FANATIC

Resumé

- Nicolai Johansen & Andreas Bonke

Denne rapport omhandler udarbejdelsen af en app, som hjælper rollespils butikken Fanatic til mersalg og et overblik over, hvad deres kunder ønsker af varer i butikken. Formålet med rapporten er at vi erfarer og demonstrerer, at vi kan udvikle software ved brug af Unified Process.

Vi vil analysere Fanatic og se hvilke behov vi kan afdække for dem. Programmet skal designes i Blend fra Visual Studios, og udvikles, modelleres og implementeres i Visual Studios ved hjælp af programmeringssproget C#. Til sidst vil vi teste appen for at sikre os, at vi har opnået det ønskede resultat.

Indholdsfortegnelse

Indledning

- Fælles

I Roskilde og København findes rollespilsbutikken Fanatic. Fanatic handler med rollespilsudstyr, kortspil, brætspil, figurspil, og meget mere. Daniel, som er ejer af Fanatic, har tilknyttet en kundeklub til Fanatic, hvor der er mulighed for at få bedre tilbud, og point som man kan bruge i butikken og på webshoppen.
Fanatic ønsker at kunne trække statistikker ud på hvad deres kunder er mest fokuseret på i butikken, så der kan gives bedre tilbud til hver individuel kunde igennem kundeklubben.
Appen baner mulighed for bedre events til størst mulige målgruppe, et godt overblik over kunder, og øget salg.
Rapporten indeholder inception fasen, og to iterationer i elaboration fasen.

Udarbejdet af Andreas Bonke, Daniel Hansen, Søren Kongsgaard, Nicolai Johansen og Nikolaj Nielsen.

Virksomhedsbeskrivelse

- Fælles

Fanatic er en rollespilsbutik som sælger rollespilsvåben, brætspil, kortspil og figurspil. Virksomheden har beliggenhed i Roskilde og København, og har fire ansatte. Fanatic har for nylig lanceret et kundeloyalitetsprogram. Dette loyalitetsprogram går ud på, at kunderne belønnes med 10% buyback, i form af points de kan bruge til et senere køb.

Vi har kontakt til Daniel, som er ejer af butikkerne.

Inceptionfasen

Projekt Etablering

Projektformulering:
"How can Unified Process (UP), C#-Programming and Relational Databases be used for development and implementation of a minor IT system?"

Business Model Generation Canvas

- Fælles

Costumer segment:

Fanatics kundegruppe er relativt lille, da deres produkter henvender sig til et lille kundesegment og derfor befinder de sig på et nichemarked.

Costumer relationships:

Fanatics kunderelation er baseret på personlige relationer. For at gøre den relationen større har ejeren startet et kundeloyalitetsprogram, hvor kunderne får en bruger og 10% "buyback" på produkter de køber, i form af point som der kan handles for i butikken eller hjemmesiden. Her vil vores program give ejeren et større overblik over, hvad kunderne kommer for i butikken. Ud fra disse data kan ejeren
lave statistik og derved lave arrangementer eller tilbud der henvender sig til bestemte kunder i hans loyalitetsprogram.

Channels:

- Awareness: Fanatic skaber dette igennem sin hjemmeside og igennem personlig tilstedeværelse i butikken samt nyhedsbrev.
- Purchase: Fanatic sælger deres produkter igennem deres to fysiske butikker og igennem deres internetshop "Fanatic".
- Delivery: Kunderne modtager deres produkt igennem post, hvis de har bestilt deres produkt igennem Fanatics internetshop.
- After sales: Igennem Fanatics webshop kan kunderne give feedback på produktet og selvfølgelig henvende sig til ejeren af butikken.

Revenue streams:

Fanatic tjener penge på de fysiske varer, som de sælger igennem deres butik. Derudover har de også en webshop, hvor man kan bestille deres produkter. Fanatic arrangerer desuden mange gange om måneden events og turneringer, hvor interesserede kan betale et beløb for at deltage.

Key partners:

Fanatic har partnerskab med Enigma og Arcanetinmen, som er de primære leverandører i Danmark af diverse kortspil, Figurspil og brætspil, og som er de eneste der kan få varer hjem i ordentlig tid fra udlandet. Fanatic

samarbejder også med diverse små leverandører som alle er leverandører af varer som bliver forhandlet i butikken.

Key activities:

Fanatics primære aktiviteter er salg af varer i butik og at arrangere events for sit netværk af kunder.

Value propositions:

Fanatic giver kunderne accessibility til produkter man normalt ikke kan købe i legetøjsbutikker og butikker såsom "Bog & Idé" i form af brætspil. Fanatic sælger også deres viden i form af rådgiving og hjælp til hvilke produkter man kunne være interesserede i og som passer til ens interesser.

Key resources:

Fanatics primære 'key resource' er deres fysiske varer i butikken, som genererer størstedelen af deres omsætning. Fanatic stiller også deres Lokaler til rådighed til spilleaftener, hvor folk kan komme og bruge butikkens Terrain og borde til at holde miljøet kørende Derudover behøver Fanatics personale en viden om rollespil, brætspil, "Magic card" og erfaring med værtskab for arrangementer. Arrangementer bliver afholdt i butikken, som også er den sidste af deres key resources.

Cost structures:

For at have varer på hylderne i butikken kommer der naturligvis omkostninger ved bestilling af nye produkter. Fanatic har nogle faste udgifter hver måned som dækker over personale omkostninger i form af løn. For at benytte sig af deres lokaler følger der under faste omkostninger også husleje med. Den mest varierende omkostning er butikkens ugentlige events, hvor der går penge til at holde butikken længere åben samt personale.

Interessenter

- Af Daniel

- ❖ Fanatic - Er interesseret i dette, for at få bedre indblik i hvad kunderne gerne vil have.
- ❖ Kunderne – De vil kunne få bedre tilbud og målrettet produkter til dem selv rent individuelt.

Via it-systemet vil Fanatic få en bedre indblik i hvad kunderne gerne vil have, da de kan trække statistik fra de data de får af programmet. Virksomheden vil på den måde få en bedre revenue stream, da virksomheden kan lave arrangementer eller tilbud på de varer kunderne er mest interesseret i. Udover dette vil cost structures også blive formindskes, da Fanatic har et bedre indblik i hvad kunderne køber, på den måde vil de bruge færre

penge på unødvendige ressourcer.

Fanatic vil på denne måde også kunne forbedre sine aftaler med deres key partners.

Visionstatement

- Fælles

Problem statement

Problemet er:	Med nuværende system har ejer ikke overblik over kunders interesser og eventuelt manglende produkter, i forhold til kunders efterspørgsel.
Problemet påvirker:	Kunne udbyde et forkert sortiment i forhold til kundernes efterspørgsel.
Problemet kan forårsage:	Manglende indtjening og høj lagerværdi grundet manglende salg.
En løsning på problemet kan være:	At udvikle en applikation med en database over kunder, og deres interesser. På den måde er det nemmere for butikken at ramme så bred en gruppe af sine kunder som muligt med relevante tilbud.

