Chalmers Tekniska Högskola

Objektorienterat programmeringsprojekt
TDA367

SDD-Dokument

Skriven av:
Milos Bastajic
Alexander Huang
Joel Jönsson
Johan Gottlander
Adam Jawad

Grupp 12 - Majja Dev Team
25 Oktober 2019

1 Introduktion

I detta dokument kommer applikationens struktur och design beskrivas. Applikationen som har namnet “bYMe” är en plattform där användare kan registrera ett konto för att sedan kunna lägga upp samt söka bland olika tjänster i form av annonser. Därav kan man skicka förfrågan till annonser för den erbjudna tjänsten. Mottagaren kan acceptera/avböja förfrågan. Om en tjänst accepteras och tiden passeras kan den som efterfrågade tjänsten betygsätta den som erbjöd tjänsten.

Applikation är byggd för att underlätta för privatpersoner och företagare att hitta sina kunder via ett enkelt och snyggt användargränssnitt.

Applikationens implementation utgår från att följa konventioner för en modulär design. För att uppnå detta används olika designmönster som exempelvis MVC[1] och Observer[2] pattern.

1.1 Definitioner, akronymer och förkortningar

MVC[1] - Model View Controller. Designmönster inom objektorienterad programmering som skiljer på modellen, representationen och kontrollen/manipulerbarhet.

Dependency Injection[3] - Designmönster som ökar abstraktionsnivån i implementationen genom att tillföra ett interface mellan en klient och konkret klass. Därmed “inverterar man beroendet”. Exempelvis behöver modellen bestämma när en modul utanför ska spara data. Istället för att modellen beror på sparmodulen kan modellen bero på ett internt interface som sparmodulen implementerar.

Observer[2] - Designmönster där en modul lyssnar på en annan via ett interface och en update-metod. Exempelvis notifieras moduler som beror på modellen när något i modellen ändras.

“Vy” - Den modulen/delen i programmet som ansvarar för användargränssnittet.

“Kontroll” - Den modulen/delen i programmet som ansvarar för kontroll/manipulerbarhet.

“Aggregate object” - Ett huvudobjekt som innehåller andra typer av objekt. Exempelvis hade klassen flygplan innehållit turbiner, vingar och cockpit.

“Parse-metod” - En metod som tolkar någonting. Exempelvis ett textformat.

“ShutdownHook” - En metod som körs precis när programmet stängs ner.

2 Systemarkitektur

Applikationens användargränssnitt är baserat på ett ramverk kallat JavaFX[4]. Det första applikationen laddar in är FXML-filer. FXML[5] är ett XML-baserat språk som är till för att bygga grafiska användargränssnitt i java. Därför är applikationens kontrollers och vyer är beroende av JavaFX-biblioteket.

Programmet laddar in användare och annonser från json[6]-filer vid start. Genom att läsa filerna skapas nya objekt i programmet som motsvarar datatypen som läses in. Detta åstadkommer programmet med “parse”-metoder som tolkar datan som läses in. Detta ansvarar klassen “AdHandler”, “AccountHandler” och “RequestHandler” för (De hittas i paketet “Services”). Objekten som skapas föreställer en del av modellen. Därav lyssnar vyn på modellen och skapar sina egna objekt för att kunna representera samtliga objekt som har lästs från minnet. Dessa vy-objekt kallas för “AdItems” och placeras i ett flöde i mitten av gui:t.

Hela proceduren vid uppstart appliceras främst på annonser. Annonser är den röda tråden i programmet som allting revolverar runt, och därför är det viktigt att läsning och skrivning av varje instans måste ske på ett konsistent och smidigt sätt. Därutöver är konton också en viktig komponent att spara. Proceduren vid uppstart för konton är nästan identisk. Skillnaden är att det inte finns ett motsvarande vy-objekt för varje konto. Istället beror vyn på det konto som anses vara inloggat dvs en “currentUser” från modellen. När ett konto är inloggat anpassas vyerna vilket ger användaren tillgång till mer funktionalitet.

