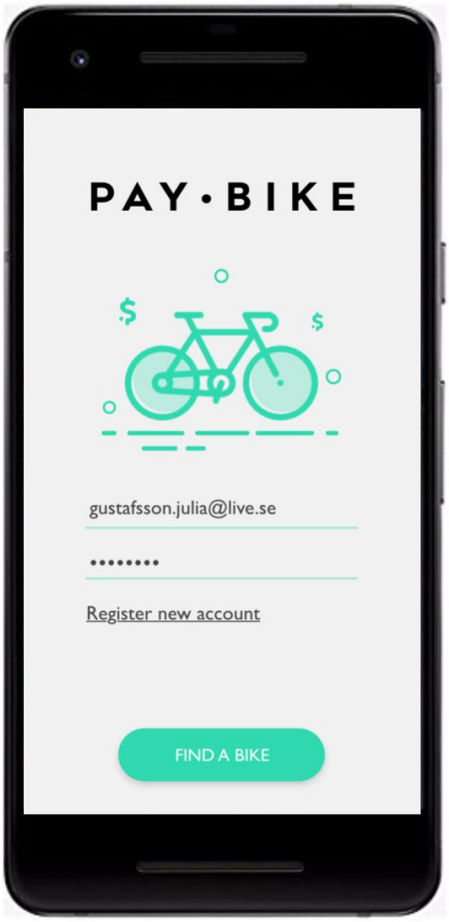
Slutrapport

för PayBike applikation



Grupp 24 Brädgård

Pontus Backman

Anton Boyertson

Oscar Kilberg

Julia Gustafsson

2018-10-28

Innehållsförteckning

1. Introduktion

De senaste åren har *delningsekonomi* blivit ett begrepp som de flesta associerar med applikationer som Uber och Airbnb. Nationalencyklopedin sammanfattar delningsekonomin som ett “samlingsnam på aktiviteter som syftar till minskad resursåtgång genom effektivare kapacitetsutnyttjande, såsom delning av tillgång till varor eller tjänster.” (Nationalencyklopedin, 2018). Dagens Industri uppgav förra året att närmare en miljon svenskar använt, eller använder sig av en applikation av detta slag (Leijonhufvud, 2017).

PayBike är likt AirBnb en applikation som vars syfte är att utgöra en mötesplats för personer som har ett eller flera överflödiga föremål att hyra ut, samt personer i behov av att hyra något de saknar tillgång till. Denna första version av applikationen är begränsad till, samt fokuserar på delning av cyklar. Personer med en extra cykel har möjlighet att tillhandahålla applikationen med information om cykeln, dess geografiska position samt till vilket pris man kan hyra den. Övriga användare kan visa tillgängliga cyklar samt begära att få hyra dessa till angivet pris. En av applikationens huvudsakliga syften är därmed att möjliggöra kommunikation mellan dessa parter.

Kommunala satsningar som Styr och Ställ i Göteborg fyller ett liknande syfte men fallerar ibland då cykelställen enbart förekommer på ett begränsat antal geografiska platser. Dessutom är dessa cyklar inte utrustade med tillräckligt många växlar för att erbjuda komfortabla cykelturer i mer kuperade områden som återfinns bland annat runt Chalmers Campus.

Applikationen karaktäriseras av ett minimalistiskt grafiskt gränssnitt som tillåter en användare att registrera sig, alternativt logga in. Inloggning är ett krav för att få åtkomst till applikationens funktionalitet. Tillgängliga cyklar presenteras i en lättöverskådlig lista som är sökbar för att möjliggöra personliga urval. Genom att trycka på ett av objekten visas mer detaljerad information samt erbjuds möjligheten att skicka en hyresförfrågan. Användare har även möjligheten att enkelt ladda upp nya annonser med titel, pris,

beskrivning, cykelns adress samt en bild. Samtliga användare har tillgång till funktionalitet för att kunna hyra ut, samt hyra cyklar, likt exempelvis Blocket.

1.1 Definitioner, akronym och förkortningar

Android OS - Operativsystem för Android

Användare - Utgörs av både uthyrare och hyrestagare. En användare kan dessutom utgöra båda dessa typer.

API - Application Program Interface, förenklar kommunikation med andra program.

Databas - extern lagring av information/data.

Epic - En övergripande målbild för applikationens användningsområde.

GUI - Graphical User Interface (Grafiskt användargränssnitt)

Hyrestagare - Person som hyr en cykel.

Instans - objekt som existerar oftast när applikationen körs.

Lifecycle - Android aktivitets livscykel

MVC - Förkortning för designmönstret Model View Controller

MVVM - Förkortning för designmönstret Model View View Model

STAN - applikation som analyserar struktur i java-applikationer.

Task - Deluppgift för att förverkliga en User Story.

User Story - En beskrivning av ett av applikationens användningsområden sett ur användarens perspektiv.

Uthyrare - Person som tillhandahåller en cykel för uthyrning.

2. Krav

2.1 User Stories

Följande User Stories har legat till grund för utvecklingen av applikationen. Varje user story inleds med ett #id, tidsuppskattning angiven i x antal timmar [x], följt av en beskrivande titel, en förklaring samt kravspecifikation. User Stories presenteras i fallande prioritetsordning där #1 har högre prioritet än #2.

#1 [10] Som utvecklingsteam vill vi ha en startbar "Hello World"-applikation till Android för att säkerställa att underliggande infrastruktur är på plats och fungerar.
Krav: Applikationen kan byggas och köras via Gradle på vilken dator som helst. Kontakt med databasen fungerar för alla användare. Åtkomst till applikationen finns via GitHub. Tester är automatiserade via Travis.

#2 [10] Som köpare vill jag kunna visa en tillgänglig cykel i applikationen.
Krav: Objektet läses utan hinder från databasen. Modellen reflekterer databasens innehåll och håller därmed samma objekt. Den grafiska representationen stämmer överens med innehållet i databasen och visar följande fält: namn, pris och plats.

#3 [10] Som köpare vill jag kunna visa en lista med tillgängliga cyklar för att få en överblick av utbudet.
Krav: Objekten läses utan hinder från databasen. Modellen reflekterer databasens innehåll och håller därmed samma objekt. Den grafiska representationen stämmer överens med innehållet i databasen och visar följande fält: namn, pris och plats. Varje cykel särskiljs med ett unikt id. Varje cykel förekommer en gång i listan.

#4 [5] Som köpare vill jag kunna visa mer information om en cykel för att kunna avgöra om den är intressant eller ej.
Krav: Korrekt objekt visas vid klick i listan av cyklar. Modellen tillhandahåller nödvändig information om objektet. Den grafiska representationen stämmer överens med innehållet i databasen och visar följande fält: namn, pris, plats och beskrivning.