Product position statement

Til:	Fanatic
Som:	Skal bruges til at skabe overblik over kundernes interesser.
Produktet:	App som kan håndtere gæster/kunder og give Fanatic overblik over kundernes indkøb og aktivitet butikken
I modsætning til:	Ejerens mavefornemmelse.
Vores produkt:	Skaber overblik over kundernes interesser. Og skaber en mere fokuseret målgruppe for Fanatic, så de kan målrette deres salg og dermed øge det.

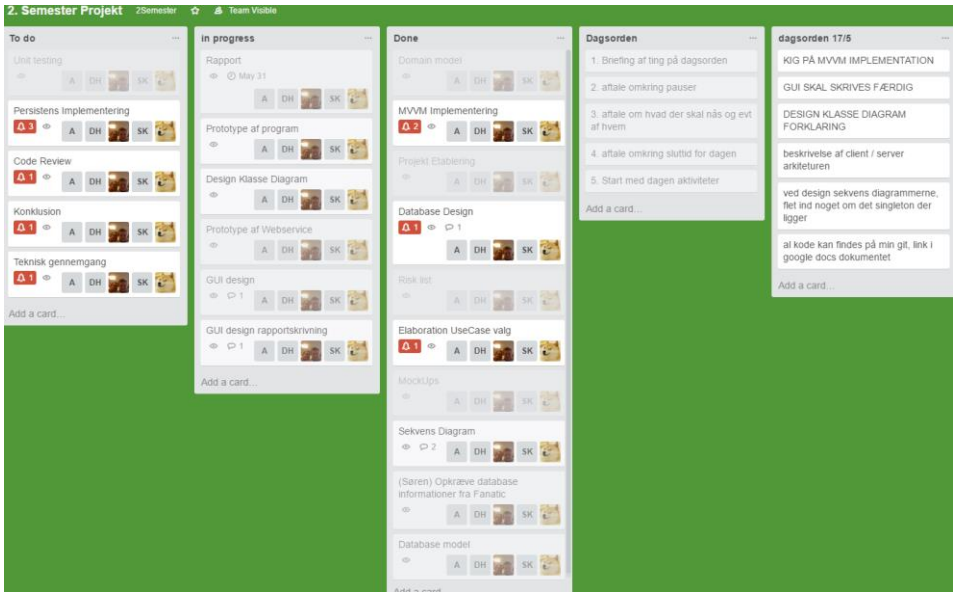
Styring af projektet

- Fælles

Vi har valgt at rapportskrivningen foregår igennem hele forløbet. På denne måde undgår vi at skulle skrive hele rapporten til sidst, samt vi forebygger eventuelt mangler i rapporten.

Vi som gruppe har aftalt, at projektet skal være færdiggjort den 21. Maj, så
vi har tid til at rette i vores projekt før deadline. Igennem projektet har vi struktureret vores arbejdsopgaver via programmet “Trello”. På denne måde kunne vi se hvor langt vi er nået, hvor meget vi mangler og hvad vi skal

følge op på. Dette hjælper os med at skabe overblik over projektet.



Softwarekrav

- Af Nicolai Johansen

- ❖ Microsoft Visual Studio 2015 til udvikling af programmet og tilhørende diagrammer
- ❖ Azure online SQL server til SQL
- ❖ Trello til scrum board og planlægning og overblik af projektets opgaver.
- ❖ Facebook til kommunikation mellem gruppens medlemmer
- ❖ Office 365 Online, Word, OneNote til noter og udarbejdelse af rapport og deling af dokumenter
- ❖ GitHub til versionsstyring
- ❖ Tredjeparts extension til Visual Studio - Udvikling af diagram til statistik af interesse formål

Risk List

- Fælles

Ved start af projektet overvejede vi hvilke udfordringer vi kunne løbe ind i, og udarbejdede en Risk list for at mindske risikoen for fejl. I risk listen har vi valgt Magnitude på en skala fra 1-10, hvor 10 er højest og 1 er mindst.

Type	*Beskrivelse*	*Magnitude*	*Impact*	*Indicator*	*Mitigation*
Technical	Får lavet for meget uden planlægning (waterfall)	7	High	Udfører arbejde ud fra løbende ideer	Gruppemøde for at planlægge forløbet.
Technical	Manglende erfaring med versionsstyring kan forårsage tab af data	7	Critical	Mistet arbejde	Planlægge brugen af Github og undervise hinanden.
Communication	Hvis vi laver det anderledes end det kunden havde visualiseret det.	6	medium	Kunde kan ikke nikke genkendende til det udførte arbejde	Bedre kommunikation med kunde evt. I form af mock ups.

Use case

- Nicolai Johansen

Til Fanatic har vi valgt at skrive fem use cases. Vi udarbejder disse, da det afdækker krav for systemet. Vores use cases indeholder en eller flere scenarier der viser hvordan systemet kan interagere med en bruger eller et eksternt system for at løse en specifik opgave. Use cases er skrevet så brugerne kan forstå det, da det er dem de er skrevet til.

Mål:

1. Kunden kan tilkendegive sit formål med sit besøg.
2. Kunden skal kunne oprette en bruger.
3. Kunden skal kunne logge ind.
4. Kunden skal kunne logge sig på som gæst (Hvis de ikke ønsker en bruger).
5. Fanatic skal kunne trække data ud af databasen.

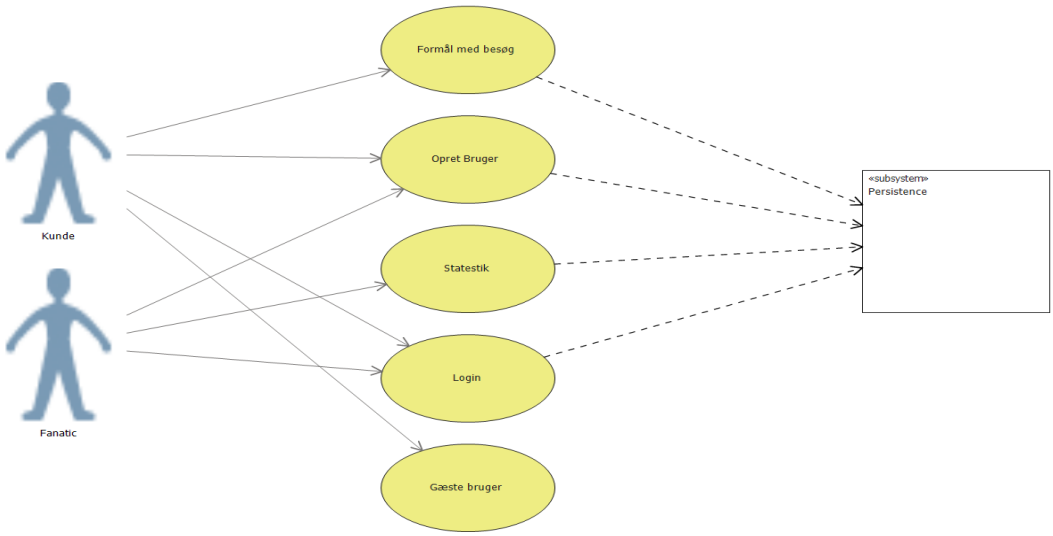
Use Case Diagram

- Af Daniel og Søren

Herunder er vores Use case diagram, der viser hvordan brugerne interagerer med de forskellige funktioner i programmet. I Use case diagrammet har kunden adgang til login, opret bruger, gæste bruger og formål med besøg. Kunden har adgang til disse funktioner, da det gør det muligt for kunden at tilkendegive sit formål med sit besøg. Derudover har kunden også mulighed for at vælge "Gæste Bruger", hvis kunden ikke ønsker en bruger på dette tidspunkt.