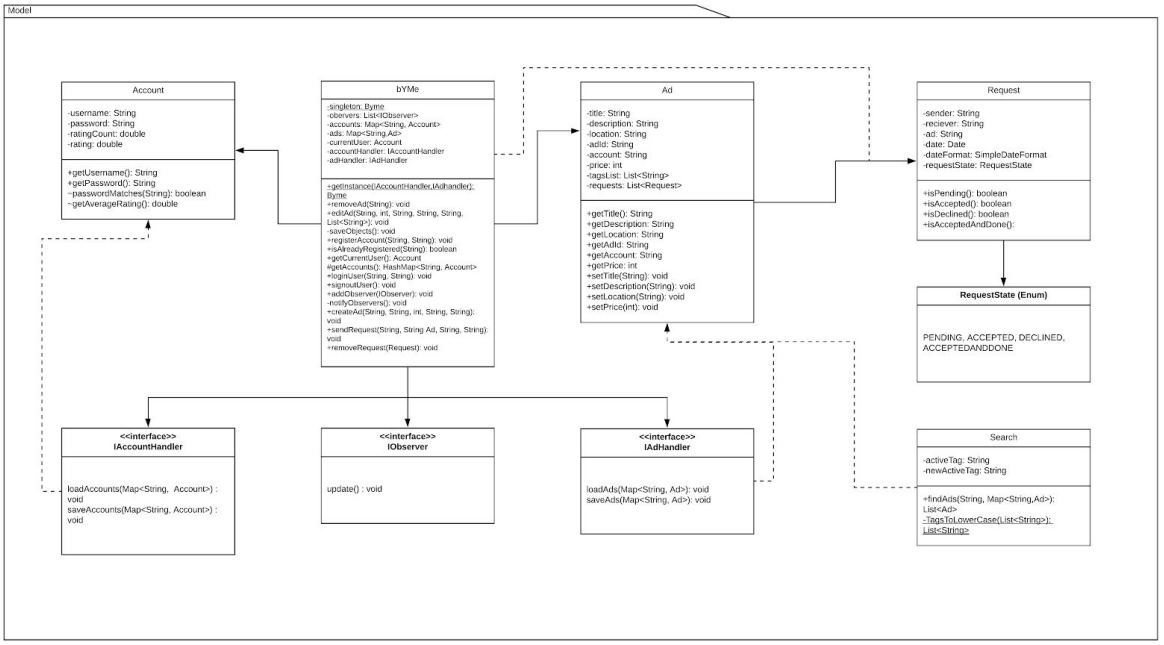
Programmet visar alltid en framsida med annonser utan att någon användare är inloggad. När programmet körs kan nya konton/annonser skapas. Dessutom kan annonser redigeras och man kan lägga till annonsbilder om man är inloggad och äger annonsen. Varje konto har en profilbild som kan ändras. Varje annons kan ha upp till 5 bilder som kan ändras i redigeringsläget. Vyn kan också ändras då det finns en inställning för mörkt tema för de som har det som preferens.

Ifall en användare stänger ner programmet när användaren är inloggad, loggas personen ut från sitt konto till nästa gång applikationen öppnas. Vid avstängning sparas all data i lokala filer för att användaren ska kunna fortsätta nästa session där förra sessionen avslutades. När programmet stängs av körs en “shutdownHook” som kallar på metoder som sparar objektens tillstånd. Nästa gång programmet körs skapas alltså nya instanser fast med samma tillstånd som de förra hade.

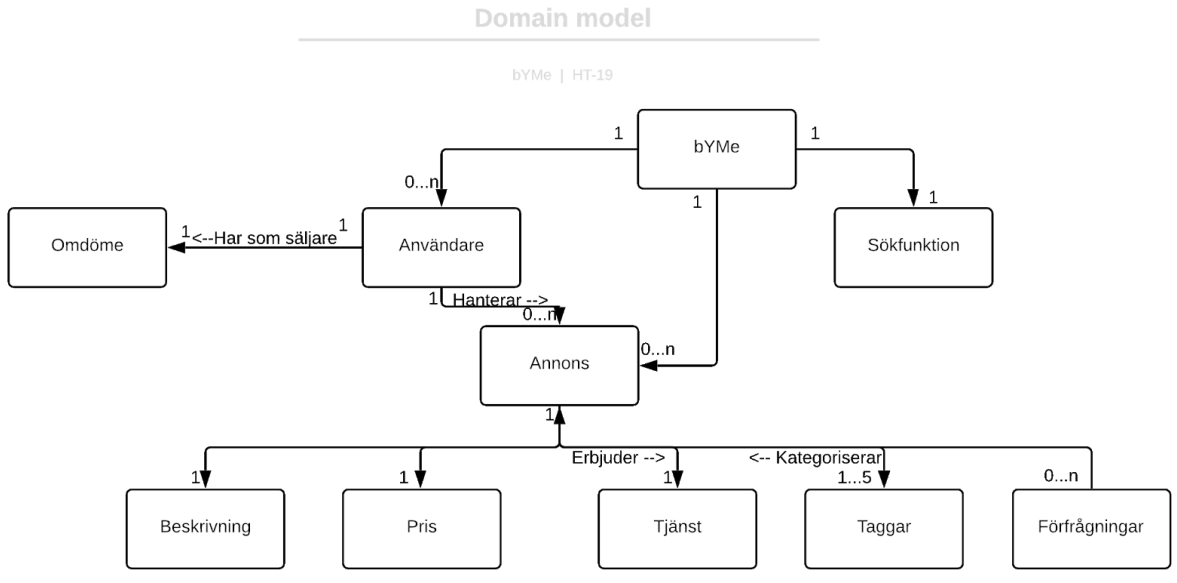
Enkel förklaring av Användarprocess:

