void onChanged(@Nullable List<StatisticsComment> statisticsComments) {
                fillBarChart(chart, statisticsComments);
                statisticsRecyclerViewAdapter = new StatisticsRecyclerViewAdapter(getActivity().getBaseContext());
                fillRecyclerView(statisticsRecyclerViewAdapter, statisticsComments);
            }
```

Figur 4.7: Hur **StatisticsFragment** observerar förändringar i databasen

Först anropas hjälpmetoden *fillBarChart()* som fyller *BarChart* med data men också bestämmer animationer och specialinställningar. I övrigt anropas metoden *randomColors()* från klassen **RandomColorGenerator** och returnerat värde sparas i arrayen randomColors som är synligt för alla metoder i **StatisticsFragment**. Efter detta skapas ett objekt av **StatisticsRecyclerViewAdapter** som ärver av klassen **RecycleViewAdapter**. Slutligen anropas hjälpmetoden *fillRecyclerView()* som fyller upp **RecycleView** med data med hjälp av **StatisticsRecyclerViewAdapter**. Den använder samma färger som **BarChart** eftersom den använder fältet randomColors. Andra viktiga funktioner som denna hjälpmetod har ansvar för är bland annat animationer och att göra varje rad 'klickbar'.
Varje gång databastabellen, som innehåller StatisticsComment, uppdateras så kommer en ny adapter skapas, och hjälpmetoderna anropas på nytt.

5
Resultat

Sammanfattningsvis har projektet lett till en uppgradering av Acobiafluxs Android-applikation med ny funktionalitet såsom kommentarshantering av larm och ett statistiskt gränssnitt för larmen så användaren snabbt ska få en överblick.

5.1 Resultat

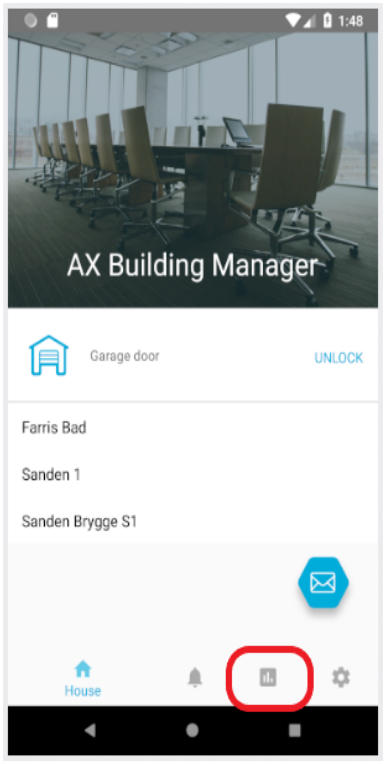
En fullt fungerande uppgradering av applikationen har skapats som använder moderna komponenter för applikationsutveckling i Android. Dokumentation av alla komponenter har skapats genom denna rapport som går igenom komponenterna som använts.

5.1.1 Lista över komponenter som dokumenterats:

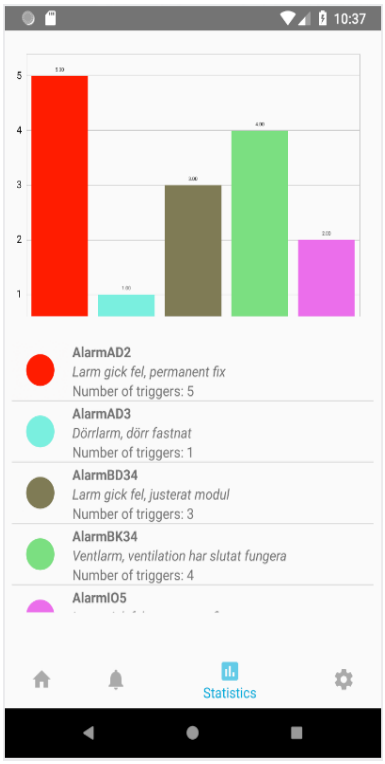
- Kotlin (Modernt programmeringsspråk för Android-utveckling)
- Dependency Injection ('Dagger 2')
- MVVM struktur (Model View ViewModel)
- Lokal databastabell oberoende av uppkoppling (LiveData, Entity, Data Access Object, Repository)
- Grafiskt gränssnitt som är uppdateras vid förändringar i databastabellen (Fragment, View, ViewModel)

5.1.2 Ny funktionalitet som lagts till:

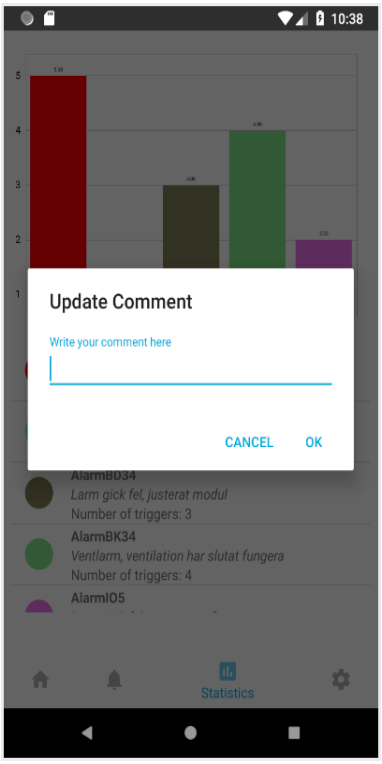
- Ny menyknapp för den statistiska vyn, visas i Figur 5.1.
