



Köp och sälj din kurslitteratur direkt i appen!

Matilda Blomquist, Isak Lindgren, Elin Melander, Joel Rudberg, Arvid Sundbom

October 2019

Innehåll

1 Inledning	4
1.1 Definitioner, akronymer och förkortningar	4
2 Requirements and Analysis Document	5
2.1 Krav	5
2.1.1 User Stories	5
2.1.1.1 Implementerade User Stories	5
2.1.1.2 Icke implementerade User Stories	6
2.1.2 ”Definition of Done”	6
2.1.3 Användargränssnitt	7
2.1.3.1 Registrering och Inloggning	7
2.1.3.2 Applikationens startsida	8
2.2 Domänmodell	20
2.2.1 Entiteter och deras ansvar	20
2.2.1.1 User	20
2.2.1.2 Advert och Favourites	21
2.2.1.3 Chat och Message	21
3 System Design Document	21
3.1 Systemarkitektur	22
3.1.1 Google Firebase	22
3.1.2 Applikationens Flöde	23
3.1.2.1 Hem	23
3.1.2.2 Detaljvyn	24
3.1.2.3 Favoriter	25
3.1.2.4 Skapa annons	25
3.1.2.5 Meddelanden	25
3.1.2.6 Mina annonser	26
3.2 Systemets design	26
3.2.1 Beroenden mellan paketen i applikationen	26
3.2.2 Model, View, Presenter	29
3.2.3 Domänmodell och Designmodell	30
3.2.4 Callbacks	31
3.2.5 Hantering av beständig data	32
3.2.6 Designmönster	34
3.2.6.1 Observer Pattern	34
3.2.6.2 Strategy Pattern	34
3.2.6.3 Singleton Pattern	35
3.2.6.4 Template Method Pattern	35
3.2.6.5 Övriga mönster	35

3.3	Kodkvalitét	35
3.3.1	Testning	35
3.3.2	Problem	36
3.4	Kodanalys	38
3.4.1	Tillgänglighetskontroll och säkerhet	38
4	Diskussion	39
4.1	Fragments med RecyclerView	39
4.2	Sign-Up och Sign-In	40
4.3	CreateAdActivity och CreateAdPresenter	40
4.4	Navigation	40
4.5	Avvikeler från MVP	41
4.6	Gränssnitt	41
4.7	Hämtning av data	41
4.8	Beständig data under körning	41
5	Peer Review	43
Referenser		46

1 Inledning

Att ägna sig åt akademiska studier innebär nästan oundvikligen införskaffande av kurslitteratur någon gång under utbildningen. Under de tre första åren på Chalmers har de flesta program många obligatoriska kurser, vilket gör att det finns ett behov av stora volymer av den kurslitteratur som används i dessa kurser. I dagsläget finns ingen digital eller fysisk plattform som samlar all handel med begagnad kurslitteratur på ett och samma ställe. Istället finns exempelvis en rad olika facebook-grupper inom högskolans olika sektioner som används för ändamålet.

Applikationen tillhandahåller ett enkelt gränssnitt för att på ett smidigt sätt låta användaren annonsera ut sin kurslitteratur till försäljning. Annonser kan både taggas med fördefinierade ämnestaggar och taggar definierade av säljaren. Dessa taggar underlättar sökningen och kan ge bättre sökresultat. På startsidan visas alla annonser i kronologisk ordning, men det finns även möjlighet att både sortera, filtrera och göra textsökningar för att avgränsa sin sökning.

I första hand är applikationens målgrupp studenter på Chalmers och Göteborgs universitet som läser på campus Johanneberg. Applikationen kommer att förenkla handeln med begagnad kurslitteratur då applikationen är specialanpassad för ändamålet samt samlar alla annonser på ett och samma ställe. Att köpa begagnade böcker är både bra för miljön men även för studentens ofta tunna plånbok.

Denna rapport redogör för de krav och specifikationer på applikationens funktionalitet som satts upp, hur applikationen navigeras, är uppbyggd och vilken systemarkitektur som används.

1.1 Definitioner, akronymer och förkortningar

- Activity - Android-specifik klass som oftast interagerar med användaren, och utgör en fullskärmsaktivitet.
Kan hålla och visa upp fragments, men utgör även en vy i sig.
- Fragment - Android-specifik klass som framför allt används som vy. Representerar ett beteende eller utgör en del av användargränssnittet i en FragmentActivity.
- Mocka - Att skapa en falsk version av ett kod-objekt med ett förutbestämt beteende. Används främst för att begränsa enhetstester till en viss kodmodul.
- MVP - Åsyftar systemarkitekturen 'Model, View, Presenter'.
- Presenter - fungerar som ett mellanlager vars uppgift är att sammanbinda applikationens view med applikationens kärna, den underliggande modellen.
- SOLID - En uppsättning designprinciper för att inom objektorienterad programmering uppnå god design.
- SoC - Separation of Concern. En av SOLID-principerna för att uppnå en god ansvarsfördelning mellan klasser och metoder.
- LSP - Liskov Substitution Principle. En av SOLID-principerna som definierar krav för att implementationsarv ska vara rättfärdigat.
- Tagg - etikett eller märkning

- Tagga - sätta en etikett på ett objekt
- Thumbnail - En bild i miniatyrstorlek, i detta fall ett produktkort.
- UML - Unified Modelling Language; ett objektorienterat språk som används vid modellering av alla typer av system, främst inom programvarukonstruktion.
- User Stories - ett verktyg som används in om mjukvaruutveckling för att beskriva den funktionalitet som användare behöver.

2 Requirements and Analysis Document

2.1 Krav

Detta kapitel listar alla krav på den funktionalitet applikationen ska innehålla och definierar även vilka kriterier som måste uppfyllas för att en viss del av denna funktionalitet ska anses helt färdigimplementerad.

2.1.1 User Stories

Nedan listas alla User stories som används för att specificera applikationens funktionalitet i fallande prioritetsordning.

2.1.1.1 Implementerade User Stories

- Som en student på Chalmers vill jag se all tillgänglig kurslitteratur för att jag vill hitta begagnade böcker.
- Som en student på Chalmers vill jag enkelt kunna annonsera ut gammal kurslitteratur för att hitta köpare då jag inte behöver kurslitteraturen längre.
- Som en student vill jag kunna komma i kontakt med säljare för att kunna köpa böcker.
- Som en säljare vill jag kunna redigera och ta bort min annons om jag angett fel information.
- Som säljare vill jag kunna tagga mina objekt för att det ska bli enklare för köpare att hitta min bok.
- Som en student på Chalmers vill jag söka på specifik kurslitteratur för att snabbt hitta rätt kurslitteratur.
- Som en köpare vill jag kunna sortera mellan de annonser som visas för att snabbt hitta det bästa alternativet.
- Som en användare vill jag kunna kommunicera med säljaren/köparen i applikationen för att enkelt kunna köpa/sälja böcker.
- Som en användare vill jag kunna favoritmarkera annonser för att snabbt kunna komma tillbaka till de mest relevanta annonserna.
- Som en student på Chalmers vill jag kunna filtrera på pris och ämnen för att enklare hitta den bok jag söker.

2.1.1.2 Icke implementerade User Stories

- Som en säljare vill jag kunna skapa en annons med ett bokpaket för att på ett smidigt sätt sälja flera böcker som jag tror att samma köpare kan vara intresserad av.
- Som student på Chalmers vill jag kunna ange vilken årskurs/program jag läser och därmed få bra rekommendationer på relevant kurslitteratur för att hitta böcker som är intressanta för mig.
- Som en användare vill jag kunna lägga upp ”sökes”-annonser för att hitta någon som vill sälja en bok jag söker till mig.
- Som en köpare vill jag kunna lägga in en bevakning på en bok för att bli notifierad när det läggs upp en annons med boken jag söker.

Att notera är dock att den näst sista funktionalitetskravet implicit är uppfyllt, då det inte finns någon särskiljning mellan ”sökes” eller ”säljes”-annonser. Det går därmed tekniskt sett att lägga upp en sökes-annons i applikationen.

2.1.2 ”Definition of Done”

Följande kriterier sattes upp för att definiera när en User-story kunde anses färdigimplementerad.

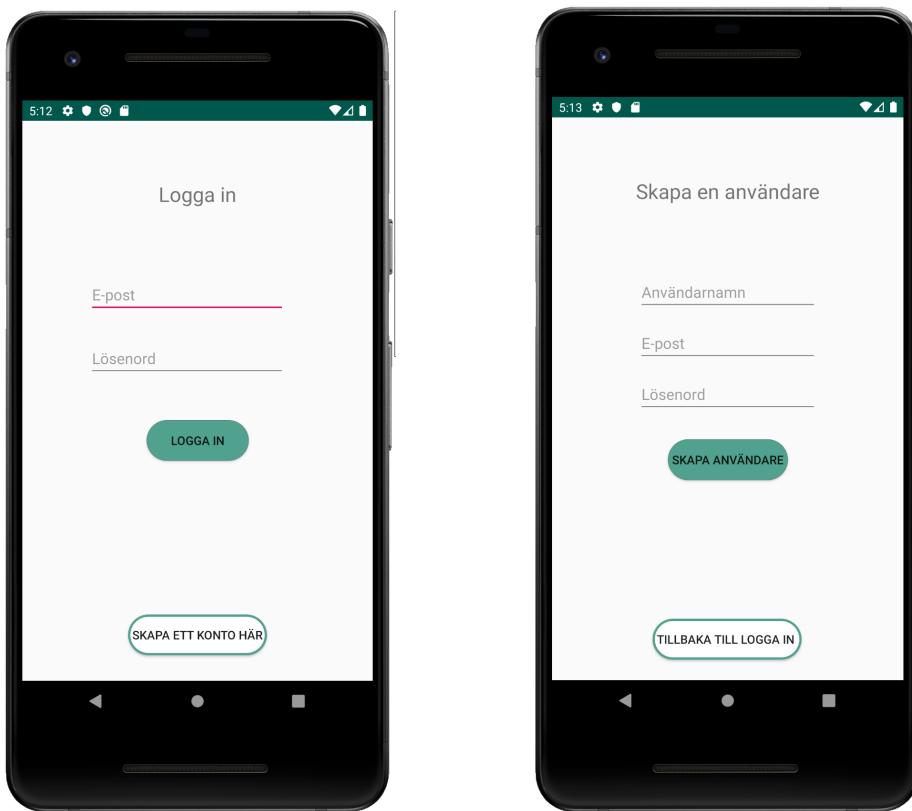
1. Den implementerade koden klarar tillhörande enhetstester som skrivits med hjälp av JUnit.
2. Projektkoden kompilerar utan felmeddelanden.
3. Koden ska uppfylla allmänna icke-funktionella krav. Dessa krav består exempelvis av att appen skall köras med god prestanda och att applikationskoden ska använda så restriktiva access-modifiers som möjligt.
4. Koden skall vara så nära självdokumenterande som möjligt.
5. Koden skall vara väldokumenterad med javadoc-kommentarer. Minimumkravet för varje klass är en inledande kommentar som beskriver klassens ansvarsområde och huvudsakliga funktionlighet.
6. Applikationens gränssnitt ska efterlikna den designprototyp som utformats i Adobe XD.

2.1.3 Användargränssnitt

Tanken bakom applikationens användargränssnitt var att det skulle vara enkelt, snyggt och följa vanliga designmönster och principer inom interaktionsdesign. Appen följer till exempel mönstret ”Few Hues, Many Values” [1], då den bara använder sig av tre huvudfärger som sedan förekommer i olika mättnad.

2.1.3.1 Registrering och Inloggning

De användare som inte varit inloggade i applikationen sedan tidigare möts av en inloggningssida när de startar applikationen, se figur 1a. Denna innehåller två textfält; ett för användarens e-postadress och ett för dess lösenord. När en korrekt kombination av dessa angivits i applikationen kan användaren navigera vidare, in i applikationen via knappen med texten ”Logga in”. Ifall användaren saknar ett konto till applikationen kan de navigera till en registreringssida via knappen som placeras längst ned på inloggningssidan. Här kan användaren ange ett användarnamn, en e-postadress samt ett lösenord via de tre textfält som placeras på sidan, se figur 1b. När giltiga värden angetts i dessa kan användaren registrera sig genom att trycka på knappen ”Skapa användare”. I och med detta loggas användaren in automatiskt och användaren dirigeras till startsidan.



(a) Inloggningssidan som användaren möts av när applikationen startas.

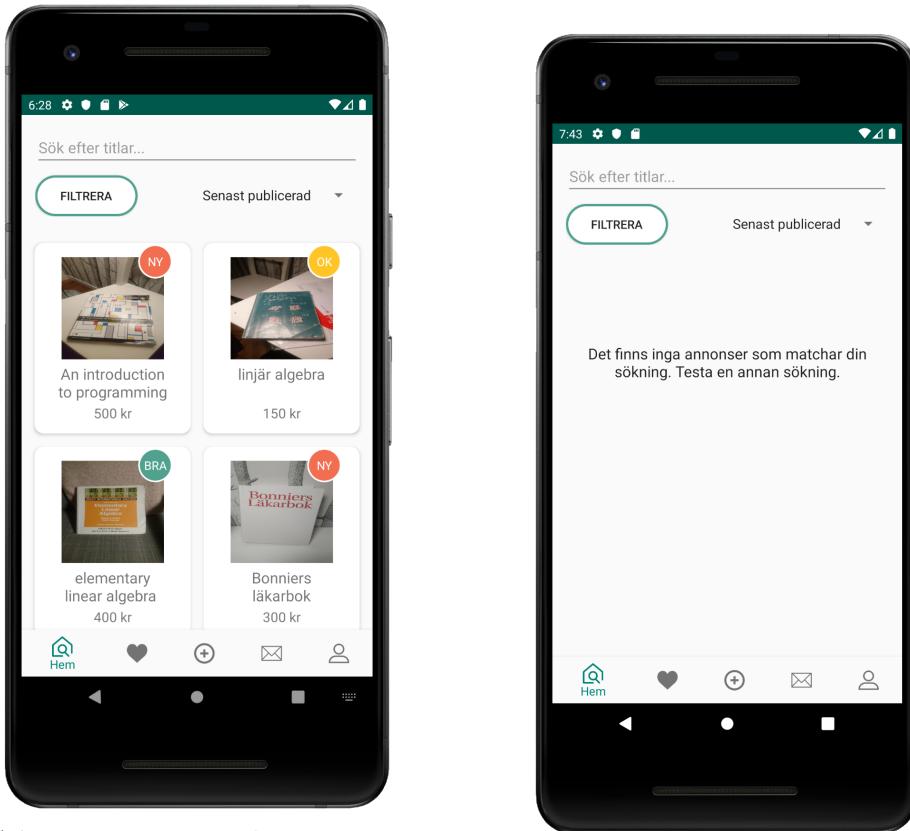
(b) Registreringssidan där användaren kan skapa ett nytt konto.

Figur 1: Inloggings- och registreringsvyn.

2.1.3.2 Applikationens startsida

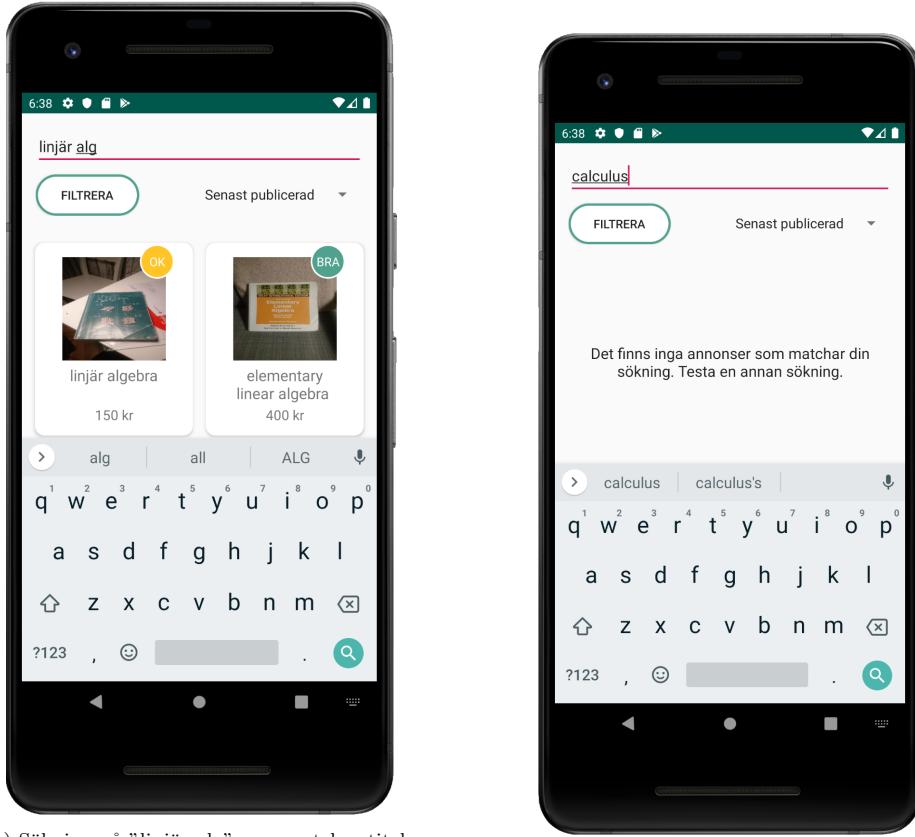
När användaren är inloggad visas en vy med ett skrollbart flöde som visar miniatyrer av de annonser som laddats upp, se figur 2a. Denna vy är även det första som visas när applikationen startas ifall användaren var inloggad när applikationen senast stängdes ned.