1. Användaren startar programmet -> FXML-filer laddas in och “loadAds()”, “loadAccounts()” och övriga läs-metoder i paketet “Services” körs för att återskapa objekt från förra sessionen.
2. Användaren skapar ett konto och loggas in automatiskt -> Kontrollers som hanterar skapning av konton påverkar modellen vilket leder till att modellen kallar på notifyObservers(), och därmed uppdateras vyn.
3. Användaren skapar en annons med titel, pris osv. -> Kontroller som hanterar skapning av ads påverkar modellen vilket leder till att vyn uppdateras.
4. Användaren trycker på sin egna annons och trycker på redigeringsläge och ändrar titel och bild och trycker sedan “spara” -> Kontroller som hanterar redigeringsläge sparar annonser och kallar på editAd() i modellen, därmed uppdateras vyn.
5. Användaren trycker på en annons som någon annan har skapar och trycker på “skicka en förfrågan”. Därefter fyller användaren in datum och tid för när tjänsten efterfrågas. -> Request-objekt skapas i modellen och hanteras för avsedd annons. Mottagaren får en request i sin inbox med requests i gui:t.
6. Mottagaren trycker på “acceptera förfrågan”. -> Request-objektets state ändras i modellen och vyn uppdateras för både mottagare och sändare.
7. Efter att förfrågan accepteras och tiden för avsedd tjänst har gått ut, kan användaren som skickat förfrågan betygsätta den som erbjöd tjänsten (med 1 till 5 stjärnor). -> I modellen ändras betyget för den som erbjöd tjänsten och därav uppdateras modellen.
8. Användaren loggar ut -> Modellen ändras och vyn för en användare som ej är inloggad anpassas.
9. Användaren stänger av programmet -> “saveAccounts()”, “saveAds()” och övriga skriv-metoder körs i paketet “Services”. Alla objektens tillstånd sparas i json-filer.

3 Systemdesign



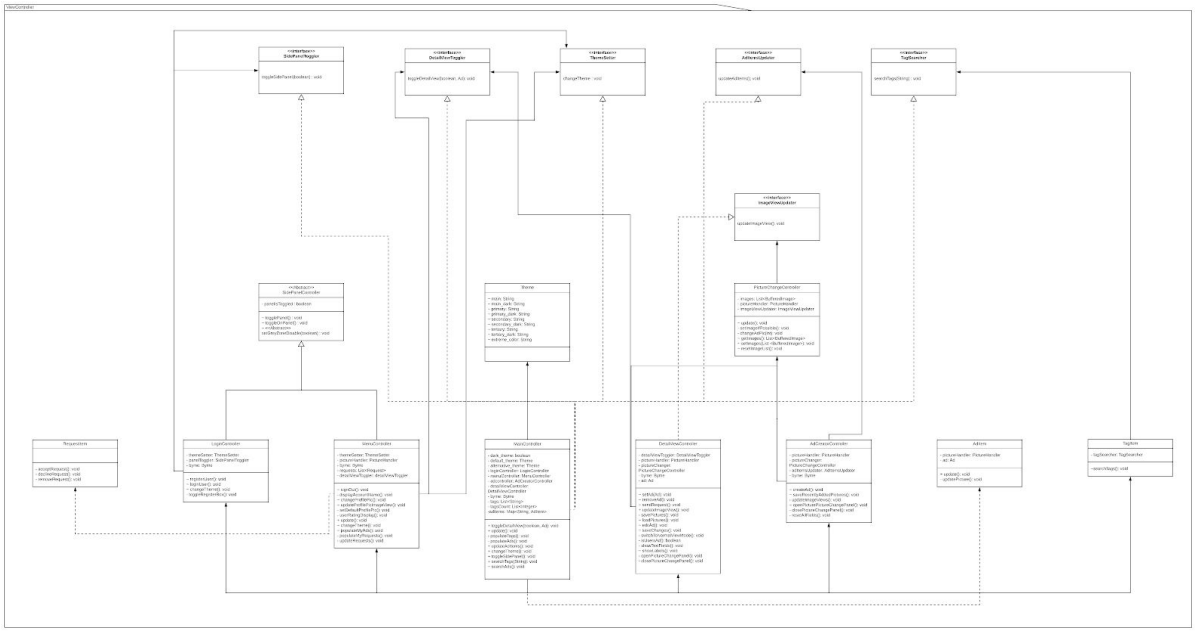
Designmodellen till applikationen.
Länk:
https://www.lucidchart.com/invitations/accept/39172a91-357f-4fc2-9857-eeb9dc94776e