#5 [20] Som användare vill jag kunna skapa en annons för att marknadsföra min cykel.

Krav: Användaren anger namn med maximal längd om 50 karaktärer, position i Sverige bestående av adress, postnummer samt stad, beskrivning med maximal längd av 1000 karaktärer och slutligen pris per timme med tak på 999 USD.

#6 [20] Som användare (köpare/säljare) vill jag kunna skapa ett konto i applikationen och använda detta för att identifiera mig samt associera förfrågningar och annonser med mig.

Krav: Lösenordet består av minst 8 tecken. Mailadressen innehåller ett @. Modellen kan hålla information om en redan inloggad användare för att undvika att inloggning krävs vid varje start av applikationen. Modellen håller inte användardatan för att undvika exponering av inloggningsuppgifter såsom lösenord. Utloggning är möjlig för att växla konto eller avsluta aktivitet.

#7 [5] Som uthyrare vill jag kunna lägga till en bild i min annons för att intressenter ska få en uppfattning om hur min cykel ser ut, vilket skick den är i etc.

Krav: Bilden kan laddas upp från bildgalleriet på telefonen. Cykelobjekt utan bild visar placeholder som ersättning. Bilden skalas för att passa in i det grafiska gränssnittets olika vyer.

#8 [20] Som köpare vill jag kunna skicka en förfrågan till uthyraren.

Krav: Förfrågan avser ett cykelobjekt och innehåller ett start och sluttid. Pris anges vid förfrågan som en funktion av antal timmar och minuter som förfrågan avser. Minsta totalkostnad är en timmes avgift. Starttid måste vara efter aktuell tidpunkt, dvs förfrågan kan inte avse en historisk tidpunkt. Sluttid måste inträffa efter starttid. Förfrågan modelleras med ett unikt id, en avsändare samt id:t för den berörda cykeln. Användare kan inte skicka förfrågningar till sig själva.

#9 [20] Som uthyrare vill jag kunna motta förfrågningar samt kommunicera en bekräftelse eller ett avslag till avsändaren.
Krav: Endast ett besked kan lämnas - avslag eller godkännande. Lämnat besked kan inte ändras. En förfrågan som förblir obesvarad och passerar sitt datum nekas automatiskt.

#10 [5] Som användare vill jag kunna navigera i applikationen med hjälp av en meny.
Krav: Menyn visas i samtliga vyer som är aktuella efter inloggning.

#11 [5] Som användare vill jag kunna ta bort förfrågningar jag ångrat eller av annan anledning inte vill fullfölja.
Krav: Borttagning reflekteras i modell såväl som databas. Förfrågningar som redan besvarats kan inte ångras.

#12 [5] Som användare vill jag kunna söka i flödet av cyklar för att lättare hitta relevanta objekt.
Krav: Sökfältet filtrerar på cykelns angivna titel samt position. Sökningen filtrerar fram samtliga objekt som matchar en angiven sträng. Databasen samt modellen är oförändrad.

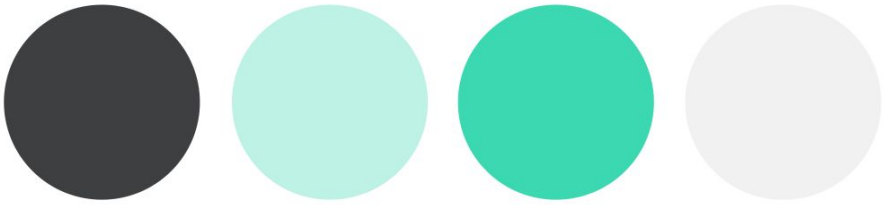
#13 [2] Som användare vill jag kunna uppdatera en vy utan att lämna den och återkomma för att upptäcka förändringar i utbud eller förfrågningar.
Krav: Nya objekt populerar modellen och databasen vid uppdatering. Den nya informationen visas omedelbart för användaren. Ladd-ikon visas för att informera användaren om bakgrundsaktivitet vid inladdning.

2.2 User Interface

Applikationen har ett minimalistiskt grafiskt gränssnitt som förmedlar modernitet. Den gröna accentfärgen representerar de miljömässiga fördelar som finns i delningsekonomi samt cykeln i sig som ett transportmedel utan negativ miljöpåverkan. Följande bilder visar skisser av applikationen innan utveckling.