Fanatic kan oprette en bruger for de kunder, der ikke selv har gjort det igennem programmet samt logge ind for kunden. Derudover kan Fanatic også se statistik på formålene med kundernes besøg, hvilket giver Fanatic et indblik i hvorfor kunderne kommer i butikken.

Hver use case i use case diagrammet, på nær gæstebruger, interagerer med et eksternt system.



Fully dressed use cases

- Af Daniel

Ud fra de fem ovenstående use cases vi har udviklet, har vi valgt to use cases og skrevet dem som fully dressed. Vi mener at disse to use cases dækker kernefunktionerne af vores app, og er en væsentlig del, af vores program og på baggrund af dette har vi valgt at udbygge dem yderligere, på den måde er de mere detaljeret og specifikke.

Den første fully dressed use case tager udgangspunkt i at kunden "Tilkendegiver sit formål med sit besøg". Her

er succes scenariet at kunden eller gæsten har valgt, hvad formålet med sit besøg er. Denne use case vil blive brugt i den første iteration.
Den anden fully dressed use case tager udgangspunkt i at "Fanatic skal kunne trække data ud af databasen". Succes scenariet her, er at Fanatic skal kunne trække data ud fra databasen og lave statistik ud fra dette. Denne use case vil så blive brugt til den anden iteration.

1. First use case

- Af Nicolai Johansen

***Use case name**:* Tilkendegive sit formål med sit besøg

***Scope**:* Fanatic Statistik App

Level: User Goal

***Primary Actor**:* Bruger

Stakeholders and Interests:

- Kunden: Kunne forbedre sin oplevelse til næste besøg, og få mere målrettede tilbud og events fra butikken som er af kundens interesser.
- Fanatic: Interesseret i at få større indblik i hvad kunden er interesseret i, så de kan lave mere målrettede events og give mere målrettede tilbud for mere salg og bedre kunde omtale.

Success Guarantee:

- Kunden har som gæst eller kundeklub medlem valgt hvad formålet med sit besøg er og det er blevet sendt og gemt i Fanatics system.

Precondition

- Kunde skal være logget ind.

Main Success Scenario:

1. Kunden skriver/vælger deres formål med besøget og logger ud.
2. Data'en bliver sendt til databasen og gemt.

Extensions:

1.A Fanatic: Internetforbindelsen kan være ustabil og derfor virker login ikke og data'en kan ikke sendes eller blive gemt.
Hvis dette er tilfældet håndterer vi det med en besked til brugeren i form af en pop up boks med teksten: "*Noget gik galt! Kontakt medarbejder i butikken!*"

Special Requirements:

Kunden skal logge ind, tilkendegive sit formål og informationen skal sendes, og gemmes. Herefter skal appen automatisk logge ud. Hele processen skal tage højst 30 sekunder, minimum 9/10 gange.

Technology and Data Variations List:

- Kundeklub medlemmer
- GæsteID, så systemet kan kende forskel på gæster og medlemmer.

Frequency of Occurrence:

- 1-25 logins, båder medlemmer og gæster, om dagen.

Miscellaneous:

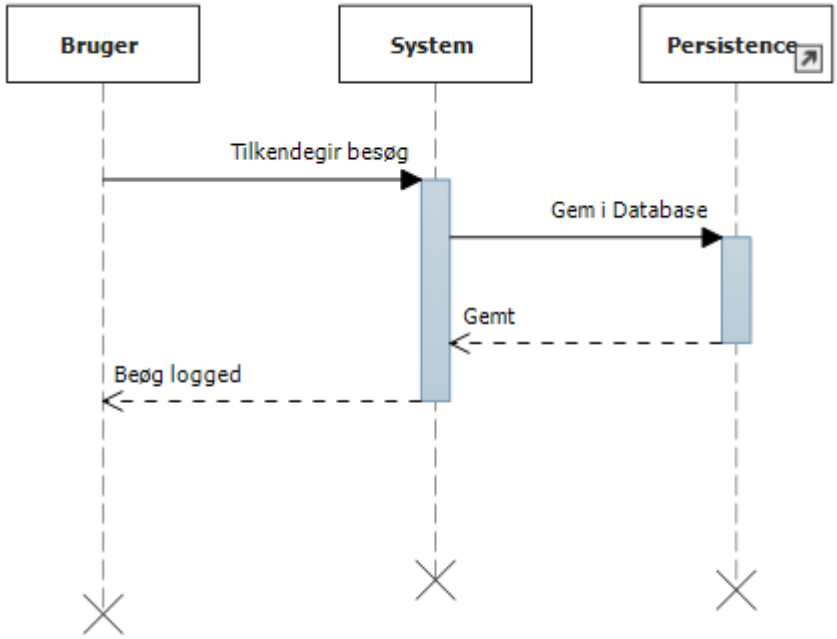
Det skal skrives i C#.

System Sekvens Diagram

- Af Nikolaj Bang.

System sekvens diagrammet giver her et overblik over, hvordan interne beskeder løber rundt i systemet, ud fra de to fully dressed use cases vi arbejder med. Systemet modtager bruger input fra brugeren, behandler disse beskeder og sender det til persistence, som agerer som kommunikationen mellem vores app og en webserver. Den beskriver success scenariet for en bruger der logger ind, og for en bruger der tilkendegiver sit formål med sit besøg.