Om inga uppladdade annonser finns tillgängliga visas istället ett felmeddelande som förklrar detta för användaren, se figur 2b. Dock visas samma felmeddelande som när en sökning inte gett några resultat vilket kan vara vilseledande.



Figur 2: Startskärmen med eller utan annonser.

För att enklare hitta relevanta annonser finns både en sök- och filtreringsfunktion. Åtkomsten till sökfunktionen finns direkt på startsidan, och sökningen utförs så fort användaren börjar skriva i sökfältet, och söker på titlar och taggar i annonsen.



(a) Sökning på "linjär alg" som matchar titeln i annonsen till vänster och en tagg i annonsen till höger.

(b) Sökning på "calculus" som inte ger något sökresultat.

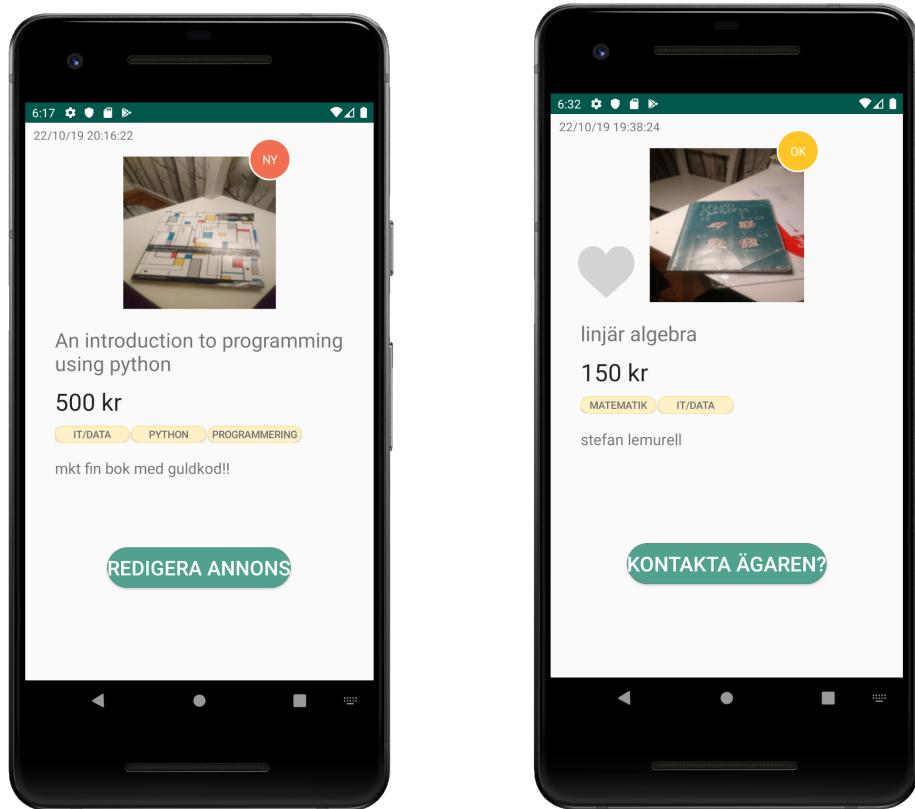
Figur 3: Sökresultat för olika sökningar.

Åtkomst till filtreringsfunktionen fås genom knappen "filtrera" i det övre vänstra hörnet. I denna vy kan användaren välja ett antal taggar denne vill söka genom att klicka på dem på samt ange ett maxpris genom att justera skjutreglaget, se figur 4. Maxpris är alltid förvalt för att inga annonser oavsiktligt ska väljas bort. Filtrering blir aktiv när användaren trycker på knappen "filtrera" och användaren skickas tillbaka till hemfliken. Eftersom att navigeringsmenyn fortfarande är synlig kan användaren fortfarande navigera till övriga flikar.



Figur 4: Filtreringsvyn

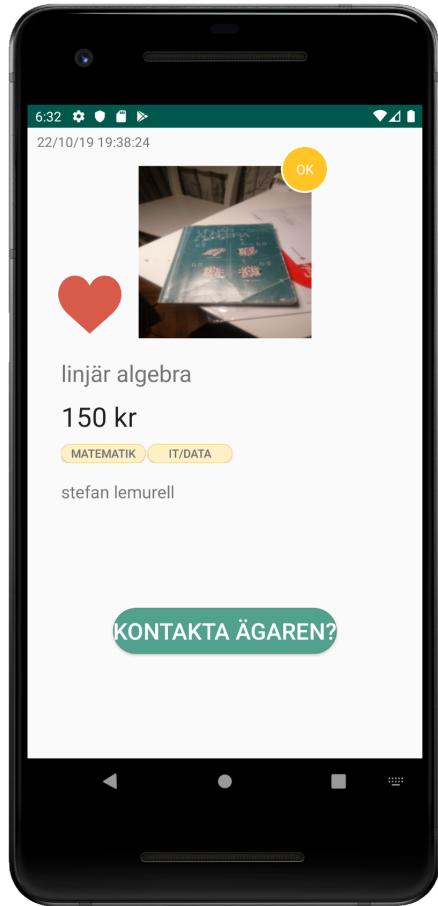
I det skrollbara flödet visas annonserna som miniatyrer. Om användaren klickar på en sådan öppnas en detaljvy och all information kopplad till annonsen visas upp, se figur 5a. I denna vy visas dessutom en ikon som har formen av ett hjärta. I sitt ursprungliga tillstånd är denna grå för att markera att annonsen i detaljvyn inte är en av användarens favoriter. Om användaren klickar på denna ikon läggs den aktuella annonsen till i användarens favoritsamling och som respons på detta färgas hjärtat rött, se figur 6.



(a) Detaljvyn som den visas för ägaren av annonsen.

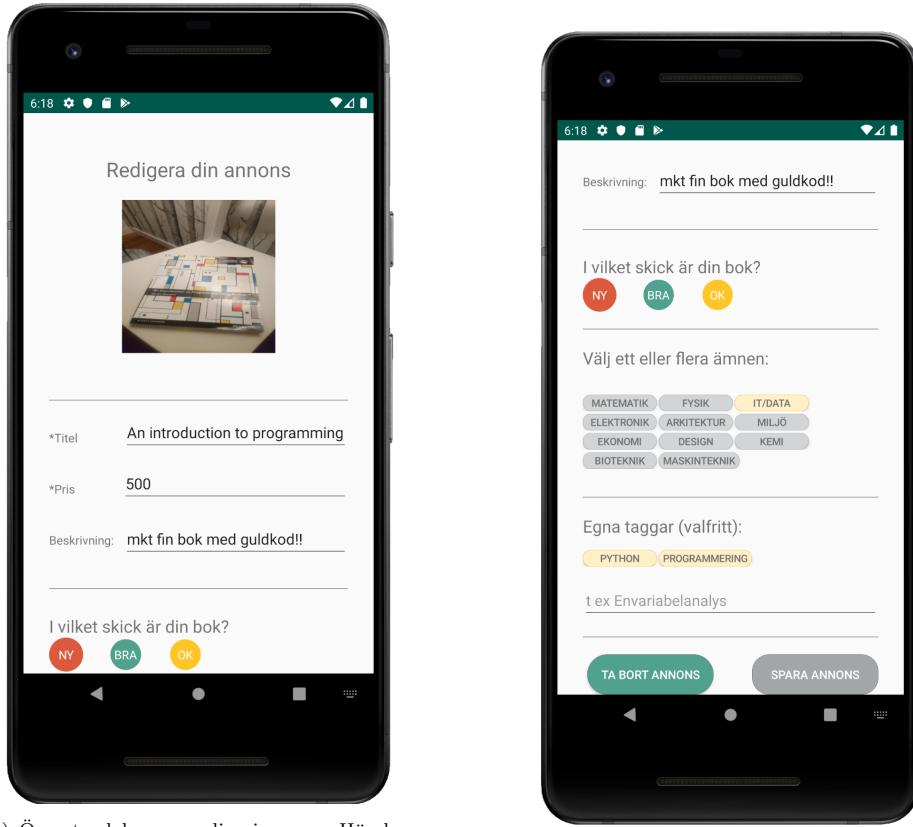
(b) Detaljvyn för en användare som inte äger annonsen.

Figur 5: Detaljvyn för annonsens ägare respektive en annan användare.



Figur 6: En detaljvy med ett rött hjärta som markerar att annonsen markerats som favorit.

För att navigera vidare från detaljvyn kan användaren använda systemets inbyggda bakåtknapp, alternativt klicka på knappen ”Redigera annons” för att öppna redigeringsvyn. I denna vy ges användaren möjlighet att ändra annonsens information, se figur 7. När användaren är nöjd med den redigerade annonsen kan de klicka på knappen ”Spara annons”. Användaren dirigeras då till hemfliken som innehåller alla annonser som laddats upp och får ett meddelande om att annonsen laddats upp. Utöver att redigera annonsen kan användaren även välja att ta bort annonsen från marknaden genom att klicka på knappen ”Ta bort annons”.

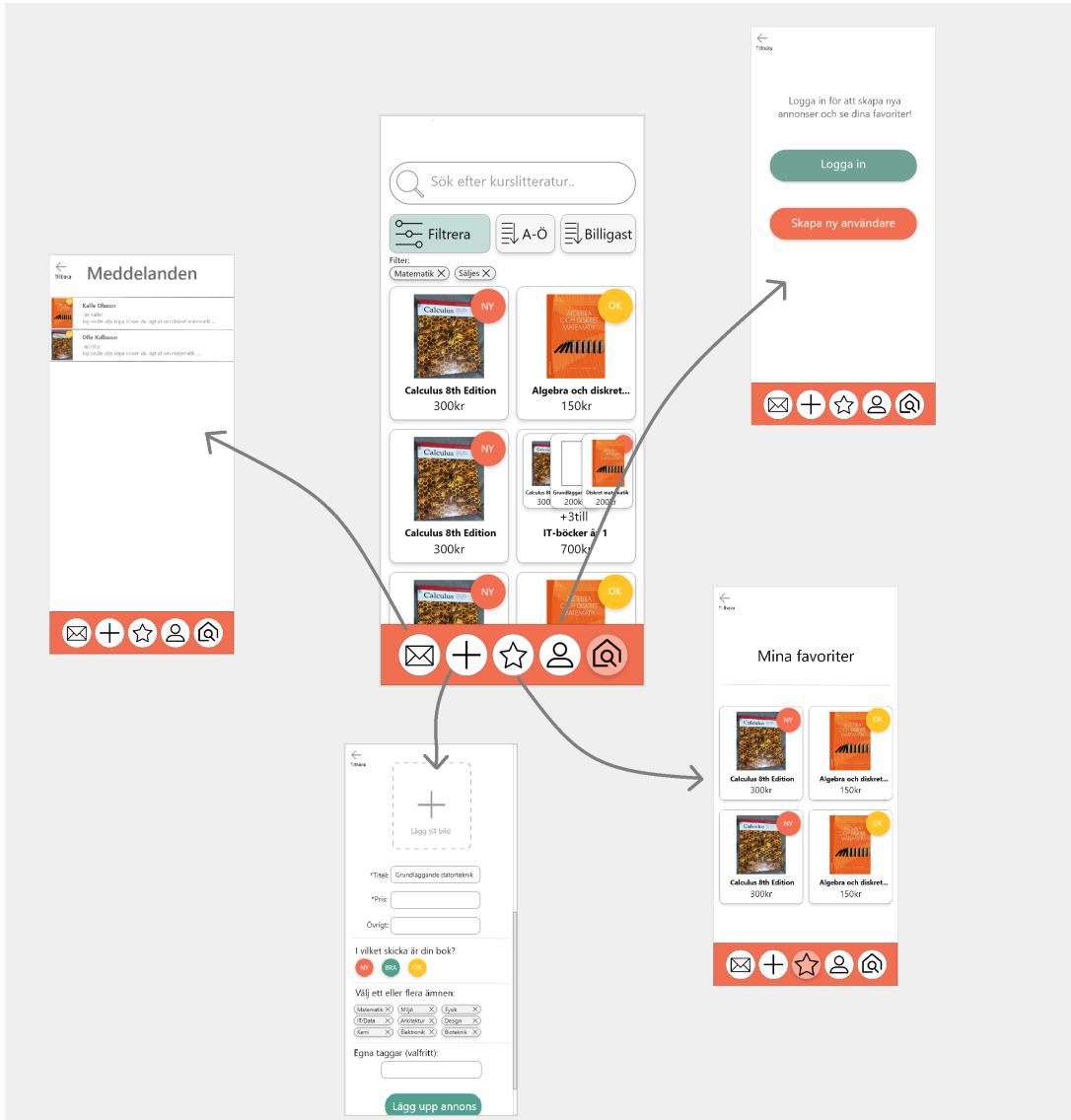


(a) Översta delen av redigeringsvyn. Här kan användaren ta en ny bild genom att klicka på bilden och öppna kameran och redigera informationen i textfälten.

(b) Skrollas redigeringsvyn kan användaren ändra val av skick och taggar genom att trycka på knapparna i vyn.

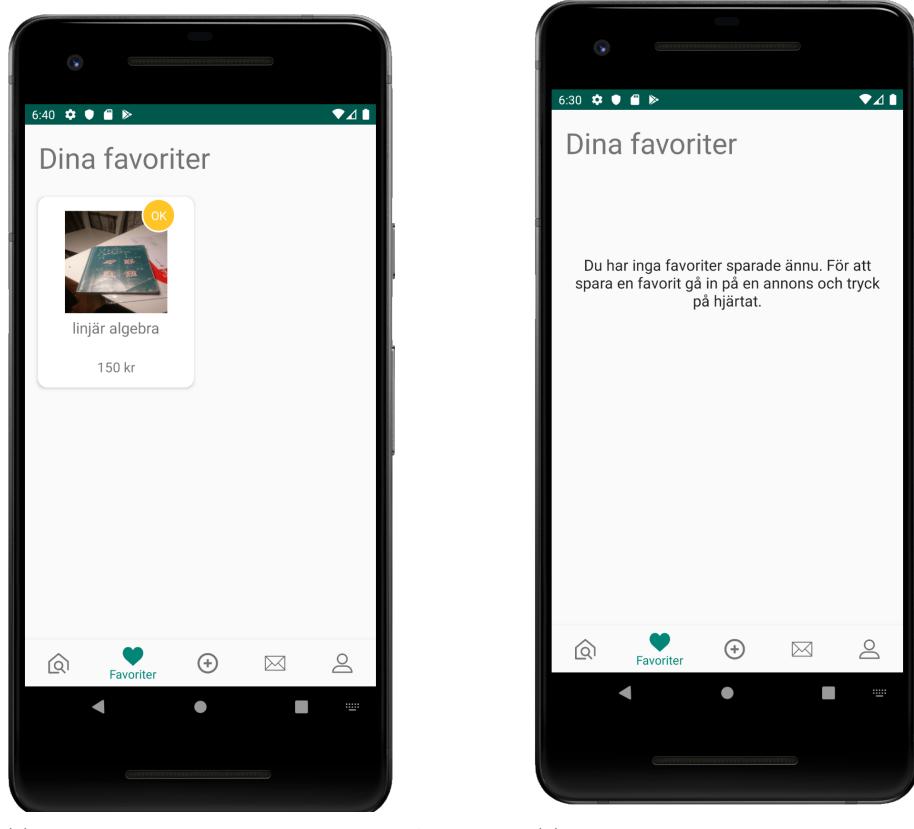
Figur 7: Redigeringsvyn.