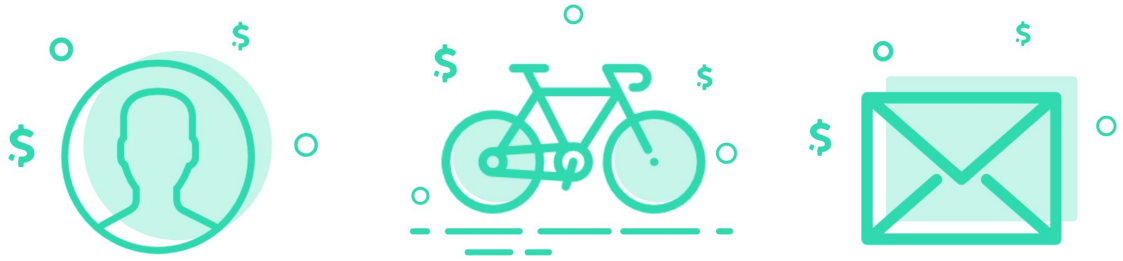
Basfärger

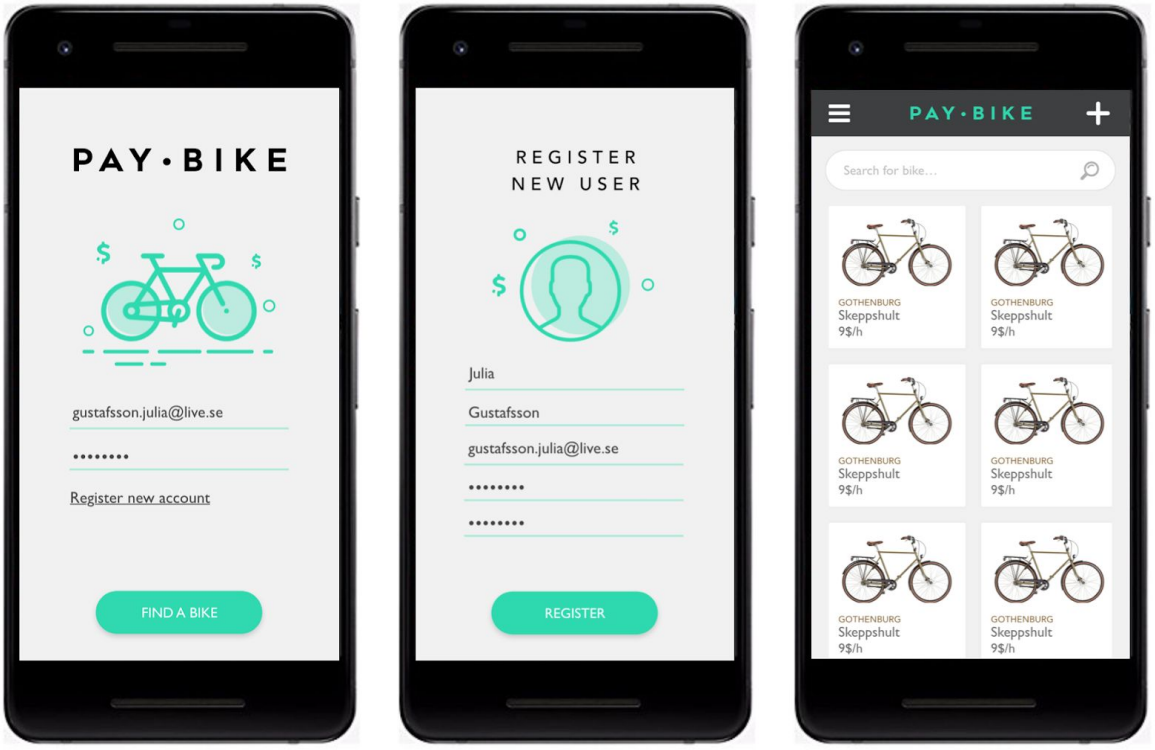
#3E3F41 #F1F1F1



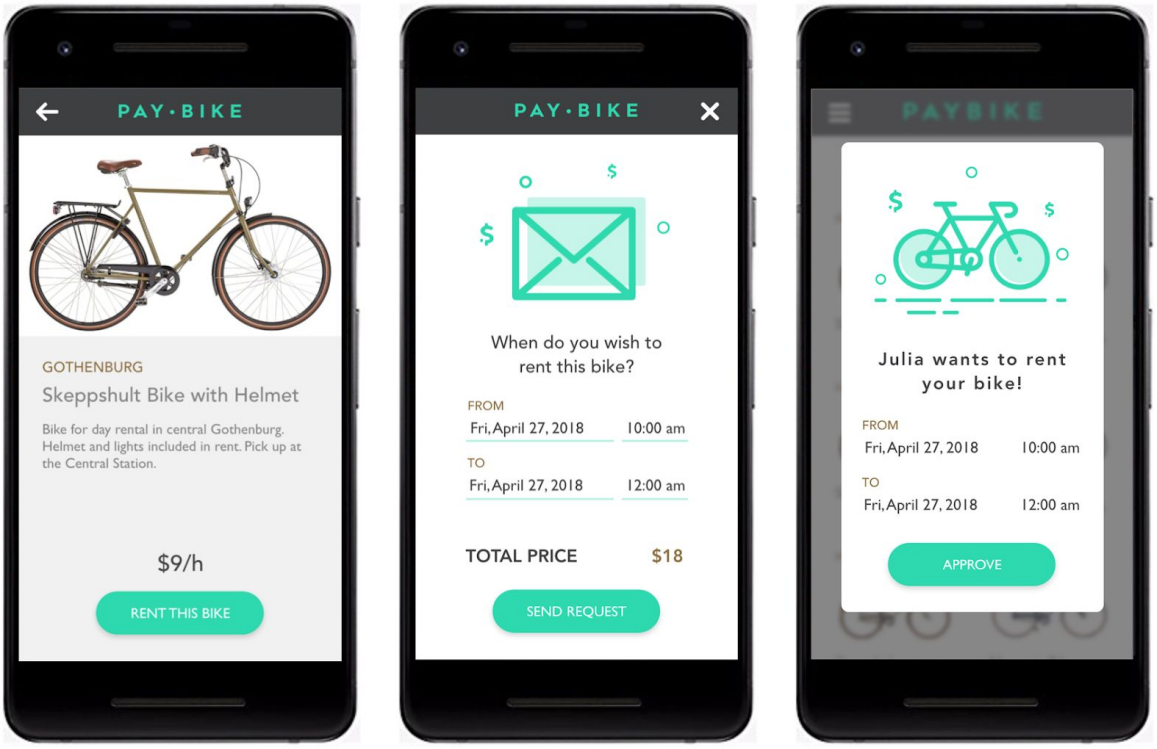
Accentfärger

#C7F5EA #30D9AF

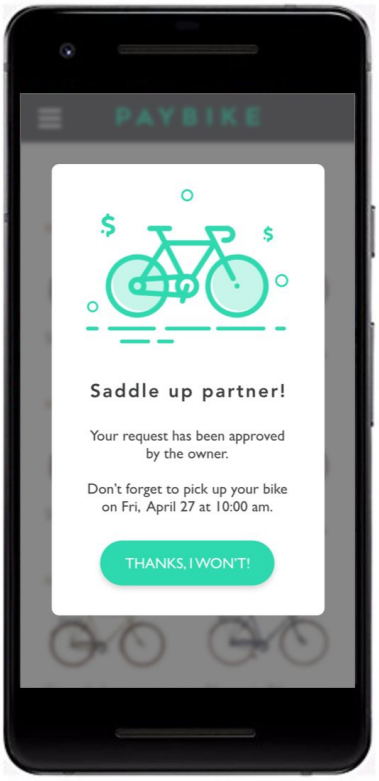




Fr. vänster: Konceptuell vy av inloggningsskärm, vid tryck på "Register new account" visas vy för registrering (mitten), efter inloggning visas ett flöde av cyklar tillgängliga för uthyrning.