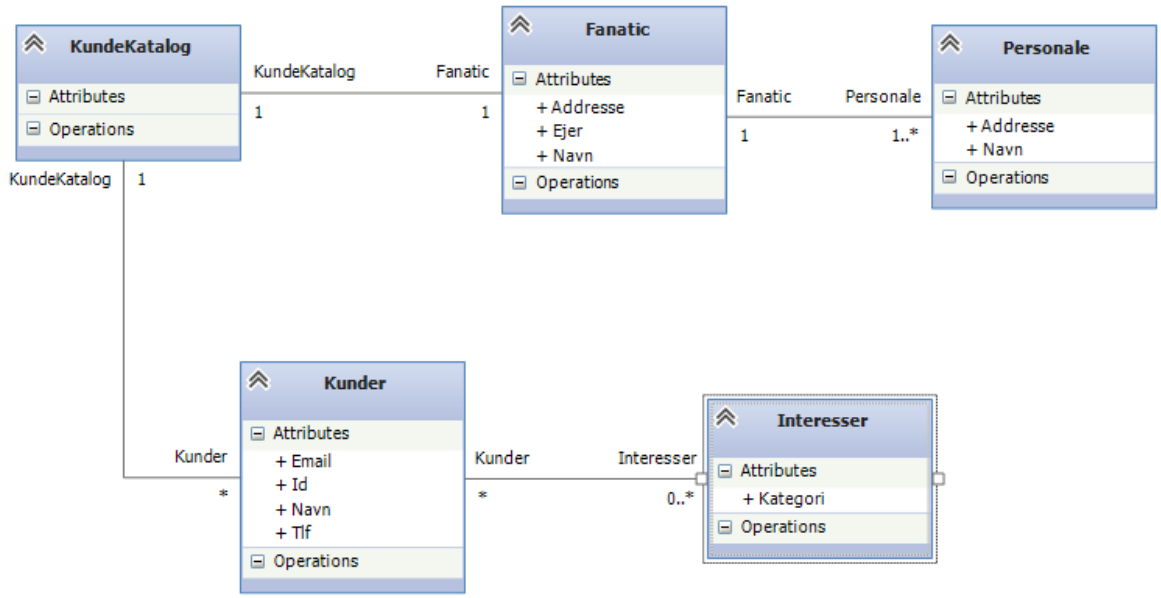
I sekvensen hvor brugeren tilkendegiver sit besøg, vælger brugeren hvilke interesser brugerens besøg involverer, hvorefter systemet behandler valget og gemmer dette i en database. Til sidst sendes der feedback til brugeren, som modtager et pop up vindue med en besked om at dataen er gemt korrekt, og et 'tak' fordi de tog sig tid til at bruge appen. Derudover gemmes de besøgendes tilkendegivelser fra appen, så de kan bruges af ejeren til at trække statistik ud på deres interesser i butikken.



Domain Model

- Af Daniel

Herunder ses vores Domain model, og den viser hvordan de forskellige konceptuelle klasser har associationer mellem hinanden. Associationerne beskriver hvordan forholdet er mellem de konceptuelle klasser. Klassen "Fanatic" indeholder attributterne adresse, ejer og navn denne klasse associerer med "Personale" og "KundeKatalog". Klassen "Personale" indeholder attributterne adresse og navn, denne klasse har ikke den store indflydelse på appen, men vi ved butikkens personale har adgang til appen. Klassen "KundeKatalog" (som er vores kundedatabase) associerer med klassen "Kunder" og denne klasse indeholder attributterne email, id, navn og telefon. Klassen "Kunder" som bliver til "User" i vores design klassediagram, associerer med klassen "Interesser", denne klasse indeholder attributten kategori, hvilket er de forskellige kategorier kunderne kan vælge imellem i appen.
Vi fandt frem til vores domain model via vores use cases og tog udgangspunkt i Fanatics business model.



Glossary

- Af Nikolaj Bang

- ❖ Fanatic: Butik i Roskilde.
- ❖ Fanatic Daniel: ejeren af Fanatic og kontaktperson.
- ❖ Brugeren: Fanatics kunder.
- ❖ Persistence: evt webserver eller database.

Bestemmelse af brugergrænseflade

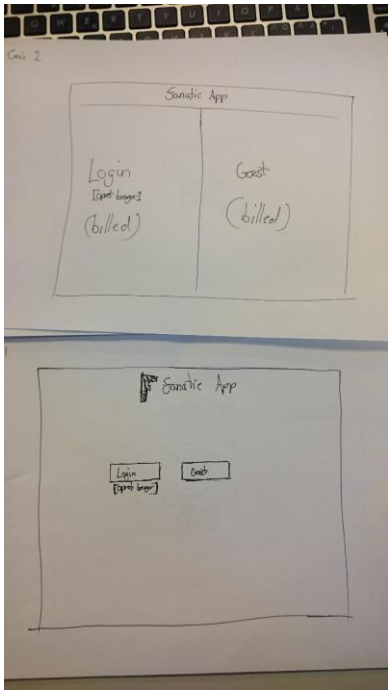
- Af Andreas og Søren

Gruppen har fra start af projektet haft en klar vision for hvordan designet skulle se ud. Vi lavede to eksempler på design, så kunden kunne se hvordan vi forestillede os en brugervenlig og indbydende brugerflade ville se ud.

I første eksempel har vi gjort brug af store knapper, så det er let at tilgå det man gerne vil. Der er valgt stor og klar tekst så det er let at navigere, og brugeren ikke skal tænke over hvad de betyder, og hvor de bliver ledt hen. Designet i sig selv forsøger altså at forebygge brugerfejl.

I vores anden prototype(Nederste billede) har vi samme ide omkring opsætningen. Designet er mere minimalistisk, og det ses blandt andet på størrelsen af knapperne. Det gør appen mindre touch venlig.

Efter fremvisning for Fanatic blev vi hurtigt enige om at det første design er det mest optimale til det formål som de også efterspørger.



Elaboration - 1. iteration

- Af Søren

I denne iteration tager vi udgangspunkt i den første use case, som er "Kunden kan tilkendegive sit formål med sit besøg". Vi vil undersøge om vores arkitektur fungerer og om vi kan holde det så simpelt så muligt. Brugen af appen må heller ikke tage for lang tid og man skal ikke bruge tid på, at finde ud af hvordan den virker.

Vi vil løbende gennem forløbet holde flere møder med Fanatic for at vi stadig holder os til det look og feel han gerne vil have i appen.

Prototype

- Af Søren og Andreas.

Efter aftale med fanatic, som valgte det første mock up[1] eksempel, har gruppen udarbejdet, en visuel experimental prototype som afspejler de exploratory mockup-prototyper som tidligere blev lavet for at give fanatic en ide om hvordan appen kunne se ud.

[1] Se afsnit om mockup for billede.

Gruppen har taget udgangspunkt i Fanatics eksisterende hjemmeside og har fået inspiration derfra. På følgende billeder har inspirationen været skrifttype, den simple hvide baggrund med sort skrift og billeder.

Efter udarbejdelse af den experimentale app prototype aftalte vi at mødes med Fanatic for at få bekræftet at designet var som de også havde forestillet sig.

På Fanatics hjemmeside highlighter de brugernes valg med farven blå. Vi valgte bevidst at gå imod deres design i dette tilfælde og bruge farven grøn, da brugeren generelt associerer grøn(Princippet om "Consistency and standards") med noget som er valgt. Fanatic var enige i vores valg af farve og Design, og var tilfredse med udviklen.

GUI-design

- Nicolai Johansen, Søren & Andreas

Kvalitetsfaktorer

Usabillity

Fanataics app er designet simpelt og brugervenligt. Alle interesserede og forbipasserende kan være potentielle kunder, så brugervenligheden i appen er vægtet højt ved udvikling af GUI. Derfor skal kunderne ikke have nogen form for it-kompetencer for at benytte appen.

Interopabillity

Da Fanatic har en eksisterende database, som skal kunne kommunikere med appen er Interoperability et krav. I en senere iteration skal appen kunne forbindes til Fanatic Wordspress database.