Via de olika ikonerna i menyfältet som placeras längst ned i applikationen kan användare navigera till applikationens olika vyer. En visuell översikt över hur menyfältet var tänkt att fungera under prototypstadiet av applikationen ges i figur 14. Den faktiska implementationen av detta menyfält särskiljer sig enbart utseendemässigt från det fält som definierades i applikationens prototyp, i form av att fältets ikoner placerats i en annan ordning.



Figur 8: Prototyp i Adobe xd [2]. Grundläggande navigation från menyfältet.

När en användare klickar på fältets hjärtikon navigerar applikationen till favoritvyn, se figur 9a. De annons-miniaturer som visas i denna vy har samma funktionalitet som de i det skrollbara flödet på startskärmen, när användare klickar på dem visas en detaljvy av motsvarande annons. Utöver detta har favoritvyn även ett implementerat ett meddelande som visas när användaren inte har några favoritmärkade annonser, se figur 9b.

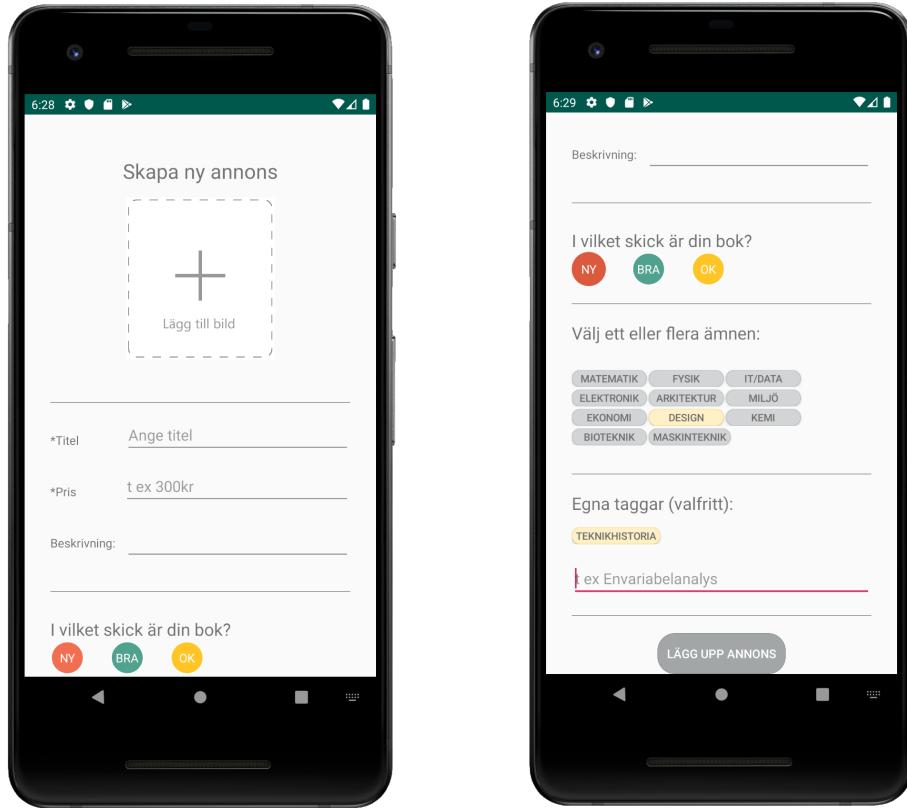


(a) Vyn som visar de annonser användaren favoritmarkerat.

(b) Det meddelande som visas när användaren inte har markerat några favoriter

Figur 9: Favorityvn

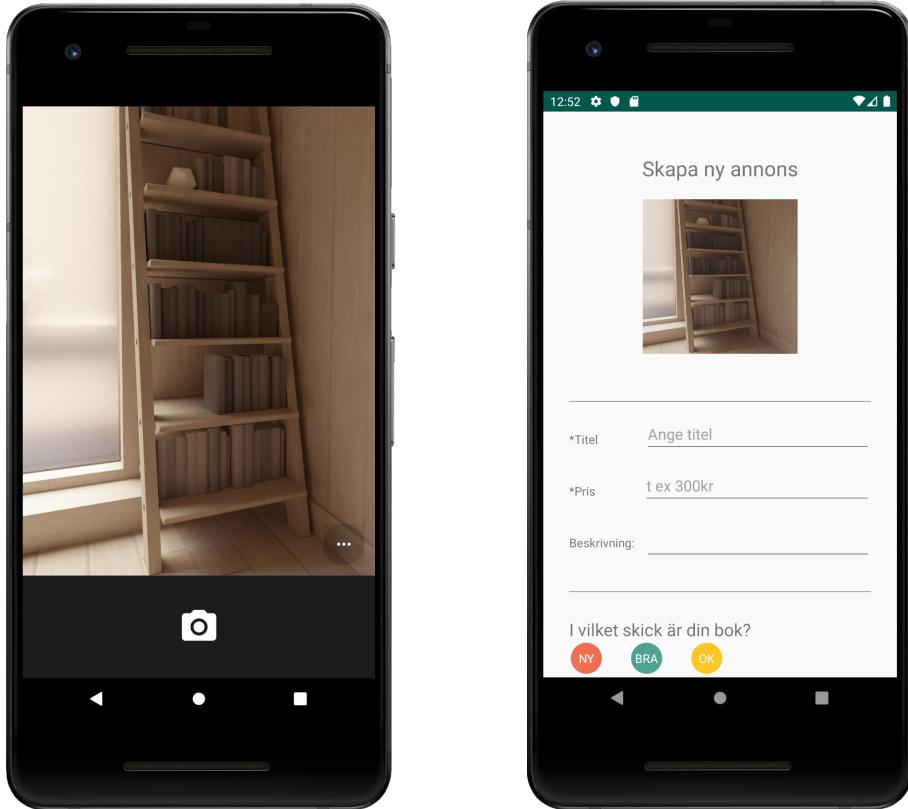
När användaren klickar på plusset i mitten av navigationsmenyn visas en vy som liknar den tidigare nämnda redigeringsvyn, se figur 10a och 10b. Istället för den information som angivits om annonsen visas istället diverse knappar och textfält där användaren kan ange den information som krävs för att ladda upp en ny annons.



(a) Den övre delen av vyn för att skapa annonser.

(b) Den nedre delen av vyn för att skapa annonser.

Användaren kan dessutom ladda upp en passande bild till annonsen. Detta görs genom att trycka på den rektangulära ikonen högst upp i vyn med texten "Lägg till bild". När användaren klickar på denna öppnas android-enhetens kamera och användaren kan ta en bild, se figur 11a. När en lämplig bild tagits och användaren återvänder till applikationen visas den tagna bilden i den uppdaterade vyn, se figur 11b. När användaren är nöjd med annonsen och all nödvändig information angetts kan användaren ladda upp annonsen på marknaden genom att trycka på knappen "Lägg upp annons". Användaren dirigeras till hemfliken och får ett meddelande om att annonsen lagts upp.

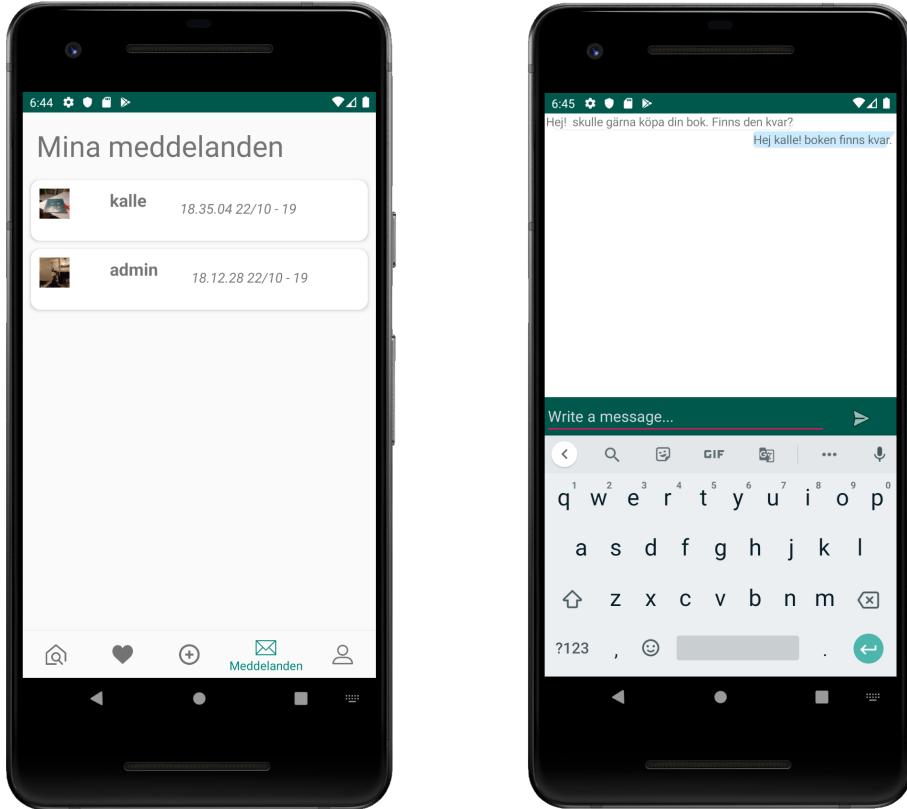


(a) Enhetens kameraapplikation som startas via applikationen för att ta en bild.

(b) Den uppdaterade vy där den tagna bilden visas.

Figur 11: Enhetens kamera används för att lägga till en bild i annonsen.

Om användaren klickar på kuvertikonen i navigationsmenyn visas en vy som innehåller ett antal radikoner, se figur 12a. Dessa representerar de konversationer användaren deltar i. Ikonerna innehåller en del information om den aktuella konversationen; Användarnamnet på den andra deltagaren i konversationen, tidpunkten när det senaste meddelandet skickades och slutligen en miniatyrversion av bilden som tillhör den annons konversationen kretsar kring. Om användaren klickar på någon av raderna i vyn öppnas konversationens chattvy, se figur 12b. I denna vy visas de meddelanden som skickats i konversationen. Här kan användaren dessutom skicka nya meddelanden via det textfält som placeras längst ned i vyn. Tyvärr saknas en del information om konversationen i chattvyn, exempelvis står det inte vem användaren chattar med eller via vilken annons konversationen startades. I nuläget måste även användare navigera bakåt via systemets bakåtknapp till den översiktliga vyn för konversationer om de glömt denna information, något som kan upplevas som onödigt och irriterande för användaren.



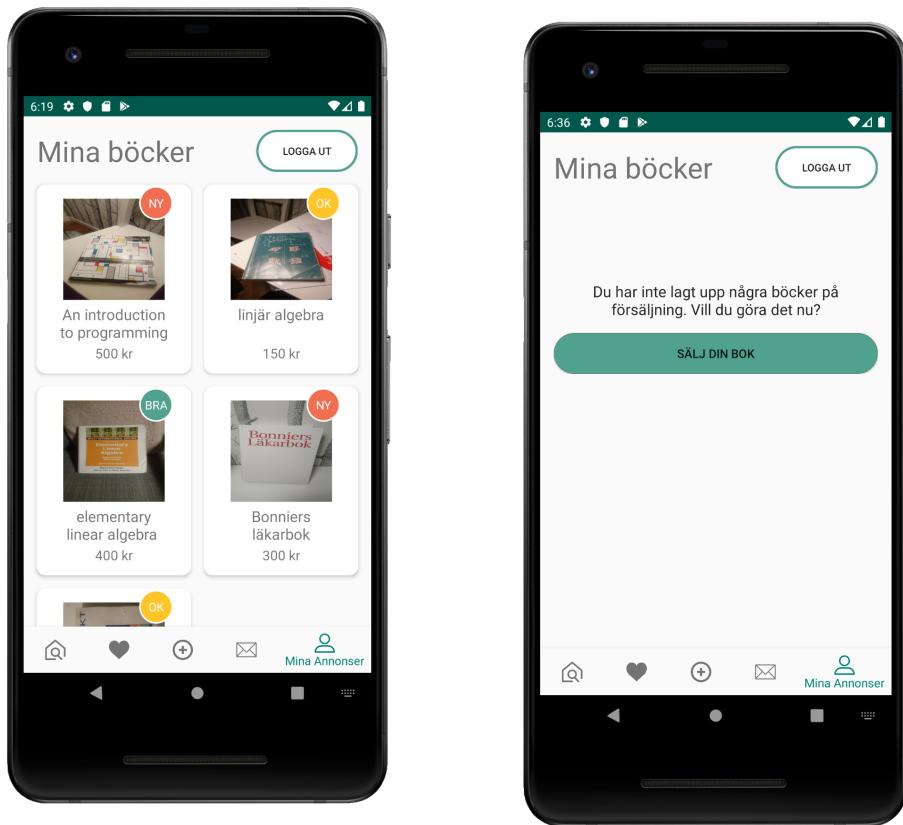
(a) Meddelandevyn, varje rad representerar en enskild konversation.

(b) En konversations chattvy, med två meddelanden.

Figur 12: Meddelande- och chattvyn

Ikonen längst till höger i navigationsmeny leder till en vy som innehåller alla annonser användaren laddat upp, se figur 13a. Här listas alla annonser den inloggade användaren laddat upp i form av thumbnails. Dessa thumbnails fungerar på samma sätt som de i det skrollbara flödet på startsidan; om användaren klickar på dem visas en detaljvy över den berörda annonsen.

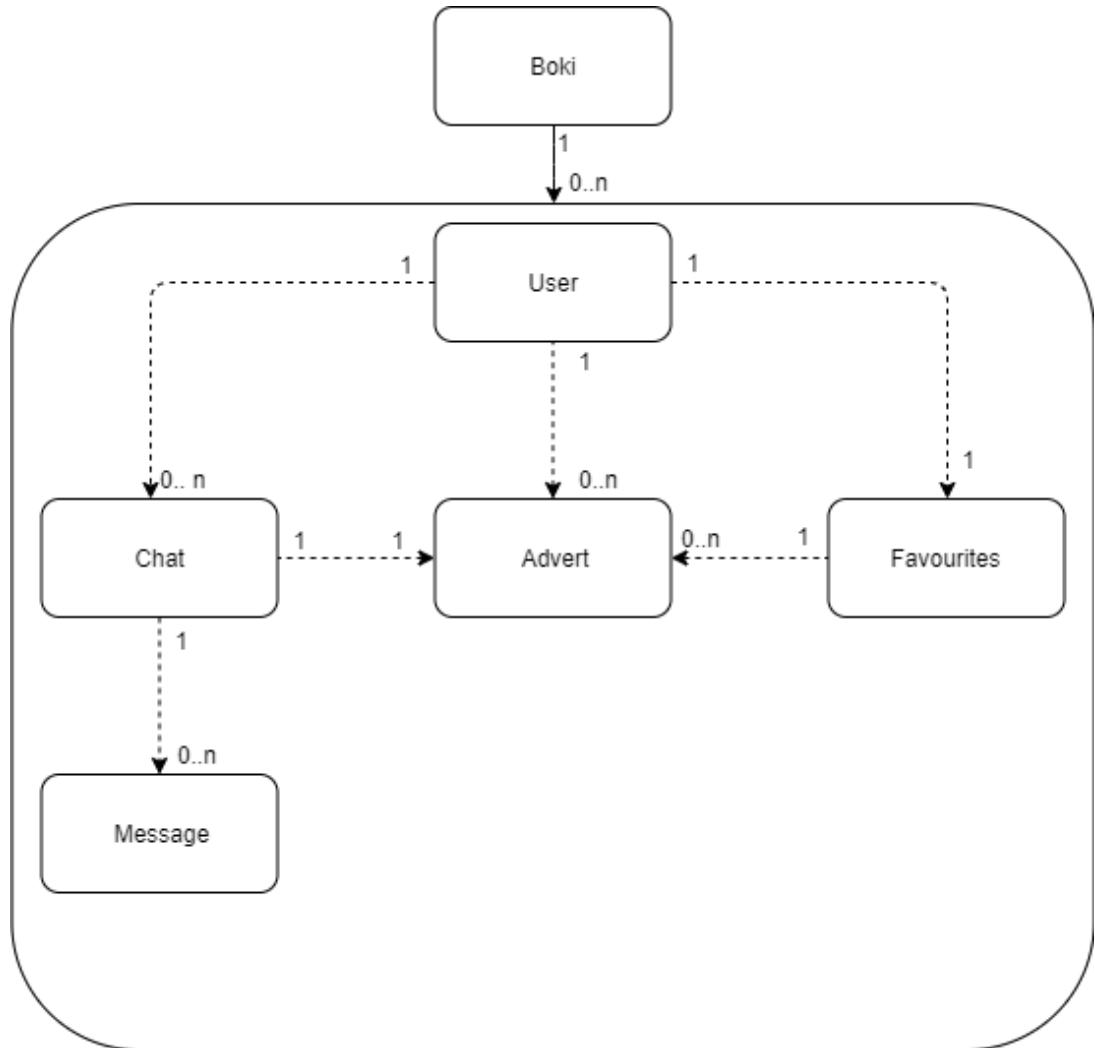
I det fall användaren inte laddat upp några annonser visas ett meddelande som förklarar detta. Dessutom visas en knapp med texten "Sälj din bok", via denna kan användare snabbt navigera till rätt vy för att ladda upp en ny annons, se figur 13b. I det övre högre hörnet av vyn visas en knapp med texten "Logga ut". Om användaren klickar på denna loggas de ut ur applikationen och navigerar automatiskt till applikationens inloggningsvy.