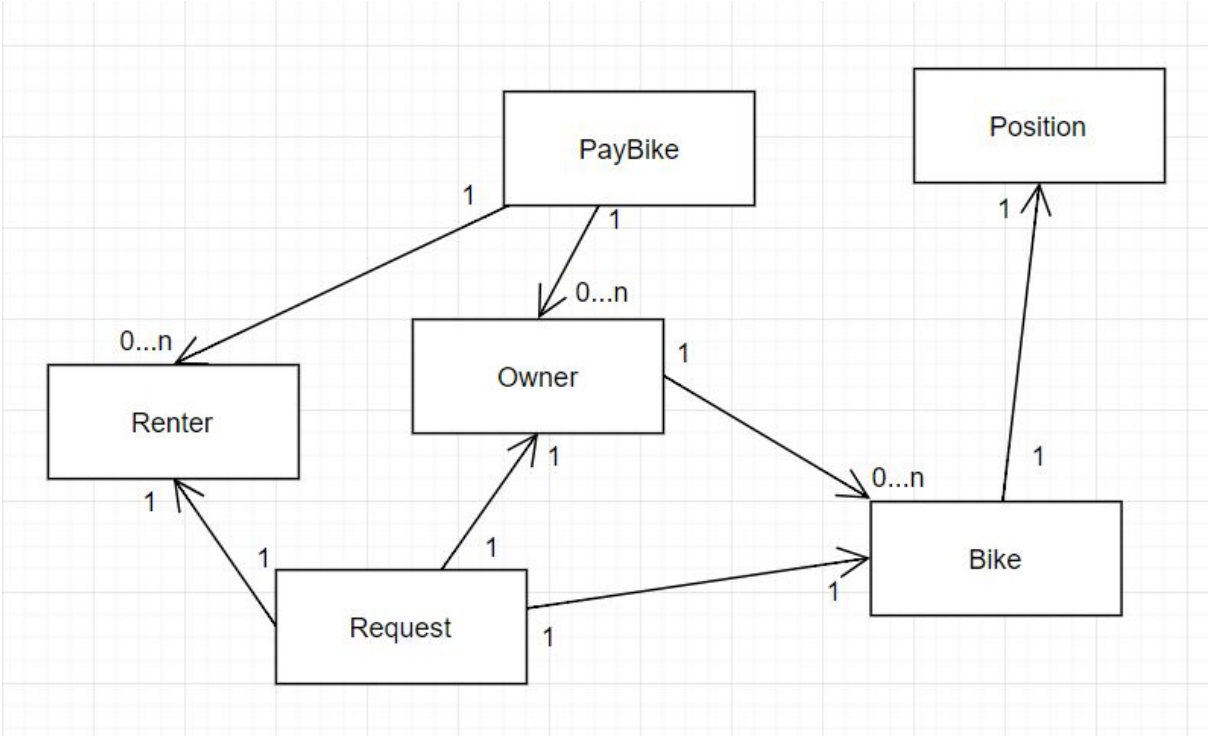


Fr. vänster: Detaljvy för ett cykelobjekt, vid tryck på "Rent this bike" visas vy för förfrågan (mitten). Tredje bilden visar en notis som uthyraren mottar vid.



Konceptuell vy av besvarad förfrågan.

3. Domänmodell



UML-diagram som översiktligt beskriver applikationen

3.1 Domänens delar

PayBike är kärnan av applikationen som består utav Owners, Renters, Requests och Bikes. Denna har koll på dessa och håller logiken bakom.

Owner är en av de olika användare som kommer existera i applikationens domän. Dessa utmärker sig i att de har cyklar och har förmågan att hyra ut sin cykel genom att publicera den och därmed synliggöra den för potentiella Renters/Hyrare.

Renter har förmågan att se de cyklar en Owner publicerar och skicka en förfrågan i form av Request.

Request är kopplad till en specifik cykel, dess ägare och hyrestagaren och skickas till en Owner. Skulle denna godkänna förfrågan får Renter tillfällig tillgång till cykeln i fråga.

Bike representerar cyklarna som går att hyras av Renter och publiceras av Owner. Däremot har alla möjligheten att hyra ut/hyra samtidigt, därmed är ingen användare enbart Renter/Owner utan kan mycket väl utföra handlingar kopplade till båda. Eftersom en cykel måste ha en position har även Bike en Position.

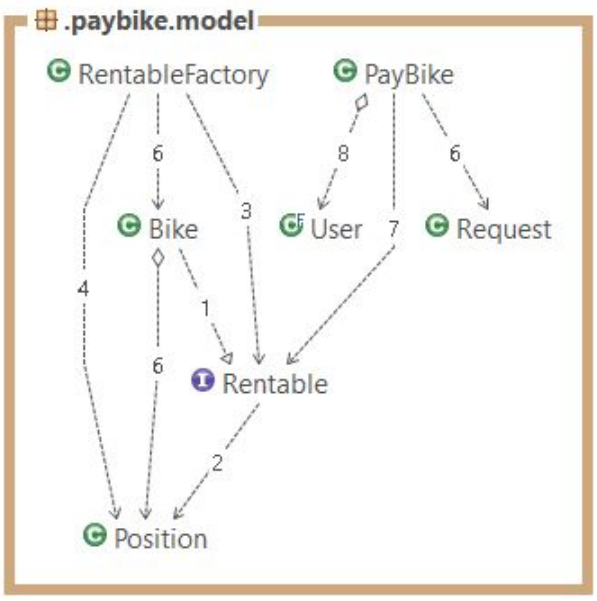
Position är den position som cykeln kommer befinna sig på vid uthyrning, och troligtvis även vid återlämning.

4. Systemarkitektur

Detta kapitel syftar till att redogöra för applikationens arkitektur med syftet att läsaren ska ges en god inblick i dess uppbyggnad och funktionalitet. Applikationen har utvecklats för plattformen Android och är bakåtkompatibel till Android SDK 26.

Programmet är strukturerat enligt principerna för Model-View-ViewModel kombinerat med Repository-mönstret. Varje viewmodel har en repository för att hämta den information som önskas av dess motsvariga activity från modellen, repository i sin tur är medlare mellan modell och den externa databas vi använder och ser till att dessa stämmer överens. Modellen som representerar domänen är oberoende av både viewmodels och repository.

4.1 Modell



Modellen är ansvarig för att hålla applikationens grundläggande logik och tillfällig data. Här hålls de hyrbara objekt, applikationens nuvarande användare och förfrågningar som sedan presenteras visuellt av Android-aktiviteterna. Modellen vill hållas oberoende av resterande system-komponenter.

4.1.1 Klasser i modellen

`Rentable`

Ett interface som är mall för alla hyrbara objekt. Tanken är att bygga den så fler klasser än `Bike` ska kunna implementeras vid behov. T ex har applikationen i framtiden möjligheten att även erbjuda uthyrning av användbara tillbehör som hjälmar och annan säkerhetsutrustning eller stödhjul.

`Bike`

Klass som representerar cykel objekt, innehåller information om cykeln själv, position och vem dess ägare är.

`User`

Användarklassen som innefattar all väsentlig information om användaren, exempelvis email, användarnamn och lösenord.

`Request`

Klassen representerar en förfrågan som användare har möjlighet att skicka vid intresse av att hyra specifik cykel.

`Position`

Modellerar en svensk adress utifrån stad, postkod och gata. Landet sätts i nuläget till Sverige i och med att den enbart hanterar svenska adresser.