Portability

Ideen med appen er at den på sigt skal kunne køre på en iPad og en computer i Fanatic butikken. Da appen også på sigt skal være kompatibel med IOS(Styresystemet på en iPad) er vi nødsaget til at vægte Portability højt.

User Interface Design Principles[2]

- Af Nicolai Johansen og Andreas

User Control and Freedom

Gruppen har fokuseret meget på at Fanatics app skal være brugervenlig og ekstrem simpel at benytte. Der skal være frihed og altid mulighed for at komme væk fra den side man befinder sig på. Derfor er der også lagt fokus på "User control and freedom", i form af at der er 2 måder at komme tilbage til forsiden. Logo'et i toppen er en "tilbage knap" som kan benyttes til hver en tid. Nede i venstre hjørne er der endnu en "tilbage knap" som brugere kan genkende.

Consistency and Standards

Der er placeret en "tilbage knap" nede i venstre hjørne, hvor dette princip er benyttet. Da det både kan være IT nybegyndere og dygtige IT brugere der benytter denne app, er det vigtigt med genkendelighed i forhold til hvordan applikationer normalt er bygget op.

Farvevalget til markering af knapperne, altså kundernes interesser, er et bevidst valg med tanke på "Consistency and Standards". De fleste brugere vil forbinde farven grøn med at interessen er valgt. Dette er gjort på trods af fanatics farvevalg på deres hjemmeside, da de benytter farven blå. Der blev også snakket om, at markeringen kunne være en grøn ramme omkring knappen eller skriftfarven kunne skifte farve. Efter møde med Fanatic, valgte ejeren Daniel at hele knappen skulle markeres med den svage grønne farve, da det illustrerer bedst at kundens interesse, blev valgt.

[2] https://www.nngroup

Help users recognize, diagnose, and recover from errors

Vi har benyttet dette princip i form af message boxes som kommer op, hvis man ikke giver korrekt information til appen. Dette kan komme i form af funktionen "Log in", hvis man ikke skriver sit brugernavn, men skriver password ind. Det vil resultere i en messagebox der kommer op og fortæller brugeren at der skal skrives brugernavn ind, for at kunne komme videre.

Visibility of system status

Selvom det er en simpel app, er det vigtigt at fortælle brugere hvor man er henne, under hele forløbet. Det vil blive gjort i form af en sætning under Fanatic logoet, hvor appen fortæller brugeren hvad de skal på siden og

hvad formålet med deres deltagelse er.

Billeder

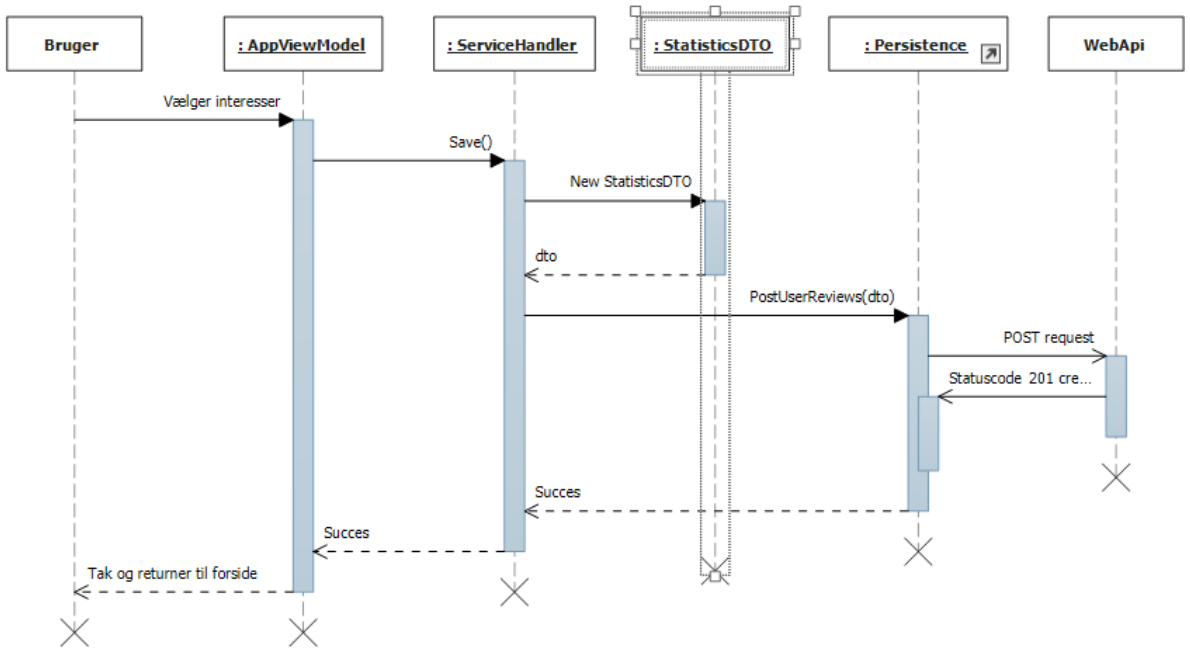
Gruppen er klar over at billederne der er blevet brugt i appen har copyright. Appen vil ikke blive lanceret eller afleveret som færdigt produkt med disse billeder. Det er blevet sagt til Fanatic at det er midlertidige placeholders for billeder som Fanatic selv har sagt de vil levere.

Sekvens Diagrammer

App

- Af Søren og Andreas.

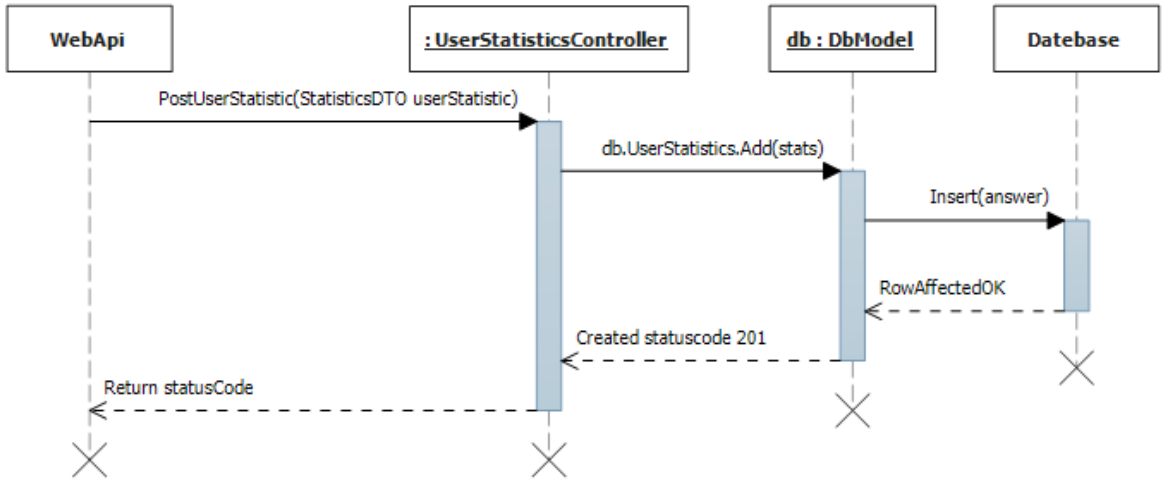
Vi har valgt at benytte os af en ServiceHandler som serviceprovider til vores GUI med underliggende viewmodel. Vores persistence klasse benytter sig så af 'Facade' pattern, da det kun er persistence som interagerer med Databasen(WebAPI). Det gør det lettere for os hvis der eksempelvis senere i forløbet vil komme ændringer i vores WebAPI, da vi så kun skal lave ændringer i persistence klassen uden at resten af systemet bliver meget påvirket. Dette er også medvirkende til GRASP princippet low coupling.