- Statistisk/Grafisk vy som demonstreras i Figur 5.2.
- Lista över larm kan ses i Figur 5.2, innehåller larm-id, kommentarer och antal gånger larmet inträffat.
- Kommentering av larm visas i Figur 5.3, vid redigering av kommentar till varje larm bemöts användaren av denna vy.



Figur 5.1: Markering av menyknapp projektet lagt till.



Figur 5.2: Statistisk/Grafisk vy, visar statistik över larm med kommentarer och antal gånger larmet inträffat.



Figur 5.3: Redigering av kommentar kopplat till larm.

6
Diskussion

Detta avsnitt kommer utvärdera metoden och resultatet för att komma fram till eventuella förbättringar ifall man skulle göra om projektet. Diskussionen kommer även innehålla planerna för framtiden av applikationen.

6.1 Utvärdering av metoden

Metoden vi använde oss av var inte optimal, i början av projektet försökte vi först skapa en förståelse för flödet mellan applikationen, API:et och databasen i molnet. Vi borde fokuserat mer på applikationsdelen som företaget ville utveckla, men vi kände ändå att vi behövde en överblick över hur datan överfördes mellan de olika delarna.

Om vi skulle göra om projektet var vi eniga om att vi inte skulle undersöka flödet mellan applikationen, API:et och databasen så mycket som vi gjorde, utan tidigare börja undersöka hur man ska göra en applikation på rätt sätt genom att följa moderna tutorials.

6.2 Utvärdering av resultatet

Resultatet vi skapade är Acobiaflux och vi nöjda med och kommer komma till stor nytta för företaget om de vill fortsätta utveckla Androidapplikationen eller göra en annan Androidapplikation från början.

De mål som vi satte upp uppnås med vår egen kod, men det ärvda projektets kod hade vi stora svårigheter att koppla ihop med vår kod. Därför bestämde vi oss för att skapa en egen databastabell som vi fyller på med hårdkodad data. Den hårdkodade datan ska simulera den riktiga datan för att visa hur de olika komponenterna ska kopplas samman i modernare Androidapplikationer på rätt sätt.

6.3 Slutsats

Vår slutsats baseras på all tid vi lagt ner på förståelsen av projektet, vi anser att applikationen bör göras om med undantag för designen. Androidapplikationen vi vidareutvecklade är ett projekt som använt komponenter som snart blir föråldrade, därför skulle vi föreslå Acobiaflux att börja om från början med projektet men behålla UI designen. Komponenterna i projektet har hög koppling och låg kohesion vilket gör applikationen komplex och inte hållbar att bygga vidare på, vi insåg detta sent i projektet vilket gjorde att vi inte utvecklade en ny applikation från början. Om vi hade gjort en ny applikation från början hade vi strukturerat applikationen på samma sätt som vi byggde vår funktionalitet i den befintliga applikationen. Vi skulle även rekommendera Acobiaflux att bygga den nya applikationen i en multiplatform miljö för att undvika underhåll av mer än ett projekt för samma applikation.

Sammanfattningsvis har vi visat hur man ska bygga en Androidapplikation enligt tidsenlig standard, dokumenterat vårt arbete så det kan uppföljas och slutligen uppmanat Acobiaflux att göra om applikationen i en multiplatform miljö.

6.4 Mål som uppnåtts

Här nedan dokumenterar vi och utvärderar de mål som presenterats i introduktionen

1. **Bygga kod efter industristandarden**
   Under projektets gång har kunskaper inom dessa ämnet införskaffats via dokumentation och texter som presenterats av olika källor som nämnds i rapporten. Exempel på detta är att använda sig av MVVM-strukturen och Dependency Injection[11].

2. **Egenbyggd applikation med liknade arkitektur som Acobiafluxs applikation**
   Efter en tutorial om Androidapplikationsbyggande har projektet producerat en applikation som följer en liknade arkitektur som Acobiafluxs applikation. Detta anses ha uppfyllt detta mål.

3. **Dokumentation av applikationen**
   Projektet har gjort stora framsteg med dokumentation av applikationen, tidigare fanns ingen dokumentation. Dokumentation har gjorts på applikationsflödet, strukturen och hur alla delar av applikation, som behandlats under projektet, hör ihop.

4. **Refaktorera den ärvda kodens struktur**
   Projektet har omstrukturerat den ärvda koden enligt MVVM strukturen. Vilket är bra för vidareutveckling av applikationen inför framtiden.