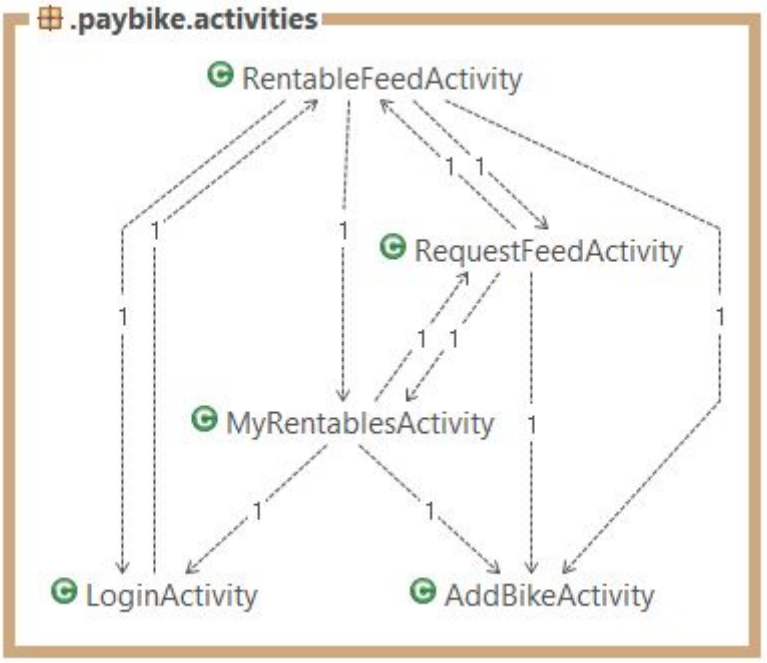
`RentableFactory`

Gömmer logiken bakom skapandet av hyrbara objekt för klienten och öppnar även denna upp för utökning av framtida objekt som behöver skapas.

`PayBike`

Singleton som håller alla `Rentables`, `Requests` och den nuvarande användaren.

4.2 Activities



Activity-klasserna håller användargränssnittet och representerar därmed View i MVVM. Med dessa visas interagerbara och synliga objekt på användarens Android enheten. Eftersom en aktivitet kan pausas eller avslutas när som helst utav användaren eller Android-systemet bör dessa inte hålla data utan istället hämta denna från en känd `ViewModel`. Bland annat ser detta till att Seperation of Concern principen uppehålls då Activities eller Fragments ej längre behöver hålla annat än logik gällande UI interaktion eller Android-specifika uppgifter. Både activities och fragment har korresponderande .xml filer som håller informationen till deras visuella objekt. Objekten hanteras sedan av metoder i aktiviteten/fragmentet.

4.3 ViewModels

Var activity använder en viewmodel för att kalla på andra komponenter och framför förfrågningar att ändra/lägga till/ta bort data. `ViewModels` känner ej till aktiviteterna och på så sätt förlorar inte användaren data ifall enhetens OS eller användare avbryter motsvarande aktivitet. `ViewModels` har en `Repository` som de med hjälp av skickar och tar emot data från modellen, repositoryn ser även till att denna data från vyobjekten sparas i databasen. För att det ska gå smidigt och för att följa Dependency Inversion där

moduler på hög nivå inte ska vara beroende av moduler på låg nivå används `Repository` av `ViewModels` istället för att dessa ska vara direkt beroende av modellen eller databasen.

4.4 Repository

`Repository` är klassen som sköter kommunikation mellan den externa databasen och den lokala modellen. Vid start ber huvud-aktivitetens viewmodel att `Repository` uppdaterar modellen med den lagrade informationen tillgänglig i databasen och sätter den inloggade användaren som aktuell användare. När en lokal ändring sker i modellen uppdaterar `Repository` databasen så att denna ändring är tillgänglig för alla kopplade enheter. Vår `Repository` delegerar databas-specifika uppgifter till en trio “handlers”.

4.5 Fragments

I Fragments hålls de fragment som är kopplade till specifika activities och används för att utöka layouten av en aktivitet. Fragment kan visas som en del av skärmen eller täcka hela.

4.6 Adapters

De adapters som används i vårt applikation, möjliggör många objekt av samma typ, fast med olika information, kan visas i form av lista eller rutnät. Istället för att Activities skulle lösa detta problemet, finns en mellanklass som “översätter” alla olika attribut till ett objekt som ges till activity-klassen som i sin tur visar alla olika objekt.

4.7 Handlers

Applikationens handlers används av repository för att utföra specifika databasrelaterade uppgifter gällande att konvertera information till och från javaklass samt lägga till och ta bort från databasen. Förutom `UserHandler` använder dessa databas-kontrollen i deras uppgift.

`UsersHandler`

Hanterar alla metoder kopplade till `User` och kommunicerar med databasen. Inloggning och skapande av nya användare sker med hjälp av `UserHandler`.

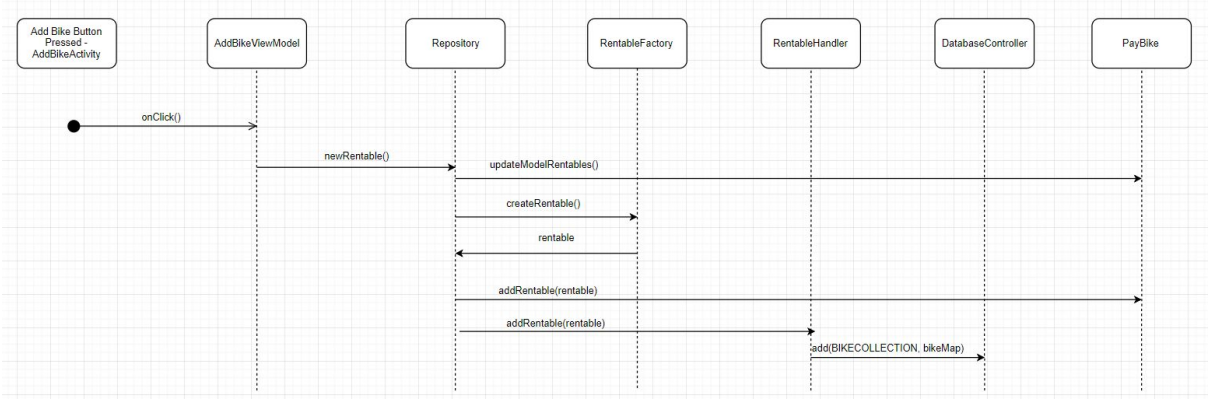
`RentableHandler`

Hanterar hämtande och givande av objekt av typen `Rentable` från och till databasen. I nuläget av typen `Bike`, men utökning för fler hyrbara objekt är möjlig.