(a) Enhetens kameraapplikation som startas via applikationen för att ta en bild till annonser.

(b) Den uppdaterade vyn där den tagna bilden visas.

2.2 Domänmodell



Figur 14: Domänmodell.

2.2.1 Entiteter och deras ansvar

Följande delkapitel ger en övergripande bild över de entiteter som utgör applikationens domänmodell samt vilka ansvarsområden dessa har.

2.2.1.1 User

Domänmodellens **User** utgör en användare av applikationen. Denna innehåller alltid en samling av favoriter och annonser. Som entitet representeras dessa samlingar av **Favourites** och **Advert**. En användare kan antingen ha inga eller ett obegränsat antal av entiteten **Advert**, och har alltså multipliciteten noll till *n*. **Favourites** som konceptuellt är en samling (som även kan vara tom) har därför multipliciteten ett till ett.

2.2.1.2 Advert och Favourites

En **Advert** motsvarar en publicerad annons i applikationen. Varje annons tillhör alltid en specifik användare men kan även sparas i andra användares favoritsamlingar. Därför finns det även en koppling mellan **Advert** och **Favourites** med multipliciteten noll till n . Entiteten **Favourites** kan alltså vara en tom samling vilket i praktiken betyder att användaren inte har sparat några annonser till sina favoriter.

2.2.1.3 Chat och Message

Entiteten **Chat** representerar en konversation mellan två användare. En användare har alltid mellan noll och n chattar. Något som inte visas i domänmodellen är att varje **chat** vet om vilka som deltar i konversationen genom att den har id:n samt användarnamn på dessa. Detta för att en konversation endast ska kunna skapas från en annons då chattens syfte är att skapa kontakt mellan köpare och säljare av en bokannon, vilket ger ett 1:1 förhållande mellan **Chat** och **Advert**. Detta förhållande är dock något förenklat, i praktiken kan det finnas flera chattar (det vill säga en multiplicitet på noll till n från **Advert** till **Chat**) mellan olika användare som berör samma annons.

Entiteten **Chat** representerar en konversation mellan två användare, vilket åskådliggörs i domänmodellen där multipliciteten mellan **Chat** och **User** är ett till två. Varje konversation måste även innehålla minst ett meddelande, vilket ger multipliciteten ett till n till entiteten **Message** som representerar ett meddelande. En konversation kan bara skapas från en annons då chattens syfte är att skapa kontakt mellan köpare och säljare av en bokannon, vilket ger ett 1:1 förhållande mellan **Chat** och **Advert**. Detta förhållande är dock något förenklat, i praktiken kan det finnas flera chattar (det vill säga en multiplicitet på noll till n från **Advert** till **Chat**) mellan olika användare som berör samma annons.

3 System Design Document

Denna del av rapporten redogör för hur uppbyggnaden av android-applikationen Boki utformats, både på en översiktig och en mer detaljerad nivå. Dokumentet gör även en ansats till att redogöra för de ställningstaganden som gjorts under utvecklingsarbetet, och försöker redovisa varför applikationen har utformats på just detta sätt.

Den översiktliga beskrivningen ges via en sammanfattad framställning av den huvudsakliga systemarkitektur som använts, en skildring över vilka externa komponenter som använts samt vilka ansvarsområden dessa komponenter har. Det ges även en överskådlig redogörelse för de vyer som utgör majoriteten av applikationens användningsområden samt hur navigationen i och mellan dessa utformats. På en lägre, mer detaljerad nivå redovisas applikationens design- och domänmodell i form av UML-diagram, samt de två modellernas relation till varandra. Det ges även en djupare inblick i hur applikationens systemarkitektur faktiskt implementeras. I samband med detta visas även hur applikationskoden delats upp i diverse kodmoduler och hur dessa moduler beror på varandra. Här framförs även vilka designmönster som tillämpats i applikationen samt var i koden dessa mönster implementeras.

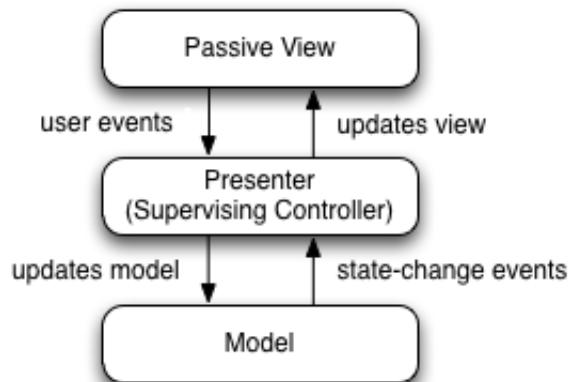
Slutligen beskrivs hur applikationen implementerat beständig datahantering, hur projektkoden testats samt var dessa test återfinns. Dessutom listas alla kända problem och buggar som finns i applikationen,

det ges även en redogörelse för hur applikationens åtkomstkontroll och säkerhet utformats.

3.1 Systemarkitektur

Boki är en androidapplikation vars funktionalitet från ett användarperspektiv består av interaktion med andra användare via kontakt med en gemensam databas. Applikationen använder sig av systemarkitekturen MVP, vilket förenklat innebär att modellen separeras från vyn med hjälp av ett presenter-lager. Utöver detta använder applikationen en databas som skapas med hjälp av Google Firebase [3]. Förutom att användas för lagring av information som sedan kan delas mellan alla användare ansvarar även Firebase för autentiseringen av dessa användare [4]. Modellen innehåller den data och logik som applikationen behöver, vilken den läser från respektive skriver till databasen via backend-paketet. Detta för att modellen inte heller ska ha något direkt beroende på en konkret databas.

I denna applikation används systemarkitekturen MVP, och delar in kodén i tre huvudsakliga moduler; Model, som endast innehåller javakod, detta är kärnan i programmet och detta som övriga moduler utgår ifrån. Den andra modulen är Presenter, det paket som översätter modellen i form av javakod till sådant som vyn i sin tur klarar av att visa upp. Den sista modulen är View, som beror på kod som importeras från android-bibliotek. Detta paket består av Fragments och Activities vars uppgift är att visa upp datan från modellen i form av ett gränssnitt. För att se hur detta implementeras mer specifikt i applikationen se avsnitt 3.2.2



Figur 15: Visualisering av en MVP-struktur.

3.1.1 Google Firebase

All hantering av beständig data i applikationen, oavsett om det är i form av annonser, meddelanden eller användarkonton, sker via Googles backend-lösning Firebase. Firebase är en plattform för utveckling av webb- och mobilapplikationer som tillhandahåller lösningar av ett flertal olika funktioner, exempelvis användarautentisering, molnbaserade meddelandetjänster, datalagring samt upp- och nedladdning av filer. De tjänster som applikationen nyttjar är uppdelade i tre olika moduler som är inriktade på var sitt specifika ansvarsområde. Dessa moduler är:

- Authentication - Denna modul ansvarar över hantering av användarkonton och förser funktionalitet

för att skapa, redigera samt ta bort användarkonton. Utöver detta ansvarar även denna modul över applikationens användarautentisering.

- Cloud Firestore - Denna modul utgör den databas som applikationen använder sig av. I denna sparas dataobjekt som innehåller beständig data för annonser, chattar, meddelanden samt användare. Databasen i sig är strukturerad på ett hierarkiskt sätt i så kallade Collections och Documents. En Collection utgör här en samling dokument där varje dokument i sin tur innehåller den data som associeras med ett specifikt objekt. Den hierarkiska strukturen möjliggör även placeringen av Collections i varje enskilt Document, något som kan leda till en stor och svårnavigerad databas.
- Storage - Denna modul ansvarar för lagringen av de filer som laddas upp via applikationen, i detta fall de bildfiler som hör till de uppladdade annonserna. Utöver lagringen ansvarar även Storage-modulen för upp- och nedladdningen av de lagrade filerna.

I applikationen sker all kommunikation med samtliga moduler via referenser till Java-objekt som nås via importerade kodbibliotek, ytterligare detaljer kring implementationen av dessa återfinns i avsnitt ??.

3.1.2 Applikationens Flöde

När applikationen startas visas en av två möjliga vyer för användaren beroende på om denne sedan tidigare varit inloggad i applikationen. Ifall användaren vid start av applikationen inte befinner sig i inloggat tillstånd möts denne av en vy som ber den logga in eller registrera sig. Användaren kan hamna i detta tillstånd om det antingen är första gången den använder applikationen eller ifall den stängde av programmet i utloggat tillstånd. Implementationsmässigt består verifikationen över vare sig användaren befinner sig i ett inloggat tillstånd eller inte av ett metodenanrop till den Firebase-modul som ansvarar över autentisering av användare. Detta metodenanrop har placerats i MainPresenter då den tillhör MainActivity, vilket är den kodmodul som körs först vid applikationens start. Det är även MainActivity som tillhandahåller majoriteten av de Fragments som utgör applikationens vyer.

När användaren väl loggat in och givits tillgång till applikationens huvudsakliga innehåll får de navigera fritt genom de olika vyerna. Användaren är alltså inte bunden till något specifikt flöde genom gränssnittet. Applikationens användningsområden kretsar kring sex centrala vyer; Hem, Favoriter, Skapa Annons, Meddelanden, Mina Annonser samt en detaljvy över en enskilda annonser.

3.1.2.1 Hem

I Hemvyn visas alla annonser som användare lagt upp på marknaden. Denna vy är implementerad som ett Fragment tillhörande MainActivity. Innehållet i denna vy är troligen det de flesta användare är mest intresserade av och därför är det den första vyn som visas för användare när de loggar in i applikationen. Här visas annonserna för användaren i form av en lista med två spalter som innehåller produktkort. Dessa produktkort innehåller annonsens titel, pris samt en tillhörande bild. Om användaren klickar på något av dessa produktkort navigerar applikationen till detaljvyn för motsvarande annons. Dessa produktkort är implementerade som ViewHolders vilka binder sig till en RecyclerView, denna utgör själva listan annonserna placeras i. Utöver listan med tillgängliga annonser finns det även en sök- samt en filtreringsfunktion. Sökfunktionen har implementerats via ett textfält som tar emot inmatning från användare.

När textfältets innehåll ändras skickas söktermen vidare till klassen SearchHelper som placeras i applikationens utils-paket. Denna sorterar sedan fram de annonser som matchar den sökterm användaren angett och returnerar resultatet till HomeFragments presenter, HomePresenter. Denna vidarebefodrar resultatet till HomeFragment som uppdaterar vyn i enlighet med sökresultaten. Allt detta sker i realtid vid förändring i textfältets innehåll, användaren behöver alltså inte explicit anropa sökmetoden via någon särskild inmatning utöver själva söktermen.

Applikationens filtreringsfunktion är implementerad på ett liknande sätt, dock har vyn där användare specificerar de gällande filtreringskriterierna implementerats i ett Fragment separat från HomeFragment, nämligen FilterFragment. All navigering till och från detta Fragment sker via HomeFragment, så i användningssyfte kan det ses som en utvidgning av HomeFragment. Filtrering kan göras på fördefinierade ämnestaggar samt i vilket intervall annonsens pris får ligga. Filtreringen som utförs med avseende på de valda ämnestaggarna är *inklusiv*; om användaren exempelvis väljer både fysik och matematik som taggar kommer även böcker som bara är taggade med en av dessa även att tas med i filtreringen.

I denna vy har även en sorteringsfunktion implementerats. Via denna kan användare välja i vilken ordning de tillgängliga annonserna ska visas. De sorteringskriterier som finns att välja mellan är

- Senast publicerad
- Alfabetisk (A-Ö)
- Omvänt Alfabetisk (Ö-A)
- Lägsta pris
- Högsta pris

Denna sorteringsfunktion har implementerats med hjälp av en användning av Strategy Pattern. Detta designmönster har implementerats via interfacet SortStrategy som definierar det publika gränssnittet som delas av alla sorteringssätt. Varje sorteringskriterie har implementerats som en egen klass, samtliga av dessa implementerar interfacet SortStrategy. I samband med detta har det även implementerats en klass som bestämmer vilken sorteringsstrategi som skall användas, klassen SortManager. Vilken sorteringsklass som skall användas avgörs beroende på positionen av det valda sorteringskriteriumet i SortManagers lista av sorteringsstrategier. Detta medför en brist i implementationen; ordningen sorteringskriterierna visas för användarna måste överrensstämma med ordningen i SortManagers lista. Ifall detta inte skulle vara fallet kan en sortering ändå genomföras, men den sortering användaren valt och den som faktiskt utförs lär då inte vara samma vilket kan leda till stor förvirring hos användaren. Även kodens utbyggbarhet blir något sämre på grund av detta. För att implementera ett nytt sorteringskriterium måste detta läggas till i både SortManagers lista samt i den vy som ansvarar för att visa upp de tillgängliga sorteringssättet. Sorteringskriteriets index i dessa listor måste även vara samma för att sorteringen skall funka som tänkt.

3.1.2.2 Detaljvyn

Implementationsmässigt utgör detaljvyn en egen Activity separat från MainActivity. I detaljvyn visas mer genomgående information om annonsen i fråga. Utöver titel, pris och en större version av den bild som visades i produktkortet visas även annonsens beskrivning samt de taggar som annonsens ägare lagt till. Utöver den extra information som visas kan användare dessutom lägga till annonsen i sina favoriter samt

ta kontakt med annonsens ägare via en knapp som leder användaren till applikationens chattfunktion. Ett specialfall av denna vy visas när annonsens ägare själv navigerar till annonsens detaljvy. I detta fall är knappen för att favoritmarkera annonsen dold. Dessutom har knappen som öppnar chattfunktionen ersatts av en knapp som låter användaren redigera annonsen. Vyn för redigering av annonser utgör ingen egen Activity i sig, den är istället implementerad som ett specialfall av vyn för att skapa annonser. Detta blev en enkel lösning då en Activity skapad i syfte att redigera annonser skulle ha en stor mängd gemensam kod med den för att skapa annonser. Denna lösning blev kanske inte helt optimal, detta diskuteras vidare i avsnitt 4.3

3.1.2.3 Favoriter

Favorityvn visar de upplagda annonser som användaren favoritmarkerat. På detta vis ges användare ett sätt att enkelt återfinna intressanta annonser utan att behöva använda hemvyns sökfunktion varje gång de vill se annonsen. Presentationen av de favoritmarkerade annonserna är väldigt lik presentationen av annonser i hemvyn, då även dessa presenteras via produktkort. Denna vy är dock lite mer avskalad än hemvyn då den saknar både sök- och filtreringsfunktionalitet. Implementationsmässigt sker presentationen av annonser på samma sätt som i HomeFragment där varje annons utgör en ViewHolder som binder sig till en lista i form av en RecyclerView.