Domänmodellen till applikationen.
Länk:
https://www.lucidchart.com/invitations/accept/6a6466e5-dddf-413f-b2d4-0920d7bd2ad0

Det finns en klar och tydlig relation mellan domänmodellen och designmodellen. Man kan se en tydlig relation mellan bägge om man kollar på uml-klassen “Byme”. Den har en Map med

användare(Account) och en Map med annonser (Ad) i designmodellen samtidigt som domänmodellen visar att den har “0...n” användare och “0...n” annonser. Även “Request” i domänmodellen är i enlighet med domänmodellens “Förfrågningar”. Sökfunktionen syns också i båda diagrammen men beskrivs på olika sätt. I implementationen är search en egen avgränsad klass som ingen i modellen delegerar. Medan i domänmodellen har applikationens huvudklass en sökfunktion. Den största skillnaden mellan domän- och designmodellen är att i domänmodellen syns pris, tjänst, taggar och beskrivning som egna “klasser” medan i implementationen är de endast instansvariabler till annonsen.

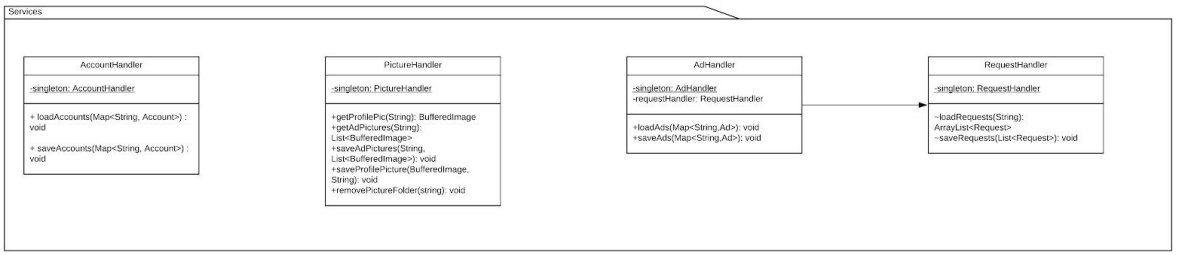


UML-diagram för ViewController. Länk:
https://www.lucidchart.com/invitations/accept/4d349c3b-555a-4e12-b029-2fb69d9be70b

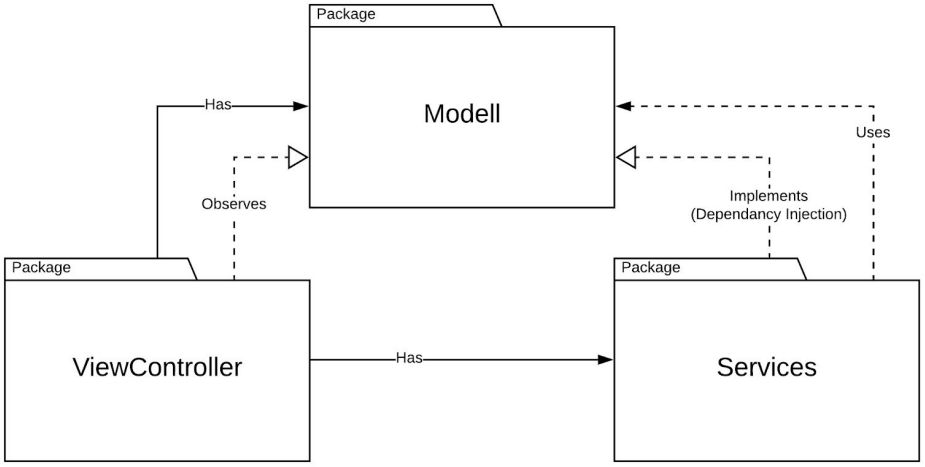
Applikationen använder sig av MVC[1]. Eftersom applikation använder sig av JavaFX[4], som integrerar vyn i kontrollern, har applikationen kontrollen och vyn ihop. Modellen är dock helt avskild från kontrollen och vyn eftersom den viktigaste regeln inom MVC[5] är att modellen inte får bero på vyn och kontrollen. Modellen ska vara helt åtskild och ska inte påverkas av hur representationen (d.v.s. vyn) ser ut. Däremot påverkas egentligen modellen indirekt av kontrollers eftersom att man som användare kan styra inloggning, konto-skapande, annons-uppläggning osv. Men modellen avgränsas från kontroller med hjälp av en modulär design. Kontroller har beroende till modellen, och eftersom att det största beroendet är till modellens “aggregate object” som är av typen Byme, kan kontroller komma åt den funktionalitet som behövs för att programmet ska kunna fungera.