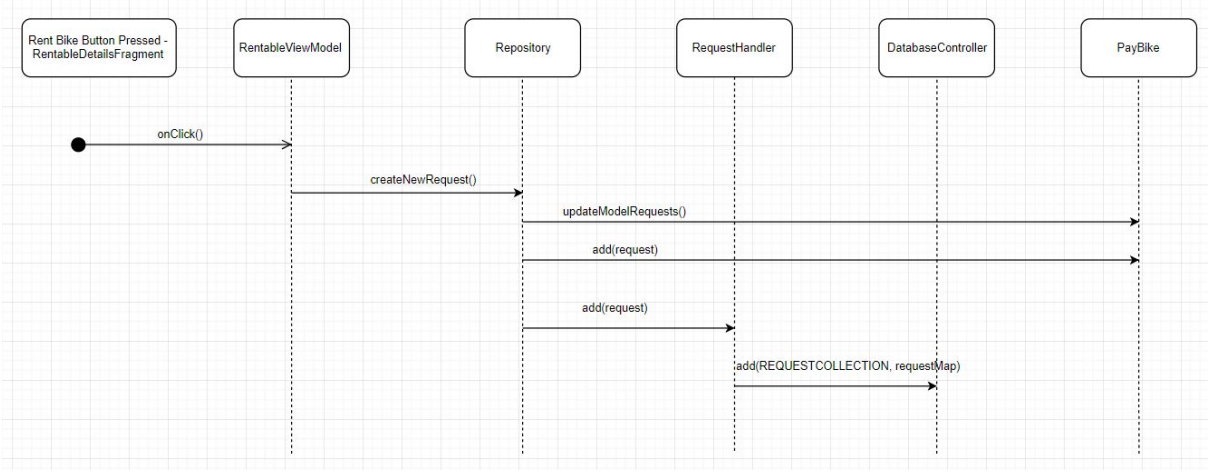
`RequestHandler`

Hanterar hämtning och tillägg av objekt av typen `Request` (förfrågningar) i databasen.

4.8 Sekvensdiagram



Användare lägger till en `Rentable`



Användare skickar en `Request`

4.9 Design-Patterns

Factory-pattern

Används vid skapandet av Rentables. Se `RentableFactory` (s. 14)

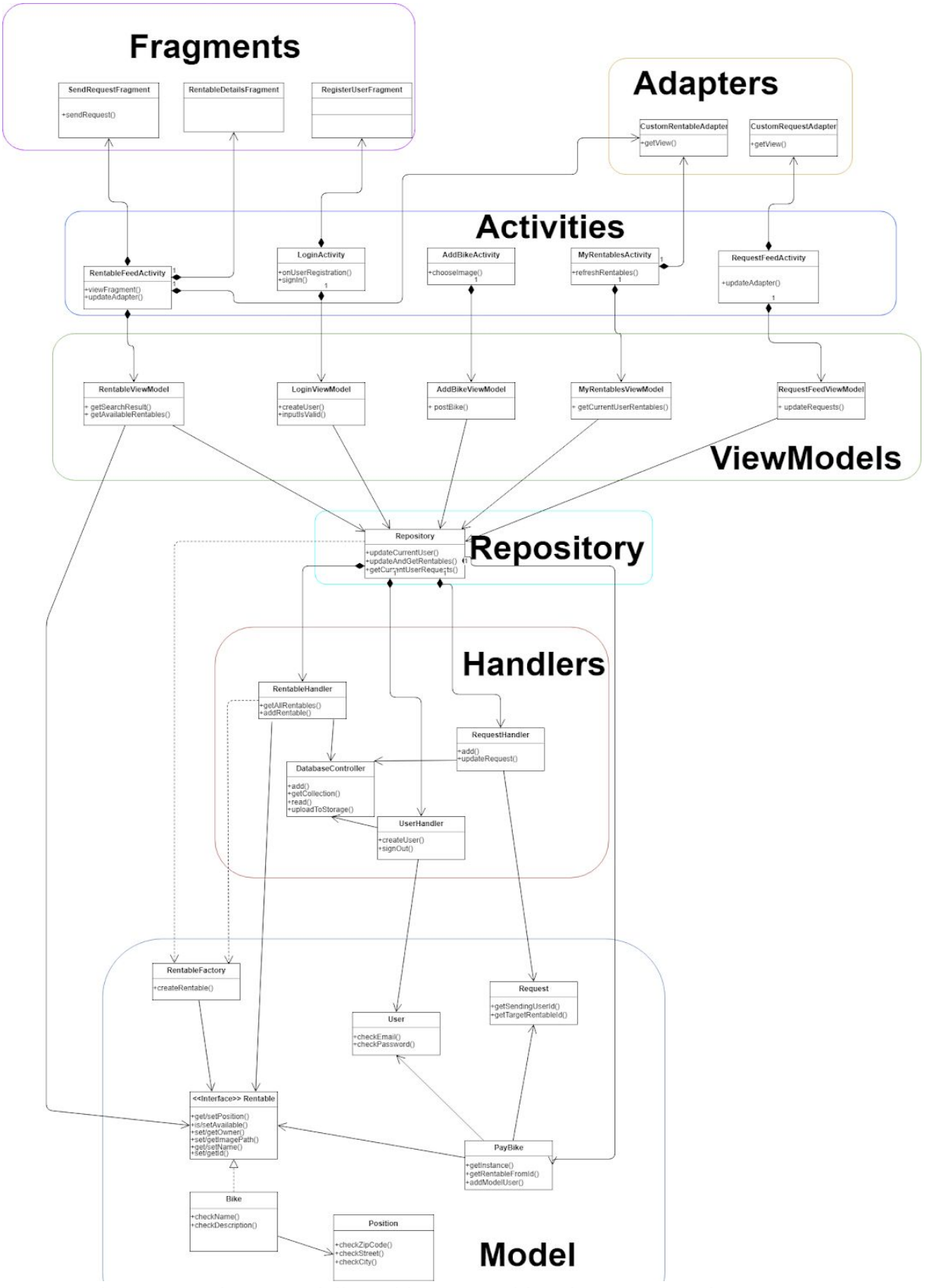
Singleton-pattern

Klasser som vi enbart vill ha en instans av använder Singleton-pattern. Denna instans kan nås via getInstance() och finns ingen instans av klassen skapas ett nytt objekt. Används t ex av PayBike för att se till att inte flera olika samlingar av cyklar/requests kan existera samtidigt i modellen.

Observer-pattern

För att notifiera vår applikation t ex när användaren klickar på ett objekt på skärmen används Observer-pattern. Förekommer ofta som “lyssnare” och säkerställer att flera objekt kan uppdateras vid ändring av tillstånd.

4.10 Analys av beroende



UML diagram av designmodellen inklusive utvalda relevanta metoder.