3.1.2.4 Skapa annons

I likhet med detaljvyn är även denna vy implementerad som en egen Activity, separat från MainActivity. I denna vy möts användaren av en uppsättning textfält och knappar genom vilka de kan tillföra den information som krävs för att skapa en ny annons. Utöver den textbaserade information som krävs för uppladdningen av en annons måste användaren ladda upp en bild som tillhör annonsen. Detta görs via en knapp i vyn som startar kamera-applikationen på den aktuella android-enheten. När användaren angivit den information som krävs för att ladda upp annonsen kan de slutföra processen genom att klicka på knappen med texten "Lägg upp annons". Om nödvändig information saknas är "Lägg upp annons"-knappen en grå nyans och saknar funktionalitet. Den återaktiveras först då all obligatorisk information är angiven.

Om användare vill redigera en annons de laddat upp tidigare visas en variant av denna vy; knappen för att ladda upp en annons har ersatts av två andra knappar. Den ena av dessa tar bort annonsen från marknaden och den andra skickar annonsens uppdaterade information till databasen så att dessa ändringar reflekteras i annonsen på marknaden.

3.1.2.5 Meddelanden

I denna vy visas en överblick av de konversationer användaren deltar i. Varje konversation representeras genom en vertikal rad som visar relevant data om konversationen. Informationen som visas är namnet på den andra deltagaren i konversationen, tidpunkten när det senast skickades ett meddelande i konversationen samt bilden som hör till den associerade annonsen. Om användaren klickar på någon av dessa navigerar den till en chattvy där den kan se vilka meddelanden som skickats tidigare men även skicka nya meddelanden till konversationens andra part.

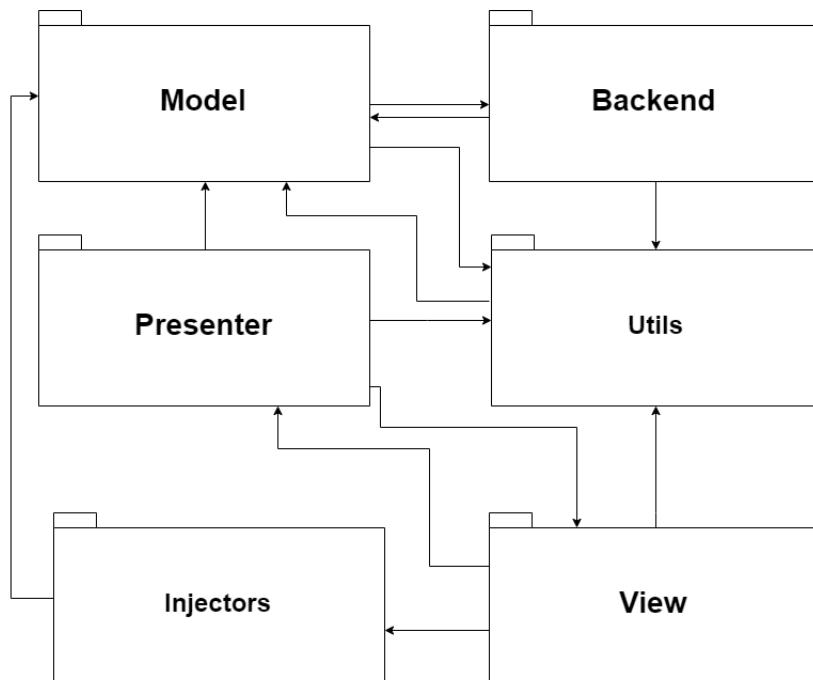
3.1.2.6 Mina annonser

Denna vy är huvudsakligen strukturerad på samma sätt som Hem- och Favoritvyn; annonser i form av ViewHolders visas i en lista som implementationsmässigt utgörs av en RecyclerView. Det som särskiljer denna vy från de andra är faktumet att det är de annonser som den inloggade användaren själv lagt upp som visas. Utöver detta fungerar även denna vy som en slags profilsida, det är nämligen via denna vy användare loggar ut ur applikationen. I likhet med favoritvyn innehåller inte heller denna vy sök- eller filtreringsfunktionalitet.

3.2 Systemets design

3.2.1 Beroenden mellan paketen i applikationen

Utöver paketen Model, View och Presenter finns även Injectors som används för att minska direkt beroende på modellen, Utils för nödvändiga hjälpklasser som används i flera delar av applikationen samt Backend som bistår modellen med beständig datahantering. Paketens förhållande till varandra går att se på en övergripande nivå i figur 16.

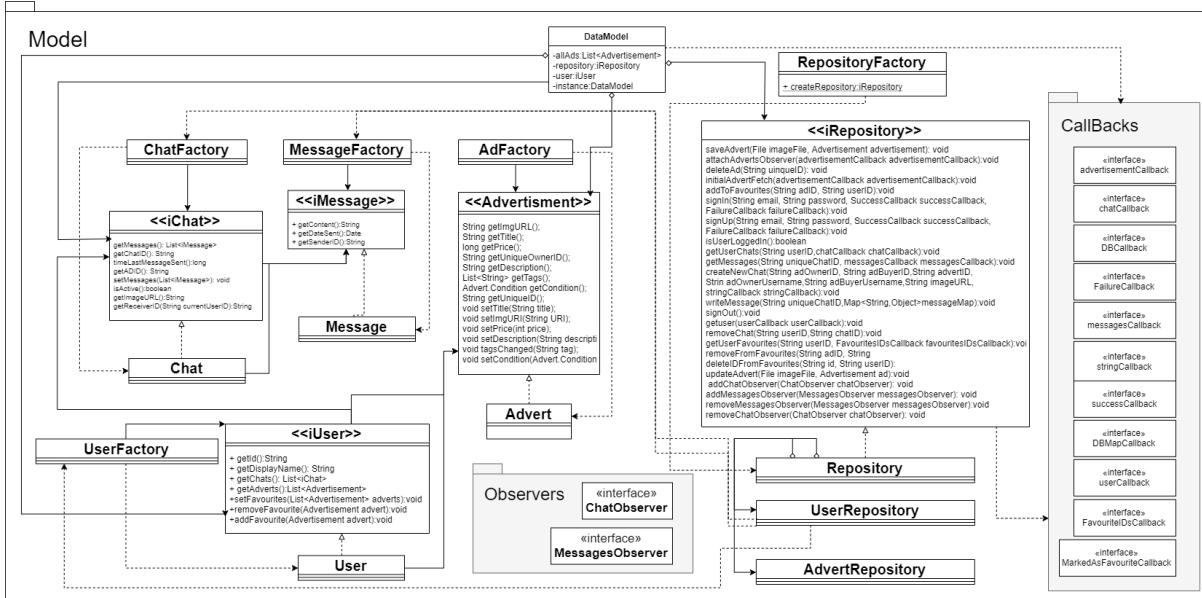


Figur 16: översikt över paketen och dess beroende till varandra

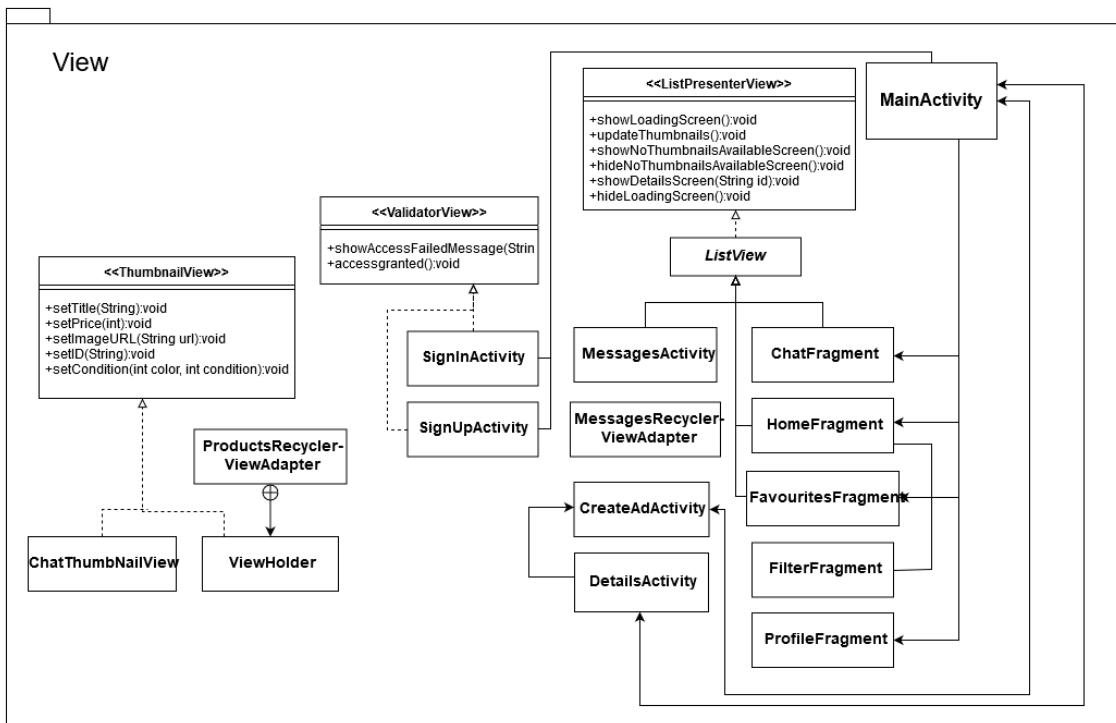
Modellpaketet har ett beroende på Backend då DataModel anropar BackendFactory som returnerar en instans av interfacet iBackend. Designvalet togs med bakgrund till att det svaga beroendet på Backend paketet ansågs godtagbart. Däremot medför det att BackendFactory och IBackend måste importeras och att modellpaketet inte helt enkelt kan tas ut och användas i en annan miljö. Med en längre tidsram hade med fördel en refakturering genomföras där ett mellanlager används för eliminera beroendet från

modellen och därmed öka modulariteten.

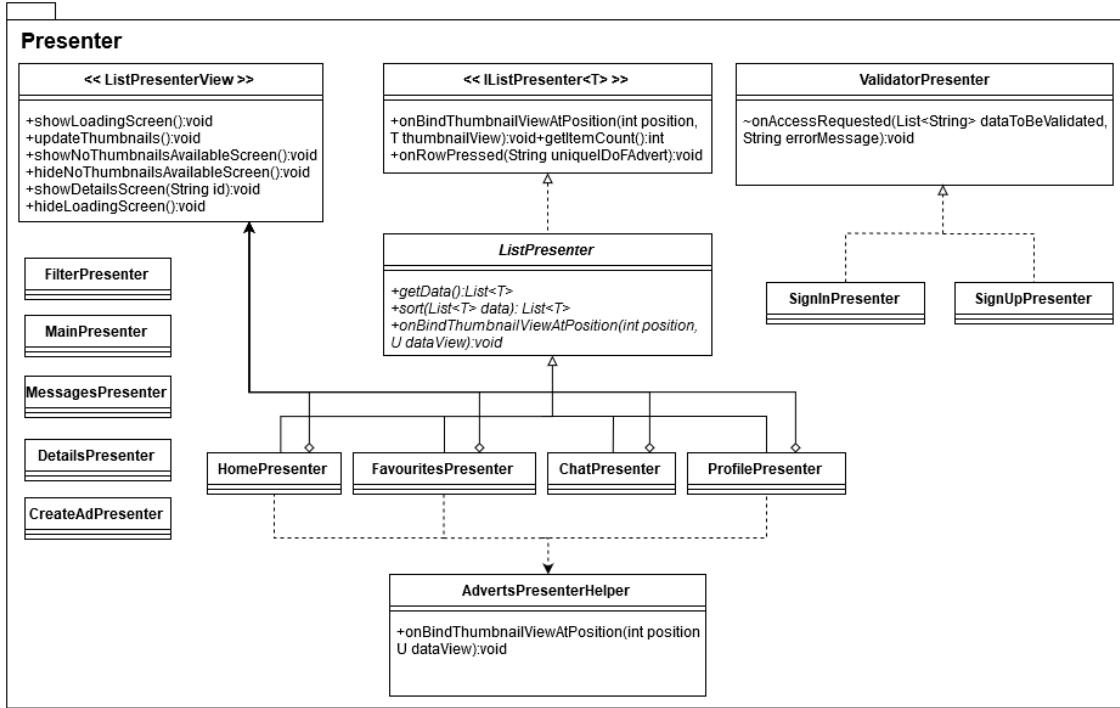
Nedan visas alla paket på en mer detaljerad nivå, vilka klasser och interface de innehåller och hur dessa relaterar till varandra.



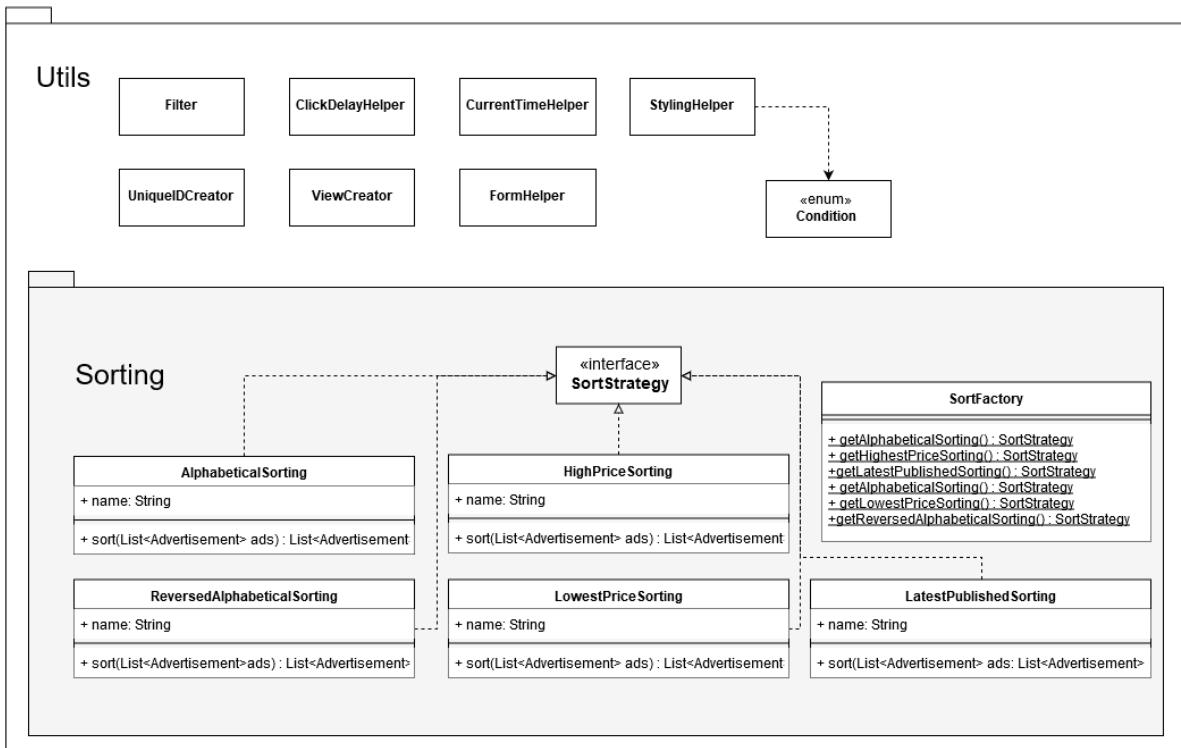
Figur 17: Modellen, kärnan av applikationen



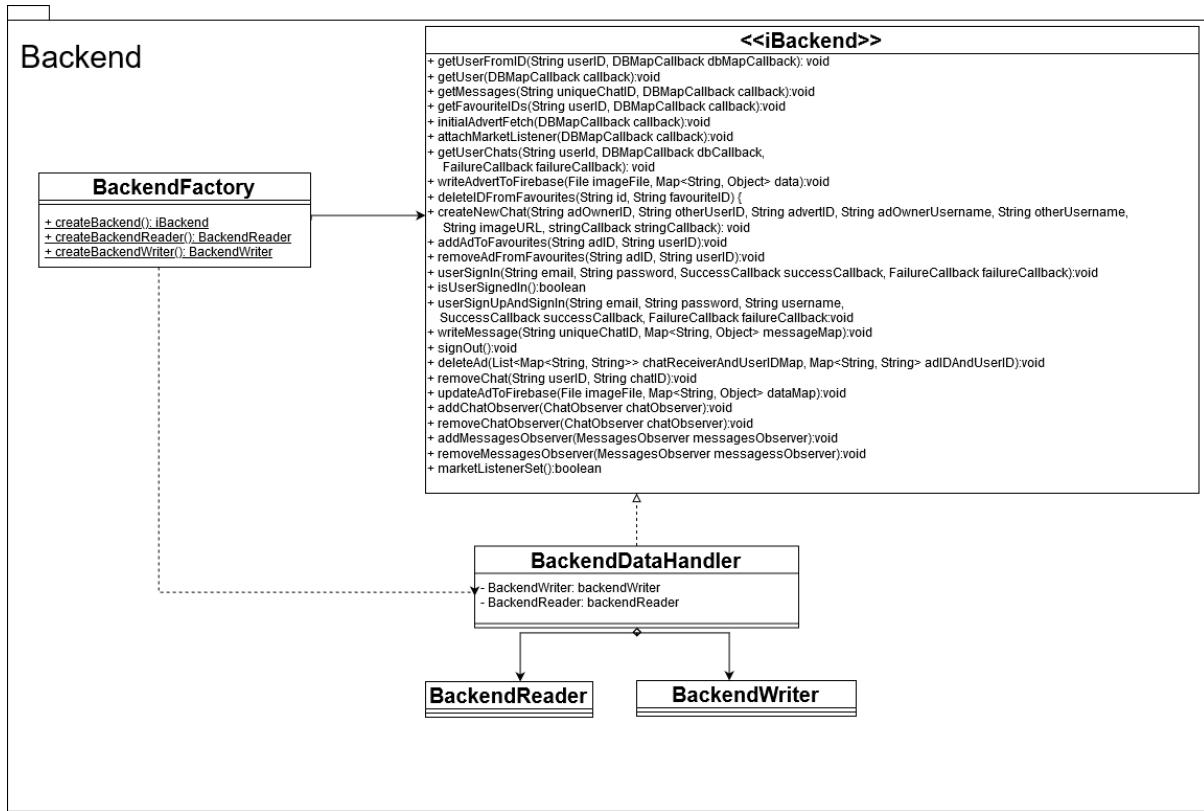
Figur 18: View-paketet med all androidspecifik kod.