För att vyn ska kunna uppdateras och visa aktuella tillstånd av objekt, måste vyn bero på modellen. För att ha en uppdateringsfunktion med en modulär design används observer-mönstret[2]. Med hjälp av “IObserve” som är ett interface för observers, kan alla vyer “lyssna” på modellen och så fort som något i modellen ändras eller skapas kan modellen eller den s.k. “observable” (i vårt fall “aggregate object”) meddela alla observers att något i

modellen har uppdaterats. När något i modellen uppdateras uppdateras alltså vyn på samma gång.



Vid uppstart och avstängning sker läsning/skrivning av data för att kunna spara och återskapa föregående session. Detta görs med hjälp av klasserna “AdHandler” och “AccountHandler” som sparar annonser och konton. De har beroende till modellen. Men modellen måste kunna utlösa läsning/skrivning utan att bero på den komponenten som läser/skriver data. Därför används “dependency injection”-mönstret[3]. Det som utgör mönstret är interface:n “IAccountHandler” och “IAdHandler” som finns i modellen och som endast exponerar läs-/skriv-metoderna. Klasserna “AccountHandler” och “AdHandler” implementerar interface:n och därav kan modellen dra nytta av de metoder som behövs i modellen utan att bero på själva klasserna som finns utanför.



UML-diagram över paketen och beroenden mellan dem.

Som tidigare nämnt är modellen ej beroende av något annat paket som inte är javas standardbibliotek. ViewController (Vyn och kontrollen) lyssnar på modellen via Observer-mönster och Services används indirekt av modellen genom “dependency injection”-mönstret. Både ViewController och Services har beroende till komponenter i modellen.

I implementationen syns det att det finns en del singleton-klasser. Detta mönster används i klasserna: Byme, AdHandler, AccountHandler, PictureHandler, RequestHandler och ErrorMessageController. De klasser med singleton är de klasser som vi endast vill ha 1 objekt av. Med hjälp av detta kan vi försäkra oss om att det ej skapas fler objekt och därav ser man till att det är samma objekt som används i hela programmets flöde.

4 Persistent datahantering

För att spara annonser och konton används JSON[6]. JSON(JavaScript Object Notation) är ett kompakt textbaserat format som används för att utbyta data. I en annons sparas ett annonsid, pris, beskrivning, taggar och område. Utöver det sparas användarens namn för att kunna identifiera vilken annons som tillhör vilken person. I ett konto sparas användarnamn, lösenord, omdöme och hur många omdömen användaren har fått. En förfrågan innehåller avsändare, mottagare, meddelande, datum, status (t.ex. accepterad eller avböjd) och ett annonsID för att kunna identifiera den korrekta annonsen. I programmet är klasserna “AccountHandler”, “AdHandler” och RequestHandler ansvariga för att läsa/skriva data i filer av typen .json. Klasserna skriver objekt på formen:

Konto:


```
[{"password":"123","rating":0.0,"ratingCount":0.0,"username":"alex
98huang"}]
```


Annons:


```
[{"adId":"3454","price":120,"tagsList":["klipp","","","","",""],"desc
ription":"Klipper alla!
Billigt!","location":"Skåne","title":"Bästa frisören i
malmö","account":"alex98huang"}
```


Förfrågan:


```
[{"date":"2019-11-02\/12:30","ad":"3454","receiver":"alex98huang",
"sender":"joel","state":"PENDING","message":"Hej! Jag är
intresserad av din tjänst!"}]
```


På detta sättet beror inläsningen av objekt varken på skiljetecken eller nya rader som skiljer objekt åt. Vid inläsning används en metod som tolkar formatet och skapar objekt som motsvarar varje “JSONObject” som i förhand läses från filer.

För att spara bilder används en “PictureHandler” som är en klass i paketet “Services”. Det finns ett par standard mapp-sökvägar i klassen som den sparar bilder i. En för alla profilbilder och en för alla annonsbilder. För att identifiera och hitta bilder som tillhör motsvarande profilbild används användarnamnet. Tillhörande bild får samma namn som kontot + .jpg. Exempelvis lägger användaren: Alex98Huang in en profilbild. Bilden sparas i en dedikerad mapp /profile_pics/Alex98Huang.jpg. Om man sparar annonsbilder skapas en ny mapp som har samma namn som annons-id:t. Man kan därmed ha flera bilder som är kopplade till samma annons. Exempelvis läggs 2 bilder in till annonsen med id:t 1234. De får sökvägarna: /ad_pictures/1234/1.jpg och /ad_pictures/1234/2.jpg. Man kan ha max 5 bilder per annons. Mappen /ad_pictures/1234 raderas om annonsen med id:t 1234 raderas.