Webservice

- Nicolai Johansen og Nikolaj Bang

Vores web API modtager et Http kald, med et DTO fra vores app hen til UserStatisticsController, som gør brug af vores DbModel, og sætter rækker ind i databasen. Når databasen har fået indsat rækkerne som er blevet modtaget, sender den besked tilbage til DbModel klassen, så vores UserStatisticsController kan sende en response besked tilbage til appen, så appen ved at informationen den sendte var korrekt.



App Design Klasse Diagram

- Af Nikolaj Bang, Andreas og Daniel

Vores design klasse diagram er udviklet ud fra domænemodellen og vores sekvens diagram. Design klasse diagrammet viser klasserne, deres attributter og metoder. Imellem klasserne kan man se forholdet imellem dem. Forholdet er vist med associationer og multipiciteter. De mulige interesser fra Domain modellen, har fået et nyt navn i forbindelse med implementeringen af databasen. Da database tillægget (se database modellering) indeholder "UserStatistic" og "IRTyper" tabellerne, har vi valgt at bruge et DTO kaldet StatisticsDTO, der består af oplysningerne af en kundes valg af interesse i Domain modellen. Kunden er blevet til "UsersTable", da dette var en del af det database udlæg, vi fik af Fanatic's eksisterende database.

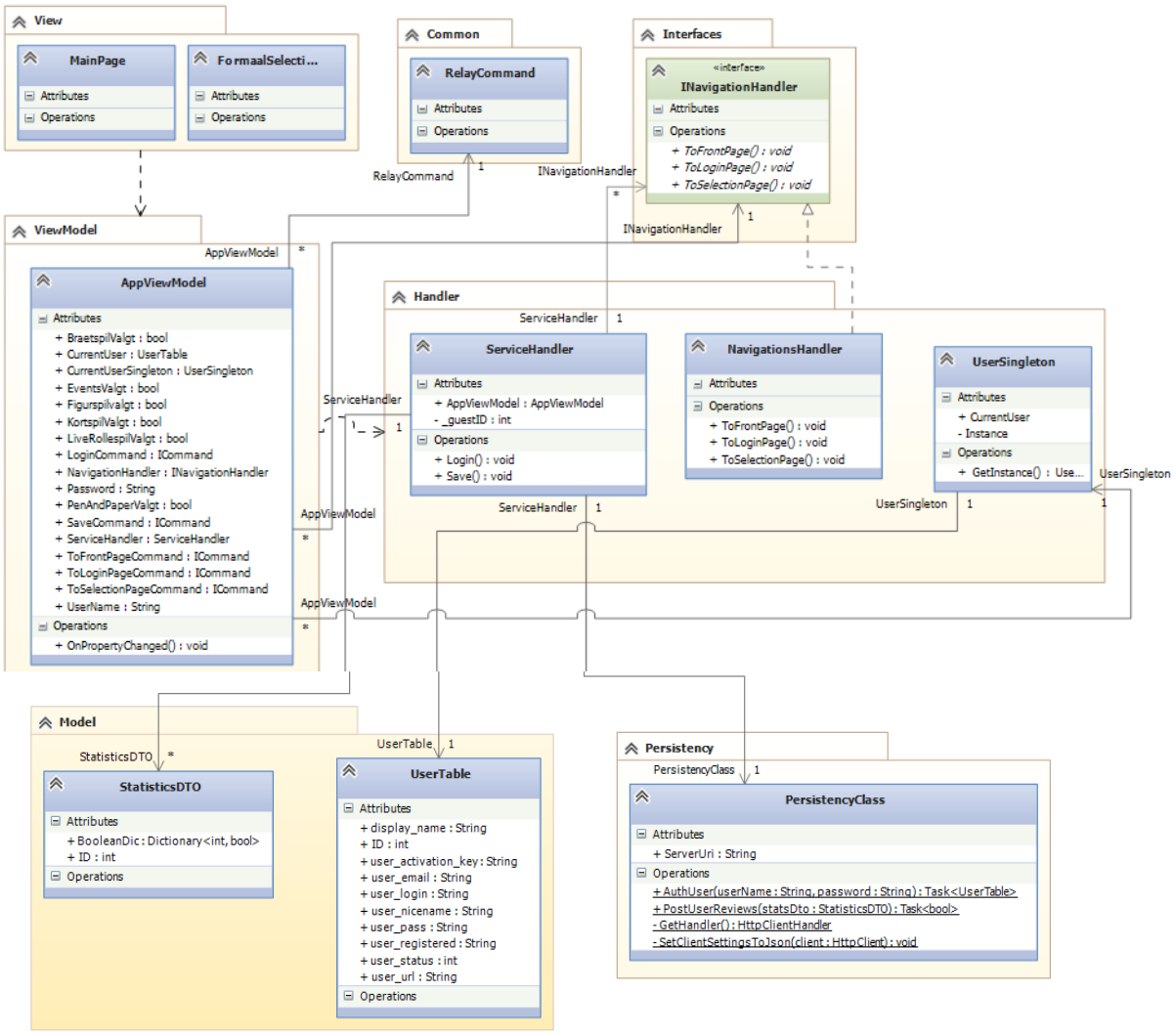
I vores views har vi Mainpage som er den første side brugeren bliver mødt med, ved start af appen. Herefter kan brugeren blive guidet videre til LoginPage, hvor brugeren kan logge ind med brugernavn og kode for at afgive sine interesser på FormaalSelection, eller vælge gæst, så de afgiver sine interesser anonymt. I denne iteration er fokus lagt i at kunne være anonym, dog i sparring med Fanatic skulle forsiden, have mulighed for begge i fremtiden.

Ved siden af vores views kan man se et interface ved navn INavigationhandler som vores Navigationshandler implementerer og vores AppViewModel har en referance til. Det gør at vi kan skifte imellem de forskellige views ved at kalde eksempelvis ToFrontPage metoden.

Vores AppViewModel håndterer alle forespørgsler og input fra view'et. Vi har valgt at lave en klasse som hedder UserSingleton, for at have et objekt, fremfor et static field, som holder styr på, den person som er logget ind, og for at være sikker på at der ikke er to brugere logget ind på samme tid.