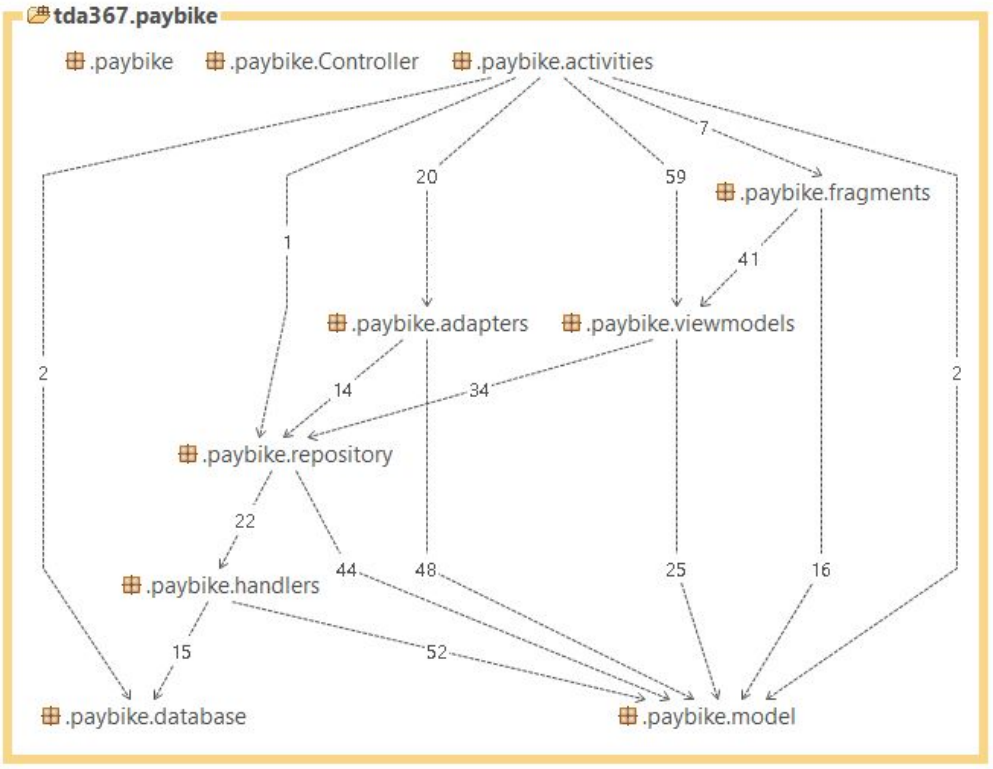


Bild från STAN som beskriver beroenden inom applikationens struktur

5. Hantering av persistent data

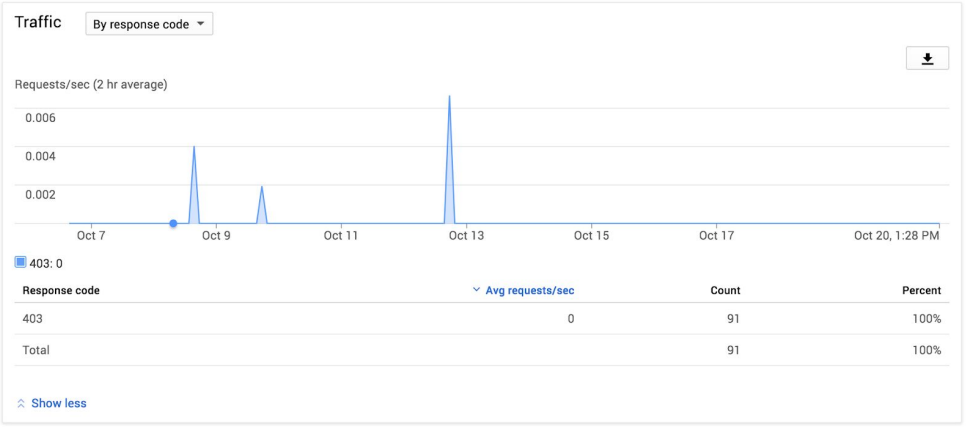
Applikationen lagrar data med hjälp av Google's Firebase Storage. En instans av Firebase Storage initieras i `DatabaseController` och används sedan i `uploadToStorage()` i samma klass. `uploadToStorage()` tar en sökväg som input, laddar upp filen sökvägen pekar på till Firebase Storage med ett slumpgenererat namn och returnerar en Firebase Task av typen `URI`. Denna Task kan sedan användas för att hämta information om filen samt en nedladdnings-URL.

Uppladdningen till Firebase Storage används i `RentableHandler` när ett objekt av typen `Rentable` läggs till i databasen. I metoden `addRentable()` hämtas sökvägen till en bild från en given `Rentable` och laddas upp till Firebase Storage via `uploadToStorage()` i `DatabaseController`.

Samtliga användare har samma åtkomstnivå till databasen.

6. Kända svagheter

För att lagra information om cyklarnas positioner blev lösningen att skapa en egen typ, `Position`, för att modellera en position med gatuadress, postkod samt stad (med land Sverige som klassvariabel). Denna lösning innehåller en rad begränsningar då utformning av adresser varierar kraftigt världen över. För att fånga in den variationen hade ett betydligt mer komplext positionssystem krävts. Longitud och latitud hade även varit bra komplement för att enkelt kunna placera ut positionen på en karta vid vidare utveckling. Ett API som löser denna problematik är Googles API för Platser och Kartor- Google Places and Maps. Det innehåller över 100 miljoner platser jorden över och möjliggör hantering av Place-objekt som kan modellera samtliga av dessa platser. Tanken var att detta API skulle nyttjas vid skapande av nya cykelobjekt i `AddBikeActivity` för att tillåta autocompletion när användare angav en adress. Vid implementering uppkom dock komplikationer som inte löstes. Google mottog de anrop som gjordes från applikationen till deras servrar men skickade tillbaka Unknown Error som enda svarskod.



Skärmdumpar från Google Cloud Console. 91 anrop mottagna för GetAutocompletePredictions varav samtliga besvarades med Error.

Sökningar på forum såsom Stackoverflow samt Googles Github repo gav ingen förklaring till vad som var orsaken. Om denna applikation byggts för kommersialisering hade det varit nödvändigt att vända sig till ett API likt Googles för att hantera positions-information. I syftet för en prototyp fungerar dock den tillfälliga lösningen med `Position` hjälpligt.