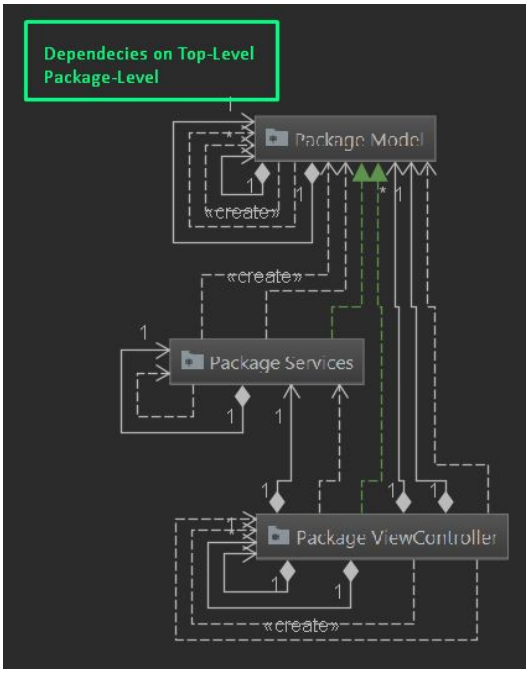
5 Kvalitet

I projektet finns en separat testmapp där tester görs per klass. Sökvägen dit är: ooprojekt\bYMe\Test. Testerna har utförts på metoderna i “Model”-mappen med hjälp av jUnit[7]. Testerna testar relativt enkel men kritisk funktionalitet som exempelvis ifall ett konto skapats, vilket konto som är inloggat, ifall det går att radera och skapa annonser m.m. Ytterligare utförs verkliga användartester själva, där vi använder programmet och försöker testa saker som skulle kunna ge oväntade resultat.

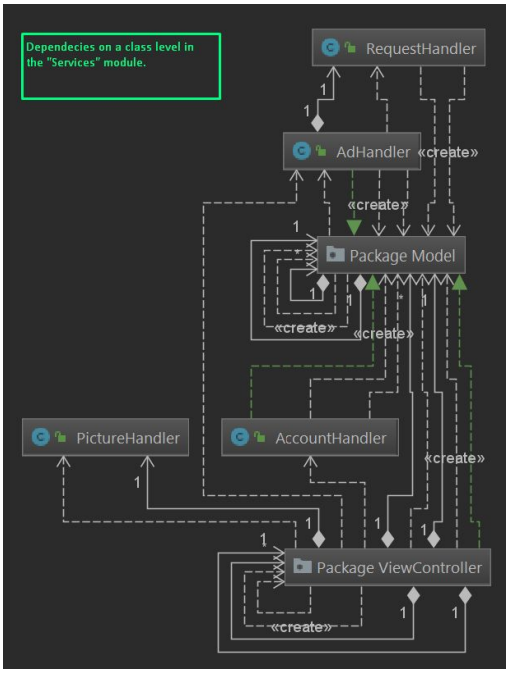
PMD[8] (med ruleset quickstart[9]) har använts för att hitta eventuella brister i koden. Bristerna som hittades bestod främst av ArrayList- och HashMap-variabler som skulle göra om till List och Map, Locale användes inte vid toLowerCase, @Override saknades vid vissa metoder och att { } saknades vid några if-satser. De flesta brister har lösts, dock har vi låtit några brister vara kvar som till exempel onödiga parenteser, stor bokstav i paketnamn och namn med understreck.

En brist vi har i koden är att vi inte använder oss av “immutable objects”. Detta gör att man kan komma åt objekten vilket möjliggör ändring av dessa även om det är objects som inte ska kunna gå att ändra. En annan brist är att vi har singleton på vårt aggregate object (Byme), eftersom den är global och allting kan komma åt den. Vidare existerar ännu ett problem vilket är att bYMe i dagsläget kan hantera upp till högst 10 000 annonser, detta eftersom deras unika ID:n består av 4 siffror som totalt har 10 000 olika kombinationer.