I vores Model niveau har vi en klasse som hedder StatisticsDTO, som vi benytter når brugeren har angivet sine interesser. Herefter modtager persistency klassen det Data Transfer Object den bliver givet, og dette DTO bliver overført til webserveren, som behandler det og sørger for at oprette statistics til databasen.

I vores Persistence klasse ligger vores metoder, der håndtere Http kald til vores webservice, her har vi både AuthUser og en PostUserReviews, disse metoder er static, hvilket gør at vi kan kalde metoden på klassen, og bruge den lige der hvor den skal bruges.



Webservice Design Klasse Diagram

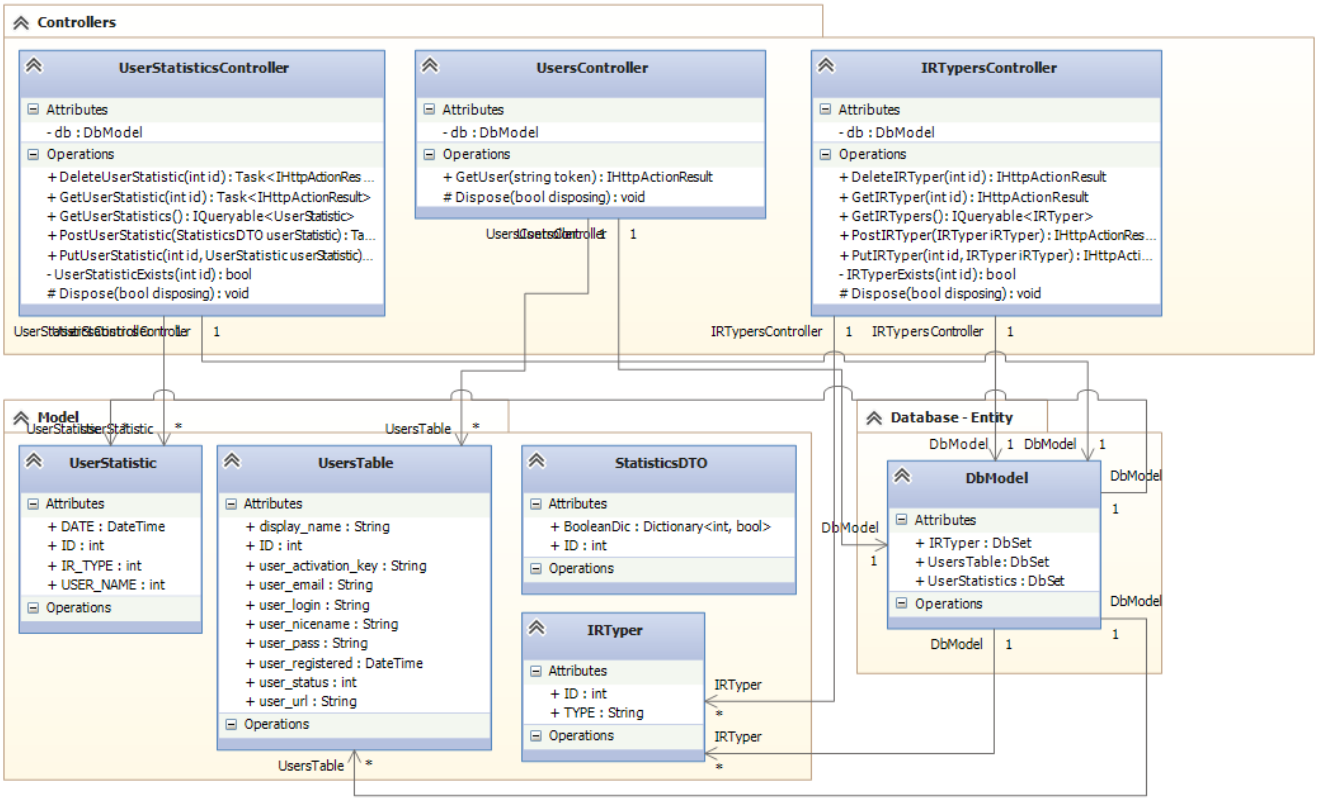
- Nikolaj Bang

Webservicen er baseret på ASP.NET 4.5.2 Web api 2, denne applikation er koblet op til en database, ved brug af Entity Framework. ADO.NET model er implementeret og har genereret klasserne ud fra vores database, samt lavet en DbModel klasse.

De model klasser, såkaldte “entities”, representerer en række i databasens tabeller. DbModel klassen er den klasse, der håndtere de genererede DbSet, det er igennem denne klasse, vi kommunikere med databasens indhold, også kaldet en DBContext.

For denne iteration er UserStatisticsController, den mest relevante klasse at kigge på, da den opretter og gemmer UserStatistic, som opfylder den del af Use casen, hvor det gemmes i en database.

Derudover har vi en StatisticsDTO klasse, dette er fordi vi har ændret i vores UserstatisticsController's PostUserStatistics metode. Denne metode modtager et objekt af typen StatisticsDTO, og laver den om til den rette mængde af UserStatistic, og gemmer dem i databasen.



Database Modellering

- Af Nikolaj Bang, Andreas & Daniel

Med udgangspunkt i ejerens eksisterende database, har vi lavet et database tillæg. I den eksisterende database har vi "Userstable", hvilket består af kundens informationer som f.eks. Login, password, email osv. Database tillægget består i sin enkelthed af to tabeller, butikkens interesse områder og kundens interesse valg. Kunden har muligheden for at vælge en eller flere af pt. seks valgmuligheder, hvor vi har implementeret et til mange forhold, så hvis kunden registrerer flere interesser ad gangen, vil der blot blive indsat flere rækker, så et bedre statistisk forhold af, hvor mange gange den enkle valgmulighed er blevet valgt, derved undgår vi unødig data i databasen i form af tomme kolonner, og eventuelt i fremtiden ville kunne ekspandere interesse valgmuligheder, eller let udskifte en interesse.