5. **Spara kommentarer lokalt**
   Kommentarer går nu att skapa, ändra och spara i applikationens lokala databas. Detta är användbart för det gör att användaren kan se vad som tidigare har hänt med kommentar även utan uppkoppling.

6. **Visa historik och statistik med text**
   En lista har skapats där man kan se hur ofta alla larm inträffat. Detta innebär att användaren snabbt kan få en överblick över vad som händer.

7. **Visa historik och statistik grafiskt**
   Ett grafiskt gränssnitt har skapats för att se hur ofta ett larm inträffat.

Litteraturförteckning

[1] Google.com. (2019, Apr.) Android architecture components. [Online]. Available: https://developer.android.com/topic/libraries/architecture/[Accessed: 2019-04-10]

[2] ——. (2019, Apr.) Android jetpack. [Online]. Available: https://developer.android.com/jetpack[Accessed:2019-04-10]

[3] Kotlinlang.org. (2019, Apr.) Kotlin. [Online]. Available: https://kotlinlang.org/docs/reference/faq.html[Accessed:2019-04-10]

[4] Google.com. (2019, May) Fragment. [Online]. Available: https://developer.android.com/guide/components/fragmentsl[Accessed:2019-05-08]

[5] ——. (2019, Apr.) lifecycle, livedata. [Online]. Available: https://developer.android.com/topic/libraries/architecture/livedata[Accessed:2019-04-10]

[6] Microsoft.com. (2019, Apr.) Cloud service. [Online]. Available: https://azure.microsoft.com/sv-se/[Accessed:2019-04-18]

[7] Google.com. (2019, Apr.) Room persistence library. [Online]. Available: https://developer.android.com/topic/libraries/architecture/room[Accessed: 2019-04-10]

[8] ——. (2019, Apr.) Viewmodel. [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel[Accessed:2019-04-29]

[9] Resocoder.com. (2019, Feb.) Forecast app - android kotlin mvvm tutorial. [Online]. Available: https://resocoder.com/2019/02/02/android-kotlin-forecast-app-14-detail-of-future-weather-mvvm-tutorial-course/ [Accessed:2019-04-10]

[10] Google.com. (2019, Jun.) Room persistence library. [Online]. Available: https://developer.android.com/training/data-storage/room/index.html[Accessed:2019-06-14]

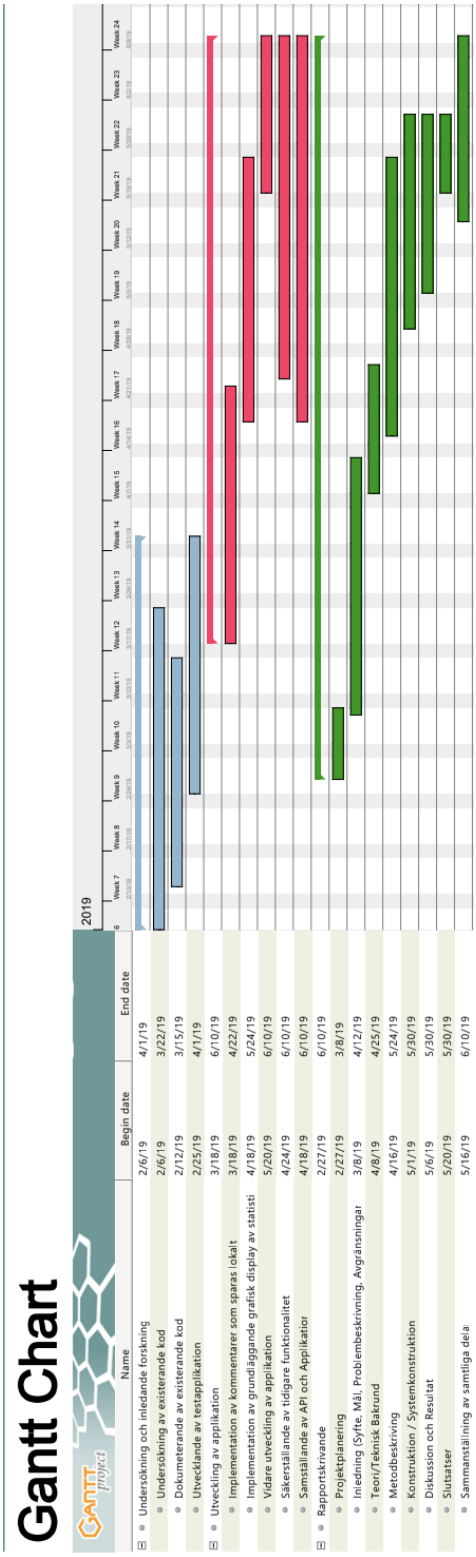
[11] ——. (2019, Apr.) Android app development. [Online]. Available: https://eu.udacity.com/course/new-android-fundamentals--ud851[Accessed:2019-04-15]

A

Bilaga

A.1 Gantt

Nedan i Figur A.1 visas ett Gantt-schema över tidsplanen för arbetet.



Figur A.1: Gantt-schema