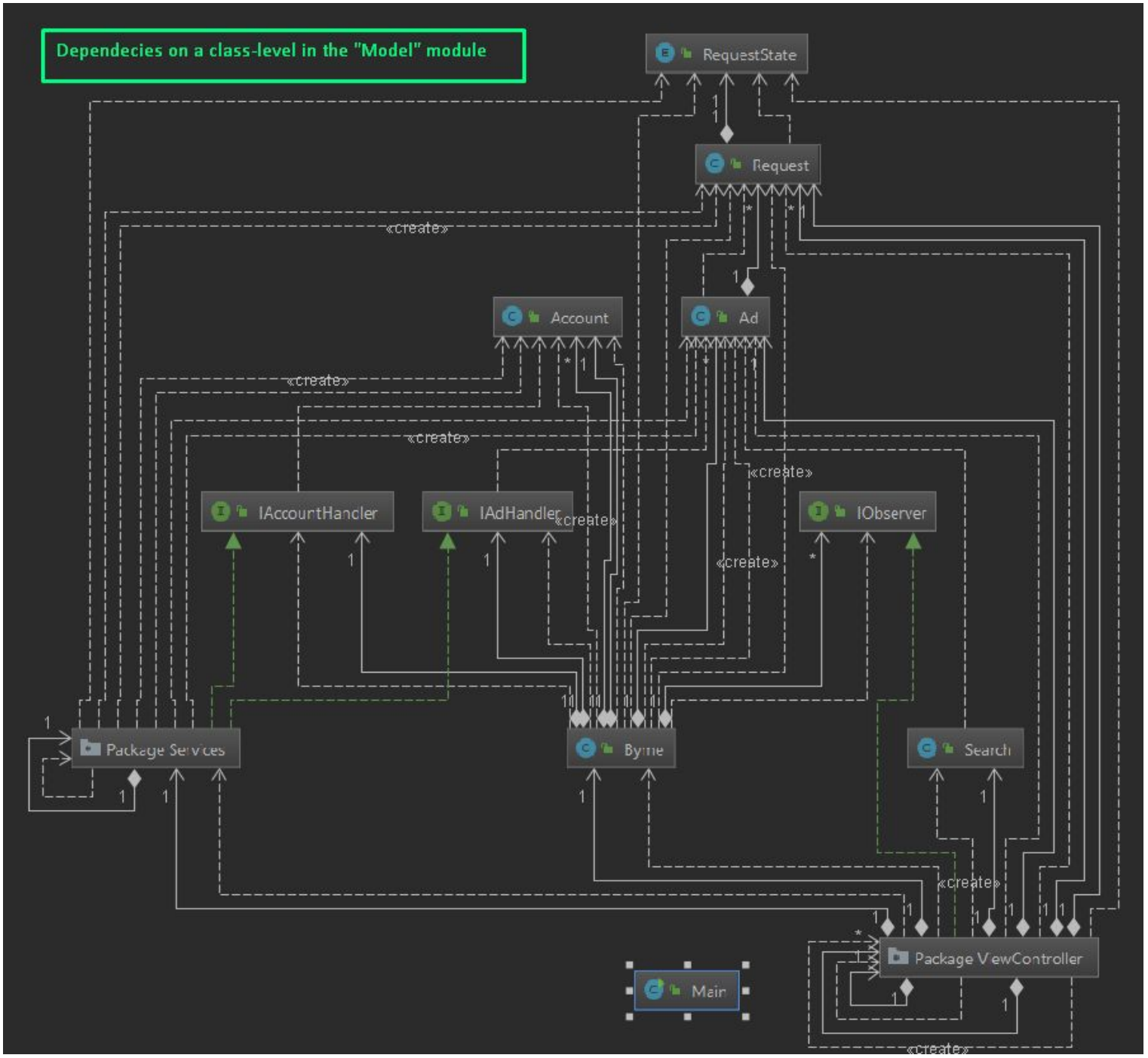
Med hjälp av Intellj-Ultimates “Dependency Analysis” har vi granskat beroendena på både en modul- och klassnivå för att säkerställa att den matchar våran Design Model. Dessutom använder vi även hjälpmedlet för att säkerställa att det inte finns några dubbelberoenden mellan klasser eller annat som strider mot principerna i OO-programmering.