Figur 19: Presenter-paketet som är länken mellan View och Model



Figur 20: Utils-paketet med alla nödvändiga hjälpklasser.



Figur 21: Backend

3.2.2 Model, View, Presenter

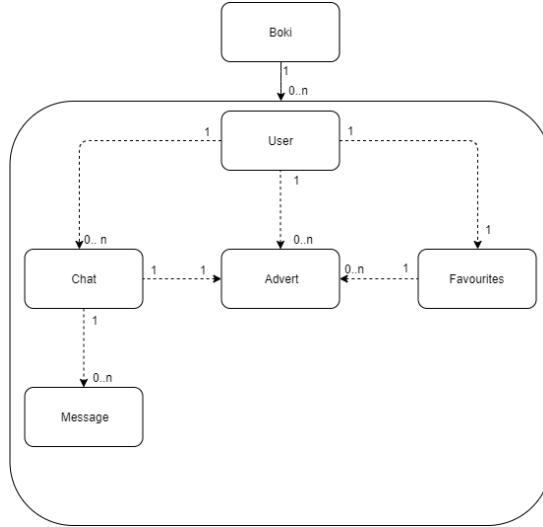
Model-View-Presenter är en systemarkitektur som ofta används vid utveckling av androidapplikationer och baseras på den traditionella Model-View-Controller-arkitekturen (MVC). MVP består av tre moduler, Model, View och Presenter [5] som har ett ansvarsområde var. Modellen innehåller enbart applikationens domänlogik, presenter agerar som mellanhand mellan vy och modell där all interaktion sker via presentern. Vyns uppgift är i sin tur att visa upp ett gränssnitt av den information som ges av presentern. Denna kedja går även åt andra hålet, där vyn meddelar presenterlagret om att exempelvis användaren matat in data i ett textfält och Presentern uppdaterar då i sin tur modellen. Alla vyer (Activities och Fragments) har ett direkt beroende till sina respektive presenter-klasser, då presenter skapas i vyns konstruktör och skickar med en instans av sig själv i konstruktorn. För att inte skapa för starka beroenden implementerar alla Activities och Fragments ett interface med de metoder som dess presenter behöver för att sköta interaktionen med vyn.

Majoriteten av presenter-klasserna har en instansvariabel av typen DataModel. Beroendet på denna klass skapas genom att det, i konstruktorn i respektive presenter, injiceras via en separat statisk klass vid namn 'DependencyInjector'. DataModel agerar som fasad för modell-paketet och står för majoriteten av alla interaktioner med detta paket. DataModel existerar för att kunna veta i vilket tillstånd applikationen befinner sig i. Den håller en lista med alla annonser, den nuvarande användaren i form av ett iUser-objekt samt en referens till ett Repository-objekt. Repository sköter all interaktionen med backend-paketet, och

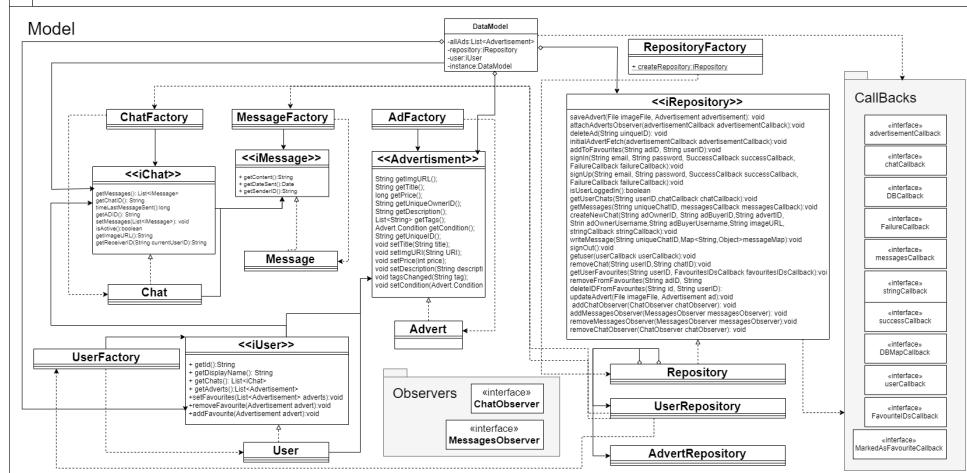
separerar modellen från backenden. Detta görs genom att det delegerar vidare anrop till UserRepository respektive AdvertRepository. Dessa sköter omvandlingen av data i form av klasser från modellpaketet och till samlingar av exempelvis strängar och heltal som kan sparas i databasen.

Tanken med denna uppdelning är bland annat att följa SoC genom att dela upp ansvarsområden mellan olika paket. Detta är något som i sin tur teoretiskt sett ska leda till att både vy och presenter ska kunna vara utbytbara utan att modellen ska behöva ändras, och en migration från en plattform till en annan ska vara enkel att utföra.

3.2.3 Domänmodell och Designmodell



(a) Domänmodellen för applikationen.



(b) Designmodellen för applikationen.

Figur 22: Domänmodellen och designmodellen

I applikationens designmodell återfinns implementationsmässiga motsvarigheter till nästan samtliga entiteter i form av konkreta klasser (som i vissa fall även har tillhörande interface), se figur 22. Eftersom att

entiteterna i domänmodellen tydligt kan knytas till en viss klass i applikationens designmodell förhåller sig dessa modeller till varandra på ett bra sätt.

Den implementationsmässiga motsvarigheten till entiteten **User** utgörs av Java-klassen User. En av instansvariablerna,’favourites’ representerar i praktiken entiteten Favourites. Detta är den enda entiteten som inte representeras av en konkret klass, utan utgörs i User-klassen av en lista av annonser. Denna lista instansieras alltid och existerar därmed även om antalet annonser är noll i det fall användaren inte har sparat några favoriter, vilket korresponderar väl med domänmodellen. Annonserna i favorit-listan är av typen Advertisement som är ett interface för den konkreta klassen Advert, vilka båda utgör en representation av entiteten **Advert**. Klassen Advert innehåller en referens till dess ägare via en instansvariabel där ägarens användar-ID finns sparad vilket kopplar ihop **User** med **Advert**.

Entiteten **Chat** motsvaras i Java-kod av klassen Chat. Alla instansierade objekt av denna klass innehåller alltid en annons av interfacetypen Advertisement, två användar-id:n, två användarnamn och en lista av meddelanden innehållandes objekt av interfacetypen iMessage, som konkretiseras av klassen Message.

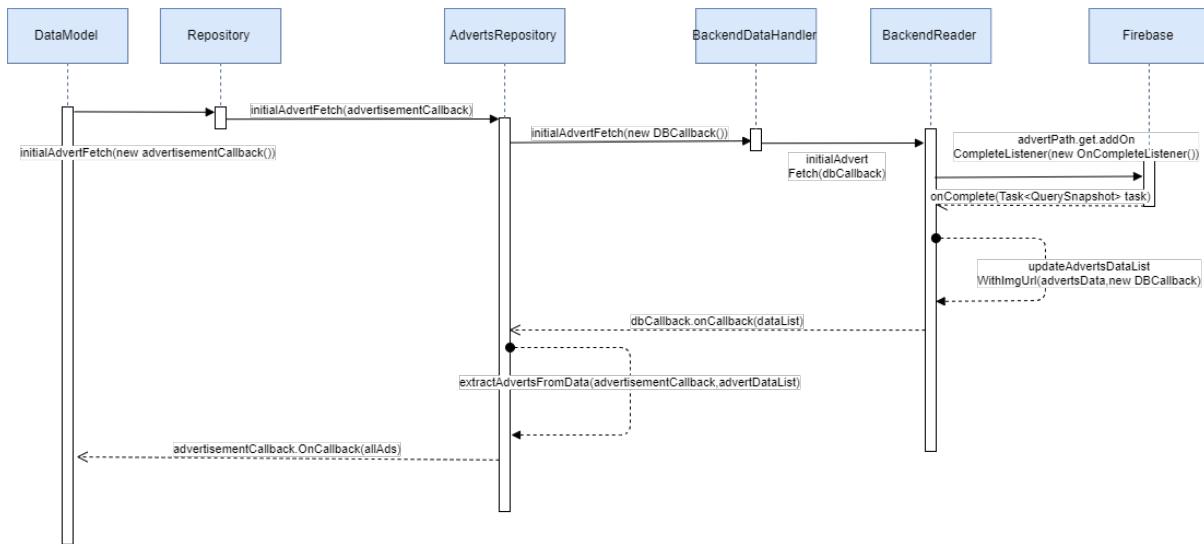
När objekt av typen Message instanseras sätts alltid referenser till sändaren genom användar-ID:t samt meddelandets innehåll och vilken tid meddelandet skickats. Ett meddelande har således ingen vetskaps om vilken konversation den tillhör, utan får endast användar-ID:t från det Chat-objekt som håller instansen av meddelandet.

3.2.4 Callbacks

En konsekvens av att bygga en applikation som implementerar användandet av en databas är hantering av den fördröjning som uppstår mellan metoden anrop och återkoppling från databasen. För att motverka att denna fördröjning skapar problem, som av sin natur gör att koden körs asynkront, används ”Callbacks”. Dessa Callbacks ersätter klassiska return-satser och kan skickas som parametrar i till de metoder som gör anrop till databasen. Detta innebär i praktiken att en metods return-sats kan styras till att köras först när svar från databasen anlänt. Något som alltså ger en möjlighet att återställa ett i övrigt asynkront metoden till att vara synkront.

Callbacken implementeras som funktionella interfaces som skickas med som parametrar fram till metoden där databasanropet körs. Efter att databasanropet är färdigt och datan från anropet är omhändertagen kan Callbacks OnCallback-metod anropas. Denna metod körs då där detta specifika Callback skapades och med den specifika implementation detta Callback har. Det finns flera olika sorters Callbacks, men principen är densamma för alla. På grund av många olika returtyper från databasen finns det många olika Callbacks, dessa kanske hade gått att göra mer generiska och återanvändbara.

Callbacks kan öka komplexiteten och göra koden mer svår begriplig. I sekvensdiagrammet (se figur 23) åskådliggörs hur Callbacks används i applikationen när annonser ska hämtas från databasen.



Figur 23: Ett exempel på hur en metodkedja kan se ut med Callbacks och anrop till databas.

3.2.5 Hantering av beständig data

Inom projektets omfattning utgör användandet av en färdig databas som Firebase en bra och enkel lösning som ligger i linje med den kunskapsnivå som förväntas av applikationens utvecklare. Trots att applikationen implementerar en databastjänst som minimerar den kunskap och kodmängd som krävs för implementationen utgör tjänsten **inte** en färdig lösning som kan införas i applikationen utan att ytterligare kod tillförs.

Databasen har implementerats via referenser till Java-objekt från kodbibliotek som importeras från Firebase. Dessa objekt har förutbestämda ansvarsområden och har på grund av detta metoder för hantering av en specifik typ av data. Exempelvis definieras ett särskilt objekt för hantering av användardata samt autentisering av användare och ett annat objekt som hanterar av skrivningar till och läsningar från databasen.

I applikationen har all kod som beror på referenser till dessa Firebase-objekt placerats i kodens backend-modul, specifikt i klasserna BackendReader och BackendWriter, vilka hanterar läsningar från respektive skrivningar till databasen. Själva kopplingen mellan applikationen och Firebase görs via en konsol på Firebase hemsida. Här ges även en överblick av olika delar av de lösningar som implementerats, samt alla tjänster som erbjuds. Exempelvis går det se vilka registrerade användare som finns och det erbjuds en översikt över alla objekt som finns i databasen. Denna data kan direkt manipuleras via denna konsol. Mest typiskt görs detta genom att manuellt lägga till eller ta bort olika databasobjekt eller användare, eller redigera den data som tillhör dessa.

En mycket användbar aspekt av Firebase är de unika id:n som automatiskt genereras och binds till varje nytt objekt som skapas i databasen. Det faktum att varje objekt associeras med ett unikt ID underlättar hanteringen av data i applikationen enormt, särskilt när det gäller jämförelsen mellan olika objekt eller att avgöra om de tillhör olika samlingar. Som standard genereras och binder Firebase dessa ID:n på egen hand. I just denna applikation genereras däremot dessa ID:n i själva applikationen istället

för i dess databas, då detta underlättade att namnge bildfilen med ID:et till den annons bilden associerades med.

Strukturellt sett är applikationens databas uppdelad i tre olika delar, Market, Users samt Chats. Dessa utgör vad Firebase kallar för *collections*, alltså samlingar av dokument där varje dokument i sin tur innehåller den data som tillhör ett objekt.

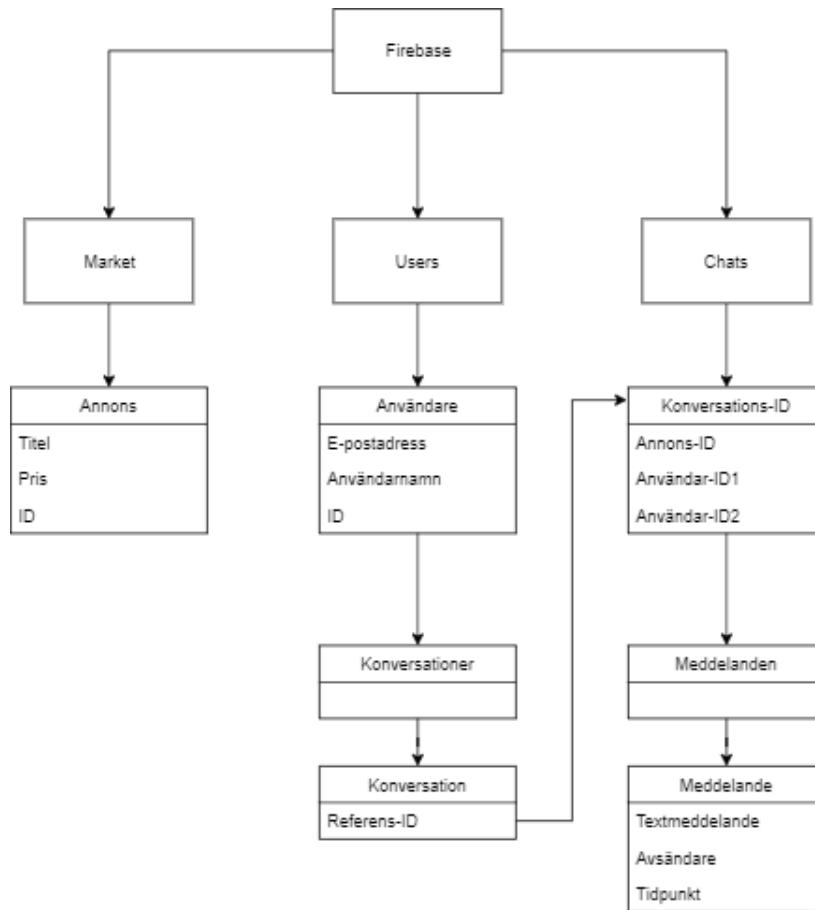
Samlingen Market innehåller alla annonser som ligger uppe för försäljning i form av ett dokument per annons. Varje sådant dokument innehåller i sin tur den data som återfinns i motsvarande Java-objekt; exempelvis bokens titel, pris och beskrivning.