Ett annat problem som hanteras på ett mindre optimalt sätt är navigeringen mellan olika vyer i applikationen. Navigeringssystemet prioriterades bort i inledande utvecklingsfas vilket ledde till att applikationen utformades på ett sätt som inte gynnade navigering med en meny i nedre delen av GUIt. Därmed installerades istället en meny i samtliga Activities som behöver tillåta navigering. Nackdelen med detta tillvägagångssätt är att menyn måste definieras i samtliga Activities istället för att definieras separat och sedan med hjälp av referenser, återkomma i fler vyer.

Nuvarande applikation tillhandahåller ingen möjlighet att kontrollera om en cykel hyrs ut flera gånger vid samma tidpunkt. Det är upp till uthyraren att avgöra om en cykel är tillgänglig vid tidpunkten som en förfrågan gäller. Önskvärt hade varit om en kalender implementerats för samtliga cykel-objekt för att lagra information om tillgänglighet vid bokning.

Ytterligare en konsekvens som följer av det faktum att applikationen ännu är en prototyp, är avsaknaden av integrerad betalningstjänst. Priser förekommer trots detta i applikationen för att tydliggöra applikationens syfte - att möjliggöra uthyrning av cyklar. Applikationen syftar inte till att utgöra en plattform för gratis utlåning.

7. Kollegial granskning

Detta avsnitt utgör en granskning av projektet Qdio utfört av Grupp 25.

7.1 Design och implementation

7.1.1 Enhetlig kodstil

Applikationens kodstil är för det främsta konsistent. Nästan alla konstruktorer är placerade längst upp precis efter klassernas instansvariabler och därefter kommer medföljande metoder. Däremot har `TrackListAdapter` sin konstruktor i mitten av klassens kod, vilket gör den svårare att hitta och modifiera den.

Användandet av *Access level modifiers* hittas i varje klass och enbart ett fåtal rekommenderas att ändra sin Access level då användandet är mer begränsat än tillgängligheten.

En vanligt förekommande typ i programmet är `LiveData` och av denna används dess subclass `MutableLiveData` konsistent och förekommer inte plötsligt i formen av sin superclass.

PlayerState klassen håller däremot enbart enum för de olika stadium Player kan befinna sig i och att ha en hel klass dedikerad till enbart så få enum är inte nödvänigt, istället hade dessa kunnat finnas i SpotifyPlayer-klassen.

7.1.2 Återanvändbarhet

Hög cohesion överlag då de flesta klasser har ett jobb och gör endast det, t.ex. `ImageFetchTask` som gör exakt det klassnamnet indikerar och inget annat. På grund av detta är det lätt att återanvända klassen. Dessutom leder exempelvis Factory Pattern till att man kan återanvända skapandet av flera klasser.

7.1.3 Underhåll

Bristen på dokumentation försvårar underhåll avsevärt. Ett exempel är `RoomDiscoveryService`, en klass som innehåller mycket logik men helt saknar

dokumentation. För någon som inte är insatt i exakt vad denna klassen gör så kommer underhåll ta onödigt lång tid.

7.1.4 Förändrad funktionalitet

I och med användandet av Factories och Interfaces för t ex `Room` och `Player` ger detta möjligheten för att skapa olika players/rooms med utökad eller minskad funktionalitet, antingen via Override (gällande interface) eller en speciell factory-metod för ny slags `Player`, kanske en med förmågan att lägga till låtar i en "mest köade låtar" lista.

7.1.5 Designmönster

Factory pattern som klassas som ett *Creational Pattern* används för skapandet av `Player`, `Track`, `Artist` och `Album` vilket är bra för då göms logiken bakom skapandet.

Även *Behavioral Pattern* Observer används för att notifiera programmet när ändringar sker, exempelvis i SearchFragmentViewModel.

Singleton-pattern används i JsonUtil-klassen. Detta är för att det aldrig krävs att det finns flera instanser av denna hjälpklass i programmet.

7.2 Dokumentation

Som ovan nämnt saknas stora delar av programmet dokumentation. Vissa klasser, till exempel de som tillhör paketet *view,* saknar helt kommentarer medan andra delar är delvis dokumenterade. Avsaknaden av kommentarer känns mest påtaglig i klasser med ansvar för stora delar av logiken och många metodanrop. Kommentarer hade bidragit i stor utsträckning till att förtydliga vilken funktionalitet som täcks in.

7.3 Namngivning

Namnen på klasser är oftast relativt lätta att förstå. De som är mer svårtolkade vid första ögonkastet blir tydligare när man navigerar runt i programmet. Exempel på tydliga klassnamn är `Room`, `QueueListFragment`, `RoomDiscoveryViewModel`.

Även namn på metoder är bra och beskrivande genom programmet, ofta förstår man metodens syfte och funktion utifrån dess namn. Likaså är variabelnamn självförklarande.

7.4 Modulär design

Analys med hjälp av STAN visar att inga cirkulära beroenden förekommer då man bortser från testklasser. Dessutom är modellen oberoende av övriga komponenter.

Applikationen använder MVVM (Model-View-ViewModel) istället för MVC men utgår ifrån samma isoleringsprincip av modellen där ViewModels känner till Modellen men Modellen inte känner till ViewModels. Aktiviteterna/View har var en ViewModel men själva ingen direkt koppling till modellen. Detta är bra för en Android applikation eftersom aktiviteterna kan plötsligt pausas eller stängas ned av Android OS, då är det viktigt att data hålls av en ViewModel och Modellen istället för i aktiviteten så att aktiviteten kan återupptas i samma skede som där den dog.

7.5 Testning

Stora delar av den testbara koden har tester. I skrivande stund täcks 58% av metoderna i Model in av testerna. Majoriteten av dessa är dock för getters. Det finns ytterligare publika metoder i flera klasser som borde gå att testa, för att uppnå högre grad av *code coverage* vilket förhindrar att framtida förändringar förändrar beteendet på ett oväntat sätt.

7.6 Säkerhet eller prestanda

Det var inte möjligt att testa prestandan, då applikationen inte går att köra. Det behövs ytterligare konfiguration vid nedladdning för att anslutningen till Spotifys API ska fungera.

7.7 RAD & SDD

Beskrivning av applikationens tänkta syfte och funktionalitet i form av User Stories är tydlig och lätt att förstå för otekniska såväl som tekniska läsare. Kan utökas med bilder av applikationen och design.

8.0 Referenser

Jonas Leijonhufvud (3 april 2017) Här är de senaste trenderna och bolagen i Sveriges växande delningsekonomi. Hämtad från Dagens Industri den 19 oktober 2018 https://digital.di.se/artikel/har-ar-de-hetaste-trenderna-och-bolagen-i-sveriges-vaxande-delningsekonomi

Nationalencyklopedin (2018) Delningsekonomi. Hämtad den 19 oktober 2018. https://www.ne.se/uppslagsverk/encyklopedi/lång/delningsekonomi