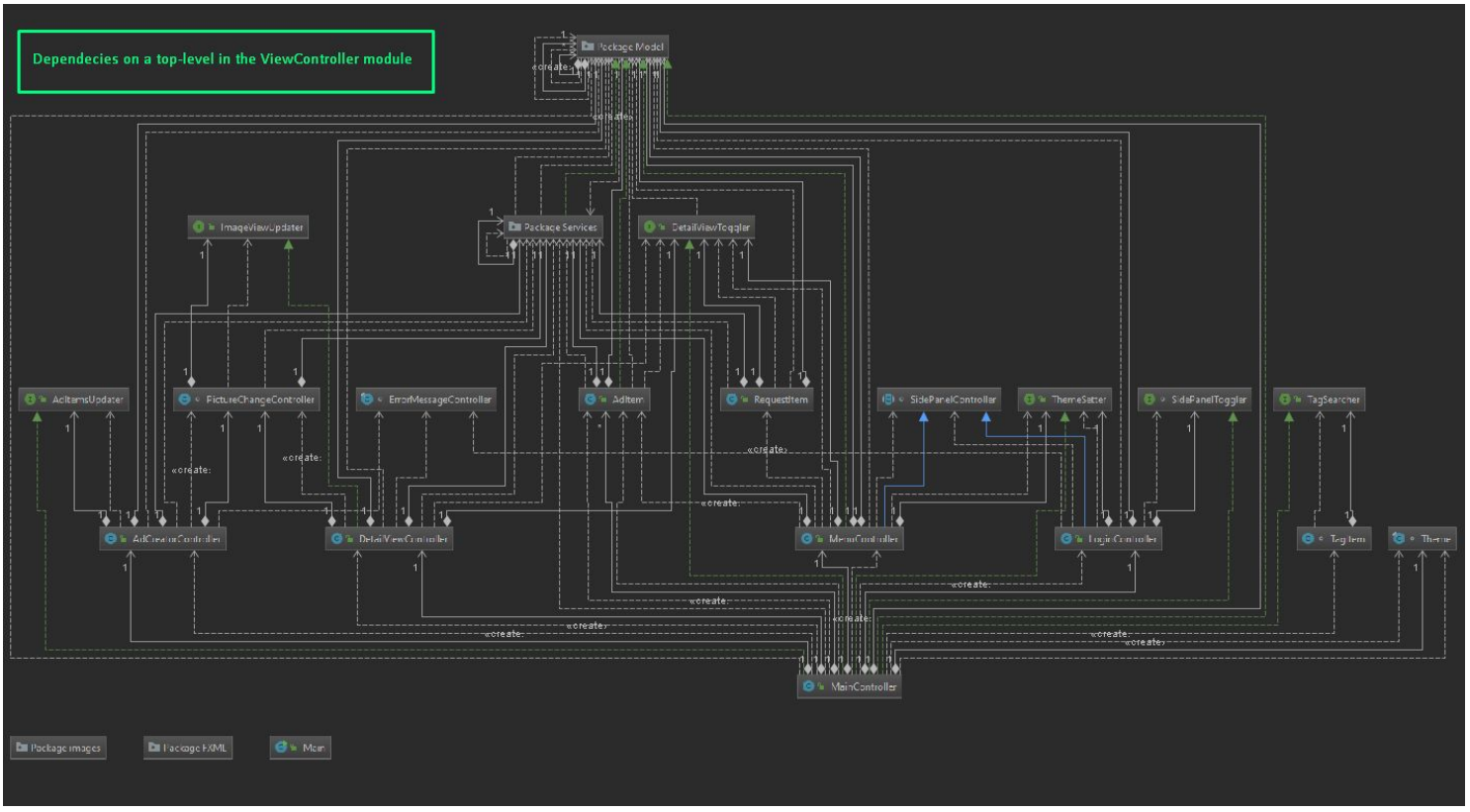
Beroenden mellan de olika paketen.



Klassberoenden i “Services”-paketet.



Beroenden på klassnivå i “Model” paketet.



Beroenden på klassnivå i “ViewController” paketet.

5.1 Åtkomstkontroll och säkerhet

För att kunna skapa annonser krävs det att man är inloggad. Är användaren inte inloggad kan inte “Skapa annons”-vyn nås. Alla kan kolla på annonser och få upp en mer detaljerad vy av annonser. I detaljvyn av annonser får användaren som la upp annonsen två extra funktioner, användaren kan redigera annonsen och ta bort den ifall användaren vill. Ytterligare restriktioner för användare som inte är inloggade är att de inte kan skicka några förfrågan på annonser. Vidare ändras fönstret som dyker upp när man som användare trycker på “profil”-knappen

6 Referenser

1 MVC: https://sv.wikipedia.org/wiki/Model-View-Controller
2 Observer-mönster: https://sv.wikipedia.org/wiki/Observatör_(designmönster)
3 Dependency Injection: https://en.wikipedia.org/wiki/Dependency_injection
4 JavaFX 8 bibliotek (API): https://docs.oracle.com/javase/8/javafx/api/
5 FXML: https://en.wikipedia.org/wiki/FXML
6 json-simple (API): https://code.google.com/archive/p/json-simple/
7 jUnit (API): https://junit.org/junit4/javadoc/latest/index.html
8 PMD: https://pmd.github.io
9 Ruleset quickstart: https://github.com/pmd/pmd/blob/master/pmd-java/src/main/resources/rulesets/java/quickstart.xml