I samlingen Users sparas varje användare som ett eget dokument. Detta dokument innehåller användarens e-postadress, användarnamn samt ett unikt användar-ID. I det fall användaren har påbörjat en eller flera konversationer med någon annan användare via applikationens meddelandefunktion återfinns dessa i en egen samling som tillhör det aktuella användarobjektet. Varje dokument i denna samling representerar en viss konversation. Konversationsdokumenten i användarens samling innehåller enbart konversationsens unika ID; inga meddelanden eller någon övrig information om konversationen återfinns i dessa dokument.

Meddelanden och all annan information om konversationen finns i motsvarande dokument som placeras i samlingen Chats. I Chats samlas dokumenten för aktuella konversationer mellan applikationens användare. Varje konversation har ett unikt ID vilket kan användas som en referens till konversationen samt den data som associeras med denna. Detta görs exempelvis via de ID:n som sparas i användardokumenten. Varje konversationsdokument i Chats innehåller viss data utöver de meddelanden som utgör konversationen. Dokumenten innehåller de ID:n som refererar till de två användare som deltar i konversationen, samt annonsens egna unika ID.

Då det är användbart att hantera varje meddelande i en konversation som ett enskilt objekt är dessa sparade i en separat samling som tillhör den aktuella konversationen. Varje meddelande i denna samling innehåller tre saker; den text som utgör det faktiska meddelandet, det unika ID som refererar till användaren som skickade meddelandet samt vilken tidpunkt meddelandet skickades.

En förenklad visuell överblick av denna struktur ges i figur 24.



Figur 24: Förenklat flödesdiagram över databasens struktur

3.2.6 Designmönster

I detta avsnitt beskrivs vilka designmönster som används och i vilka delar av applikationen de implementerats.

3.2.6.1 Observer Pattern

Observer Pattern implementeras bland annat i BackendHandler och genom denna BackendReader och BackendWriter. ChatPresenter som implementerar interfacet ChatObservers och av MessagesPresenter som implementerar interfacet MessagesObserver lyssnar på BackendHandler för att direkt kunna uppdatera vyerna då modellen uppdaterats även från utomstående användare.

3.2.6.2 Strategy Pattern

SortManager implementerar Strategy Pattern genom att ta in olika strategier beroende på vilken sorts sorteringsmetoder enkelt ska kunna läggas till.

3.2.6.3 Singleton Pattern

Singleton återkommer i flera klasser där man vill hålla data under körningen av applikationen utan att denna ändras. Detta mönster implementeras av SortManager, Filter och DataModel. DataModel får inte ändra tillstånd och måste till exempel kunna kontrollera att det endast finns en inloggad användare. De andra är singletons då de är hjälpklasser som också ska kunna erbjuda ett konsekvent beteende i tjänsterna de erbjuder.

3.2.6.4 Template Method Pattern

Template Method Pattern används på ett flertal ställen i ListPresenter där metoder innehåller ett delbeteende av metoden men där skillnaden mellan subtypernas beteende i metodanropet implementeras genom en abstrakt metod, exempelvis den abstrakta metoden `getData()` i metoden `updateAdverts()`. Även ListView implementerar detta mönster, exempelvis genom den abstrakta metoden `OnSetUpHeaderLayout()` som används i `setUpHeader()` och avgör hur vyns sidhuvud ser ut beroende på vilken subklass som anropas.

3.2.6.5 Övriga mönster

Det finns även ansatser till implementation av andra designmönste, till exempel Factory Pattern. De *factories* som används är följer dock inte mönstret exakt utan används mest för att minska beroenden på konkreta konstruktorer. Det går även att argumentera för att klassen DataModel är använder en form av Facade Pattern och utgör en fasad för modellen.

3.3 Kodkvalitét

Detta avsnitt berör kodkvalitén i applikationen i avseende på testning och vilka problem som finns.

3.3.1 Testning

Testerna till denna applikation finns alla tillgängliga under sökvägen `/Boki/app/src/test/java/com/masthuggis/boki` i projektmappen. Dessa är implementerade med hjälp av JUnit, Mockito [6] och Powermock [7]. De används för att kontrollera funktionaliteten av den logik som utförs på data i applikationen. I linje med detta testas framför allt Presenter-, Utils- och Modellpaketet eftersom dessa tre moduler innehåller majoriteten av applikationens logik. Det är även enklare att skriva omfattande enhetstester för dessa paket jämfört med övriga paket i applikationen. Detta beror på att de enbart innehåller java-kod och inte beror på externa android-bibliotek eller Firebase-bibliotek.

Den enda kodmodul i applikationen som beror på externa android-bibliotek är View. I detta paket har all kod som berör applikationens användargränssnitt placerats. I linje med applikationens systemarkitektur MVP har målsättningen under utvecklingen varit att ingen logik ska placeras i detta paket. Tyvärr återfinns ändå en liten mängd logik i detta paket. Om detta inte var fallet hade paketet inte medfört någon logik att skriva enhetstester för, vilket hade berättigat avsaknaden av enhetstester i paketet.

Backend-paketet som har olika Firebaseberoenden är även detta svårt att skriva tester för när den beror på ett externt API. Givet mer tid för refaktorering där Dependency Injection av Firebase-modulerna skulle kunna appliceras i BackendReader och BackendWriter hade det gått lättare att skriva tester för detta paket. Det är dock inte helt okomplicerat då det kräver användande av externa bibliotek som till exempel PowerMock för att mocka statiska metoder. Detta uppenbarades först i slutet av utvecklingen av denna applikation och har i dagsläget därför inte hunnits med. Innehållet av detta paket som hade varit intressant att testa är omvandlingen från data hämtad från Firebase till javaobjekt. Detta försvaras dock av att mycket resurser går åt att mocka databasanrop. Något som endast kan ge lyckade och förväntade svar.

3.3.2 Problem

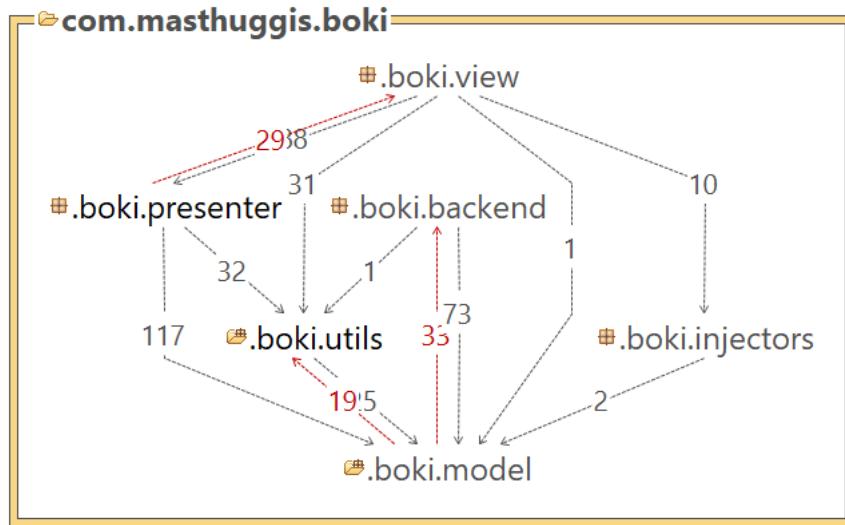
Följande lista innehåller alla kända problem i applikationens funktionalitet. Givet en större tidsram för projektet hade dessa problem kunnat åtgärdas. Vissa problem och mindre optimala kodlösningar diskuteras vidare i avsnitt 4.

- Applikationen ger ingen feedback för vad som är giltiga värden när man ska lägga upp en annons. Kan vara frustrerande att vi använder mönstret *Responsive Enabling* utan en förklaring om när man kan gå vidare. Här hade *Same Page Error Messages* med fördel kunnat implementeras.
- Det finns ingen granskning eller kontroll av innehållet som publiceras i applikationen. I ett verkligt scenario krävs moderatorer eller algoritmer som hanterar olämpligt innehåll, samt en anmälnings eller rapporteringsfunktion.
- Det skapas en ny konversation så fort en användare trycker på ”kontakta användare” i en detaljvy, oavsett om denne väljer att skicka ett meddelande eller inte. Chatten blir synlig först när ett meddelande skapats, men bryter mot domänmodellen.
- Användaren kan inte ta bort sina egna chattar, dessa tas enbart bort automatiskt när annonsen tas bort.
- Om en annons tas bort tas alla dess tillhörande chattar också bort. Användaren får ett tillfälligt meddelande när den går in i sin chattvy om en konversation har tagits bort men det kan vara enkelt att missa och förvirrande för användaren.
- Användaren får inga notiser om denne fått ett nytt meddelande, och kan inte särskilja chattar med olästa meddelanden från andra. Endast tidsstämpeln på chatten uppdateras, vilket gör att chatten med nyast meddelanden alltid hamnar överst i listan.
- Det finns en fördröjning när en användare tar bort en annons, vilket gör att annonsen fortfarande är synlig i någon sekund när den är borttagen. Den går däremot inte att öppna, och försöker användaren klicka på annonsen får den ett meddelande om att annonsen inte längre är tillgänglig.
- Sökfunktionen är inte helt intuitiv. För att få bort irrelevanta resultat visas inte matchningar mitt i en tagg eller en titel mot en sökning på en enskild bokstav. Enbart annonser där titelns inledning matchar en enskild sökbokstav visas direkt. För att få en sökning på en tagg eller en del av en titel

krävs en matchning på fler än tre bokstäver, men innan dessa tre bokstäver skrivits in kan texten ”Det finns inget som matchar din sökning” visas.

- Sökfältet har ingen knapp för att ta bort eller avbryta en sökning, användaren måste radera texten i sökfältet för att ta bort sökningen vilket kan vara frustrerande.
- Sorteringen bör ha ett standardvärde som innebär att ingen sorteringsord är vald, eftersom sökningar inte använder någon av standardsorteringarna.
- Det finns inget implementerat sätt att återställa sina kontouppgifter ifall användaren glömt dessa.
- I dagsläget finns inget sätt för en användare att ta bort sitt konto. Detta tillsammans med att det inte finns någon implementation för återställande av lösenord gör att applikationen vore svår att lansera.
- Filtreringsfunktioner är antingen inkluderande (inkluderar matchningar mot något av alla filter som valts) eller exkluderande (alla filter som valts måste uppfyllas för att det ska matchas mot en annons). Detta kan vara svårt att som användare förstå vid första anblick, och i Boki är filtreringsfunktionen inkluderande.
- När en annons skapas eller redigeras kan en användare lägga till egna taggar. Dessa tas bort genom att klicka på dem, men det går även att ta bort en tagg genom att skriva in samma tagg en gång till vilket kanske inte användaren förväntar sig.
- Om en användare som har installerat den senaste versionen av applikationen blir kontaktad av en användare med en tidigare version av applikationen uppstår problem. Detta beror på en omstrukturering av konversationernas plats i databasen och därmed applikationens sökväg till dessa. Till följd av detta finns risk för att NullPointerExceptions kastas och att applikationen därmed kraschar varje gång den kontaktade användaren försöker logga in.
- Finns i dagsläget ingen implementerad time out eller liknande för datahämtningar. Vilket skulle kunna innebära att att programmet fastnar i laddningssekvenser om applikationens internetuppkoppling slutar fungera under en databashämtning.

3.4 Kodanalys



Figur 25: STAN-genererat beroendediagram av paketen i applikationen.

Enligt diagrammet ovan ser beroendena mellan moduler ut som tidigare beskrivet med undantag för beroendet mellan view och model-paketet. Detta beroende har inte helt kunnat lokaliseras i koden, så det finns ingen tydlig förklaring till detta. Enligt STAN [8] är anledningen att CreateAdActivity har en referens till DataModel i sin onCreate-metod. Detta verkar inte stämma då CreateAdActivity varken importrar DataModel eller använder sig av innehåll i någon Bundle i dess onCreate-metod.



Figur 26: STAN:s förklaring till varför beroendet mellan view och model existerar.

3.4.1 Tillgänglighetskontroll och säkerhet

Det tillgänglighetskrav som ställs på användarna av applikationen är att alla användare måste vara registrerade samt inloggade i applikationen. Alla konton är av samma typ, dvs. det finns inga 'admin-konton'. Som användare är det tillåtet att registrera flera olika konton. I applikationens nuvarande tillstånd erbjuds enbart inloggning via ett konto som skapats specifikt för denna applikation, det går alltså inte att använda ett konto från en annan plattform som autentisering. Ett nytt konto skapas genom att förse applikationen med ett användarnamn (behöver ej vara unikt), en giltig e-postadress samt ett giltigt lösenord, som autentiseras av den tidigare nämnda Firebase-konsolen. Firebase står med andra ord för säkerheten kring inloggning och hashande av lösenord.

Autentiseringen av användare sköts via den Authentication-tjänst som erbjuds av Firebase. I applikationskoden används denna via ett importerat Authentication-objekt som tillhandahåller funktionalitet

rörande hanteringen av användare. Detta objekt sköter exempelvis in- och utloggning av den nuvarande användaren, samt registrering av nya användare. Ur ett programmeringsmässigt perspektiv är det svårt att avgöra hur säker denna autentisering faktiskt är då det inte ges någon större inblick i hur den konkreta funktionaliteten av denna är implementerad. Trots denna brist på insikt är det troligtvis ett rimligt antagande att autentiseringen är säkerhetsmässigt bättre än något denna applikations utvecklare kunnat utforma, då den utvecklats av ett världsledande IT-företag som Google.

Säkerheten som berör åtkomsten av objekt i databasen samt de bildfiler som tillhör de uppladdade annonserna utgörs av en uppsättning regler som definieras i Firebase. Dessa regler utgör de krav som ställs på användare för att de ska ges åtkomst till filer och olika delar av databasen. Reglernas restriktivitet kan variera enormt, från inga tillgångskrav alls, så även de som inte är registrerade användare ges tillgång, till att tillgångskraven är så restriktiva att ingen användare ges tillgång till applikationens data. Applikationens nuvarande regler ger alla autentiserade användare tillgång till den data som sparats i applikationens databas. Detta ger även teoretiskt sett alla autentiserade användare möjligheten att ta bort den data de har tillgång till. Dock lär detta inte utgöra ett särskilt stort problem då funktionalitet som implementerats i klient-sidan av applikationen ser till att annonser endast kan tas bort av deras ägare. Trots denna försiktighetsåtgärd skulle mer restriktiva regler gällande borttagandet av annonser och filer associerade med dessa självklart utgöra en säkerhetsmässig förbättring, något som med all sannolikhet skulle implementeras av en framtida version av applikationen.

4 Diskussion

I detta avsnitt diskuteras specifika delar av applikationen och de implementationsval som gjorts. Diskussionen lyfter fram ett antal exempel på funktioner som har en tydlig förbättringspotential eller inte hann implementeras. Denna applikation är samtliga projektmedlemmars första androidprojekt vilket inneburit vissa svårigheter i implementationen av androidspecifika element som till exempel Activities, Fragments och navigering mellan dessa. Mycket tid fick också läggas på att samla information och försöka förstå hur en androidapplikation utvecklas på bästa sätt. Då det upptäcktes att mycket av androids egen dokumentation hänvisar till arkitekturen MVVM hade denna kanske varit att föredra före MVP, dock fanns ingen tid till en så omfattande refaktorering.

4.1 Fragments med RecyclerView

Gränssnittets implementation består i dagsläget till stor del av fyra Fragments som använder sig av RecyclerViews. Dessa fyra Fragments är HomeFragment, FavouritesFragment, MessagesFragment och ProfileFragment. Eftersom flera av vyerna liknar varandra valde vi att skapa en abstrakt superklass, ListView, till de fragments som använder sig av RecyclerViews för att visa upp sin data.

ListView håller en XML-layoutfil som innehåller en RecyclerView och utgör baslayouten för dessa vyer. Varje fragment har sedan en egen layout-fil som ändrar utseendet på sidhuvudet för respektive vy. Genom abstrakta metoder appliceras resterande skillnader i gränssnittets utseende som exempelvis att konversationerna i MessagesFragment visas i en kolumn medan resterande vyer visar upp annonser i två kolumner.

Denna struktur löste de problem som fanns med kodduplicering, tillåter att subtypen själv definierar hur gränssnittet ska se ut och minskar antalet XML-layoutfiler. Det finns dock ett problem med detta. Arv. För att arv ska rättfärdigas enligt LSP måste fem krav uppfyllas[9]. Denna situation uppfyller alla krav förutom behovet av framtida polymorfism. Arv är därför egentligen inte den bästa lösningen i denna situation, det hade istället kanske gått att implementera en version av antingen Decorator Pattern eller delegation på ett liknande sätt som föreslås i avsnitt 4.2.

Presenter-klasserna som associeras med dessa Fragments delar även viss funktionalitet. Därmed har en liknande arvstruktur definerats för dessa Presenter-klasser som ärver från den abstrakta klassen List-Presenter. Även denna arvstruktur misslyckas med att uppfylla kravet för framtida polymorfism, en lösning byggd på delegation eller Decorator Pattern hade alltså varit fördelaktigt även här.

4.2 Sign-Up och Sign-In

Ursprungligen fanns det två Activities och två Presenters som tillhandahöll vyer för inloggning och registrering. Dock fanns det liknande funktionalitet i båda presenter-klasserna där de kontrollerade om indata givet av användaren var godkänd eller ej, varefter de reagerade på detta. Därför skapades en ny, tredje presenter vars uppgift är att tillhandahålla denna funktion genom delegering. Denna lösning är att föredra över arv då delegering är mer flexibelt och inte ställer lika höga krav som Liskov[9] gör på arv. Nackdelen med detta är en större mängd kod i och med införandet av en extra klass, men minskad koddupliceringen uppnås ändå.

4.3 CreateAdActivity och CreateAdPresenter

Om det funnits mer tid för detta projekt hade en refaktorering av CreateAdActivity och CreateAdPresenter varit högt prioriterat. I dagsläget innehåller båda klasserna en stor mängd kod, har mycket ansvar och bryter mot SRP då de används för att både redigera och skapa nya annonser. Detta var ursprungligen två separata klasser men hade så mycket kodduplicering att de slogs samman till en och samma klass. Förståelsen om att detta var ett felaktigt beslut finns, då problemet med kodduplicering borde lett till en annan lösning. En bättre lösning hade varit att låta dem dela supertyp eller använda delegering för att uppnå *code reuse*, alternativt en kombination av de två tidigare nämnda alternativen.

4.4 Navigation

Den dokumentation Android själva tillhandahåller gällande implementation av "Conditional Navigation" använder sig av arkitekturen Model-View-ViewModel (MVVM) [10], vilket i denna applikations fall skulle inneburit en refaktorering av hela kodbasen till denna systemarkitektur. Av denna anledning har enklast möjliga navigationslösningar applicerats där det krävts, samt helt prioriterats bort i vissa fall. Detta resulterade i en förändring i ett tidigare beslut gällande gränssnittsdesignen. Den ursprungliga idén var att applicera designmönstret "Deferred Choices"[1] genom att låta användaren navigera genom i princip hela applikationen innan den blev tvingad att logga in eller registrera sig. En sådan implementation vore att föredra då det ger användaren en bättre upplevelse genom att ge tillgång till applikationen utan att

registrera sig. Tyvärr valdes denna lösning bort då kunskapsökande och efterforskningar tog för mycket tid från att implementera funktionalitet med högre prioritering.

4.5 Avvikelse från MVP

Enligt MVP ska vyer innehålla en minimal mängd logik. Detta är något som alltid funnits i åtanke under utvecklingen. Det finns dock ett par brott mot detta där logik faktiskt placeras i vyer. Det är ofta som i t.ex. MainActivity där conditional navigation appliceras beroende på olika indata och information som skickas med när Activities startas. Detta beror som tidigare nämnt på att kunskapen gällande hur navigation inom Android bör implementeras är för låg. Andra exempel på detta relaterar ofta till hur saker ska visas upp. Som i t.ex. DetailsActivity där logik utförs för att visa upp ett ifyllt hjärta gällande ifall annonsen är favoritmarkerad eller inte. Att åtgärda detta skulle i de flesta fall endast innebära att ersätta logiken med ett anrop till respektive presenter som i sin tur utför logiken som krävs, varefter den säger åt vyn att agera därefter.

4.6 Gränssnitt

Gränssnittet kräver viss IT-vana, användaren måste exempelvis förstå att denne ska klicka på en fördefinierad tagg för att välja den när den skapar/redigerar en annons, eller ta bort en egendefinierad tagg genom att klicka på den. I redigeringsvyn måste även användaren förstå att det går att ta en ny bild genom att klicka på den gamla bilden. Dessa lösningar är inte helt optimala och inte heller helt intuitiva men om det funnits mer tid hade detta kunnat åtgärdas. Exempelvis hade små kryss-ikoner kunnat läggas till på en egendefinierad tagg för att enklare förstå att de går att ta bort, eller hjälptexter för att guida användaren.

4.7 Hämtning av data

Hämtning av data kan ibland ta ganska lång tid, och hur applikationen hanterar problematiken med att data måste returneras innan nya anrop körs förklarades närmare i avsnitt 3.2.4. Dock kan det finnas problem med hämtning som tar lång tid om till exempel användaren loggar ut direkt efter en påbörjad hämtning, om man öppnar en chatt där annonsen tagits bort men chattvyn ännu inte hunnit uppdateras etc. Det är svårt att säga om alla sådana utfall är ordentligt testade, men leder i de fall som testats inte till att applikationen kraschar utan tas om hand och användaren meddelas.

4.8 Beständig data under köring

För att ge användaren en bra upplevelse vid användning av appen vill utvecklare gärna tillåta *Changes in Midstream* [11], det vill säga att låta användaren navigera fritt i appen och komma tillbaka till vyer i samma tillstånd som de lämnades. Eftersom Activities och Fragments har en ändlig livscykel, kan man klassvis ändå spara tillstånd när exempelvis en Activity stängs ned med hjälp av Instance State[12]. Tyvärr misslyckades försöken med att implementera detta då det visade sig vara ganska komplext. Ingen mer tid kunde heller spenderas på detta eftersom det redan fanns andra lösningar implementerade, dock mindre optimala sådana. Till exempel finns singleton-klassen Filter som under köring av applikationen håller information om de filter användaren angett för att sedan visas upp igen i filtreringsvyn

om användaren går tillbaka dit. Denna lösning hade kunnat undvikas med Instance States alternativt MVVM-arkitekturen.

5 Peer Review

Applikationen SvetIt gör en ansats till användning av MVVM (Model, View, ViewModel) som arkitekturmönster. Utförandet av arkitekturen brister dock i flera aspekter. Som exempel på detta finns olika former av vyer (tex. fragments eller activities) i viewModel-paketet. Dessa har även ofta beroenden direkt på modellen vilket tyder på att de innehåller logik som bör placeras i modellen. Modellen är därmed inte heller isolerad från vyn, vilket gör att strukturen tappar sin modularitet och modellen skulle därmed inte kunna återanvändas ifall exempelvis en desktop-version av applikationen skulle utvecklas. Paketindelningen verkar något slumprövidigt på fler ställen, exempelvis innehåller paketet *planning* både en planner, exercises samt routines som skulle kunna placeras i ett eget paket. Många klasser har även ett användande av access-modifiers som inte verkar genomtänkt eftersom nästan alla metoder är publika. Detta bidrar till att onödigt många metoder och variabler är tillgängliga för övriga klasser, något som kan leda till low cohesion och high coupling. Paketens potential för inkapsling av kod blir därmed mer av en enbart strukturell sortering av klasserna. Med undantag för detta är innehållet i model-paketet i det stora hela enkelt att förstå, bra dokumenterat samt med en bra uppdelning mellan klasserna.

Gällande SOLID-principerna visar kodbasen på ett flertal brister. Brott mot *Single Responsibility Principle* är återkommande på både metod- och klassnivå. Exempelvis är onCreate i både *CreateSession* och *CreateRoutine* väldigt stora och innehåller mycket funktionalitet. Dessa metoder skulle kunna delas upp i ett flertal metoder för att bättre följa SRP och göra koden mer lättläslig. Ytterligare ett exempel på en metod som gör flera saker och därmed bryter mot SRP är *saveInfo()* i *FragStrRow*. Här skapas en ny *StrengthExercise* trots att inte alla fält är ifyllda och därmed borde ingen ny exercise skapas alls. Den aktuella lösningen här är att skapa objekt i onödan för att sedan filtrera bort det i ett senare skede. Detta leder till onödiga metodenanrop som eventuellt skapar fler problem än det löser.

Som tidigare nämnt förhåller sig dock java-klasserna i model-paketet bättre till SRP då klasserna inte innehåller lika mycket kod och inte har för stora ansvar. Däremot är några klasser i det minsta laget, exempelvis Repository som enbart verkar hålla en User som den kan returnera.

Kodens design bryter även mot Open-Closed Principle, vilket leder till att det är svårt att ta bort eller lägga till funktionalitet. Exempelvis används if-satser i Statistics som med hjälp av *instanceof()* kontrollerar vilken typ av exercise-objekt det är för att anropa en uppdatering av objektet. Om applikationen skulle utökas med fler typer av exercises måste denna metod byggas ut för varje ny typ av exercise som läggs till, något som gör det svårt att bygga ut applikationen. Samma typ av problem finns i metoden *isStrength()* i *CreateRoutine*, som används för att avgöra vilken typ av rutin som ska skapas. I nuläget är det antingen strength, eller inte strength, vilket implicit innebär cardio. Detta skulle skapa problem om en ny träningsform läggs till som märkbart skiljer sig från de existerande.

Ett återkommande tema i koden är att konkreta implementationer av datastrukturer används. För att göra koden mindre beroende på konkreta implementationer skulle högre abstraktioner av datastrukturer

som t.ex. List eller Map användas istället för ArrayList eller HashMap. Strängar som används är ofta hårdkodade och använder sig inte av string-resources. Detta innebär att applikationen inte går enkelt går att översätta till flera språk och att byta ut strängar blir onödigt svårhanterat.

Klassen *Statistic* implementerar interfacet *iSessionObserver* i en del av ett Observer Pattern, som lyssnar om någon ny Session lagts till. Dock verkar inte detta mönster användas eftersom metoderna för att lägga till och ta bort en *Observer* aldrig anropas. En version av Singleton Pattern verkar även implementeras i klassen Repository genom en privat statisk inre klass som alltid returnerar samma instans av Repository. Funktionaliteten i detta är samma som i ett konventionellt Singleton Pattern men det är en krångliga lösning.

StatisticsAdapter har en lista med objekt av den abstrakta typen IStatistic, detta interface innehåller två olika *integers* som anger vilken typ av interface det är samt en getter-metod för att hämta detta värde. Dock skapar detta interface inte den polymorfism den verkar ha varit avsedd för, utan läsningen av IStatistic-objekt i metoden onBindViewHolder sker genom att ta reda på vilken typ varje objekt är via getter-metoden och sedan explicit *type casta* objektet till motsvarande konkreta implementation av interfacet. Ytterligare ett exempel på en suboptimal arvsstruktur är Exercise-klassen som alla exercises ärver från. Denna klass innehåller bara ett textfält och metoden *getName()*, vilket inte bidrar till någon nämnbar code reuse och hade kunnat vara ett interface istället för att åstadkomma subtypspolymorfism.

Det finns i dagsläget kod som inte har någon märkbar funktionalitet. Till exempel finns en metod som heter *destroyFragment* i både FragCarRow och FragStrRow, som förstör det aktuella fragmentet (*this*). Innan det förstörs ändras värdena på fragmentets editTexts till diverse hårdkodade strängar till synes utan anledning. Detta hinner dock visas för användaren innan fragmentet försvinner. Tas dessa kodrader bort verkar applikationen ändå fungera likadant, och även om de borttagna raderna skulle ha någon viktig funktionalitet för appen gör detta koden svår att förstå.

Kodstilen är inte helt enhetlig; det går att urskilja att olika personer har skrivit koden. På vissa ställen används t.ex. List som statisk typ för variabler och på andra ställen används konkreta ArrayLists. I vissa klasser används Java Doc-kommentarer och i andra klasser kommentarer på en rad. En del av fragment-klasserna innehåller sin typ sist i klassnamnet, t ex CalenderFragment och GoalsFragment, vilket följer de kodkonventioner som är standard för Android. Andra använder en förkortning som t ex FragStrRow och FragCarRow. Activitiesklasserna har dock inte med "Activity" i sitt namn. Förkortningar som *Str* och *Car* används ibland för *strength* och *cardio*, men inte alltid. Inkonsekventa användningar av förkortningar och klassnamn är något som kan göra det svårt att förstå vad metoder och klasser har för funktion och ansvar vid första anblick. Särskilt då förkortningar som "str" vilket är en vanlig förkortning för String används. Effekten förmildras något med hjälp av att det finns mycket kommentarer.

Hanteringen av activities livscykler kan skapa problem då det vid navigation skapas nya activities utan att tidigare activities dör. Detta kan både skapa prestandaproblem samt leda till ett oförutsägbart bete-

ende av applikationen. Något som bl.a. kan göra den mer svårnavigerad.

Det finns en del död kod, tomma klasser och metoder med hårdkodad testdata men detta beror mest troligt på att applikationen fortfarande är under utveckling. Testerna är inte heller heltäckande, utan har i dagsläget 13 % code coverage. Något att ha i åtanke är att det är svårt att skriva JUnit tester för Android-aktiviteter. Vilket märks då View-paketet har 0 % code coverage som därmed sänker den totala täckningen. Däremot är det bra att skriva tester under projektets gång och försöka åstadkomma *Test Driven Development* och inte skriva alla tester när kodningen är slutförd.

Referenser

- [1] J. Tidwell, *Designing interfaces*. Sebastopol, CA, USA, O'Reilly Media, Inc., 2011, ISBN: 978-1-449-37970-4.
- [2] Adobe. (2019). Adobe xd, [Online]. Tillgänglig: <https://www.adobe.com/se/products/xd.html> (hämtad 7 okt. 2019).
- [3] Google. (2019). Firebase, [Online]. Tillgänglig: <https://firebase.google.com/docs> (hämtad 23 okt. 2019).
- [4] ——, (2019). Firebase for android, [Online]. Tillgänglig: <https://firebase.google.com/docs/reference/android/packages> (hämtad 20 okt. 2019).
- [5] A. Leiva. (2019). Mvp for android: How to organize the presentation layer, [Online]. Tillgänglig: <https://antonioleiva.com/mvp-android/> (hämtad 2 okt. 2019).
- [6] Mockito. (2019). Tasty mocking framework for unit tests in java, [Online]. Tillgänglig: <https://site.mockito.org/> (hämtad 24 okt. 2019).
- [7] A. Zagretdinov. (2019). Mockito powermock, [Online]. Tillgänglig: <https://github.com/powermock/powermock/wiki/mockito> (hämtad 24 okt. 2019).
- [8] STAN. (2019). Stan, [Online]. Tillgänglig: <http://stan4j.com/> (hämtad 23 okt. 2019).
- [9] A. G, Liskov, 2018. [Online]. Tillgänglig: http://www.cse.chalmers.se/edu/year/2018/course/TDA552_Objekt_Orienterad_Programmering_Design/files/lectures/tda552_lecture2-2.pdf.
- [10] Android. (2019). Conditional navigation, [Online]. Tillgänglig: <https://developer.android.com/guide/navigation/navigation-conditional> (hämtad 23 okt. 2019).
- [11] J. Tidwell, *Designing interfaces*. Sebastopol, CA, USA, O'Reilly Media, Inc., 2011, s. 12–13, ISBN: 978-1-449-37970-4.
- [12] Android. (2019). Instance state, [Online]. Tillgänglig: <https://developer.android.com/guide/components/activities/activity-lifecycle#instance-state> (hämtad 24 okt. 2019).