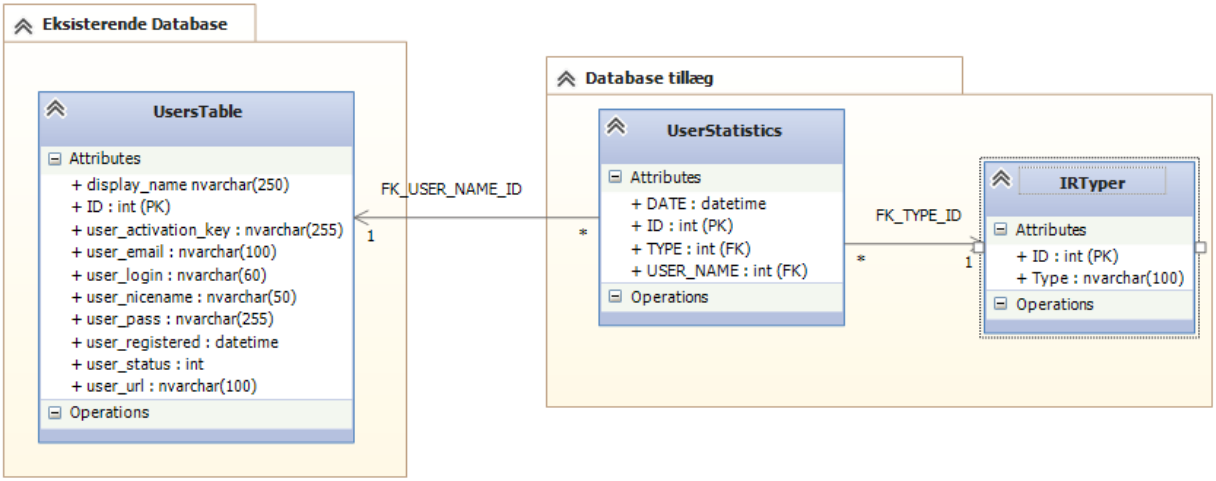
Vi har oprettet et særskilt tabel til kundens interesser valg, så vi opnår et sæt af relateret data. Vi har undgået, at bruge det fulde brugernavn i "UserStatistics", for ikke, at have en masse redundant data i tabellen, så derfor referere vi til en bestemt bruger, via en foreign key "FK_USER_NAME_ID". Denne key fortæller, hvilken bruger fra "UsersTable", som en række i "UserStatistics" referer til.
I "UserStatistics" har vi besluttet, at undgået en composite key som primary key, fordi en bruger kan tilkendegive flere interesser på en gang. Primary key'en for "UserStatistics" opnår vi ved, at bruge en auto incrementing (I MSSQL hedder det identity) integer, denne generes af databasen, med henblik på at være unik. En række i "UserStatistics" har et datetime værdi der identificere, hvornår denne række er blevet indsat i databasen. Ligesom "USER_NAME" er "TYPE" en foreign key, denne key referer til et ID for en række i tabellen "IRTyper", via foreign key "FK_TYPE_ID".

I overvejelserne omkring design af databasen har vi gjort brug af normalisering. Vi har overholdt første normalform ved at sikre os at der kun står én værdi i alle kolonner. Vores userstatistics table kommer altså ikke til at have mere end en Type i hver kolonne, selvom den gældende User måske har flere interesser. For hver Type af interesse vil der blive tilføjet en ny række i tabellen. Dette gør det også lettere at trække statistik ud på "UserStastics" tabellen.

Da vi havde første normalform på plads kunne vi gå videre med anden normalform. Første normalform er et krav for at kunne arbejde på anden normalform. Hvor første normalform har regler for værdierne i cellerne, siger anden normalform noget om relationerne mellem kolonnerne. Vi har sikret os at anden normalform bliver overholdt i databasen, da alle værdier i tabellen er afhængige af den primære nøgle. Ser man nærmere på UsersTable kan man altså se at alle attributter er afhængige af ID som er primær nøgle. Databasen indeholder ikke nogle composite keys og er derfor også i anden normalform.

Databasen passer nu til kravene for første og anden normalform. Den tredje normalform er også en regel for relation mellem kolonnerne. I databasen er der ikke nogle non-key værdier som afhænger af hinanden eller som kan regnes ud hvis man har to andre værdier. De er alle kun afhængige af tabellens primary key så vores database udfylder alle tre regler for normalisering.

Databasen i projektet, samt webservicen, er hostet af Azure Clouding, så vi opnår fleksibilitet, og gør det muligt for mange at kunne kommunikere med den. Azure Clouding bliver lavet af Microsoft og er et hosting sted hvor vi kan have vores server og Database liggende og derved vil kunne tilgå Databasen fra flere maskiner.[3]



MVVM Implementering

- Daniel og Søren

Vi bruger MVVM modellen for at skabe lavere kobling mellem vores modeller, klasser, controllere og GUI. En lavere kobling vil f.eks. sørge for, at en ændring i en klasse har en lavere indflydelse på andre klasser, da der er lavere afhængighed.

MVVM giver en lagdelt arkitektur, som gør det muligt at kunne skifte et lag ud uden, at det vil have en stor indvirkning på funktionaliteten i vores program og det vil være let at implementerer det nye lag. Vi ville eksempelvis hurtigt kunne skifte et View ud, da der ikke er lavet noget code behind til view'et, da det meste kode ligger i vores ViewModel . Dermed vil det ikke være et problem, at lægge et nyt View ind, da de funktioner der skulle bruge allerede ligger i ViewModel laget.

I MVVM skabes der en Datacontext fra vores view til viewmodellen, dette sikre vi kan bruge de funktioner, viewmodellen indeholder.

[3] https://support.microsoft.com/da-dk/kb/283878 30-05-2016

Ved at bruge MVVM kommer vi også til at bruge GRASP da de regler man bruger til at lave MVVM ligger meget op af de Design Principper der er i GRASP blandt andet low kobling og high cohesion.

Client / Server arkitekturen

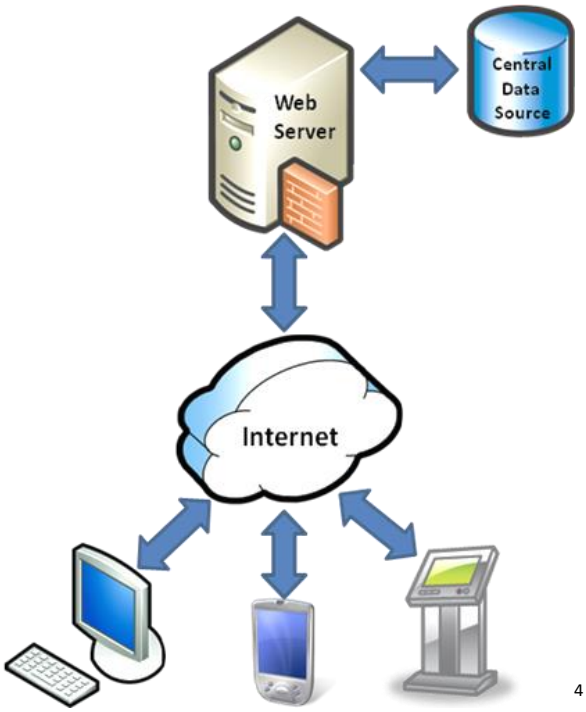
- Af Daniel & Nikolaj Bang

Arkitekturen er delt op sådan, at vi har en klient, server og en database. Hvis klienten er hentet eller installeret på det rette hardware, kan den tilgå en web service, hvis den er koblet til internettet.

Webserveren er hostet f.eks. i Azure Cloud, hvilket sikre at flere kan tilgå denne side ved evt. at logge ind og hente data eller sende data. Webserveren står for at behandle alle de Http requests den møder, give respons til klienterne og kommunikationen med databasen. En Klient tilgår web serveren via en webadresse og en udvidet sti, så den navigerer til den rette API for den kategori af data.

Kommunikationen over internettet foregår med Http protokollen, hvor det er klienten der sender en request, som webserveren besvarer. Klienten modtager svaret fra webserveren, behandler det og evt. giver respons til brugeren af klienten.

Webserveren kommunikere med en database, det kan være lokal database for webserveren, eller også kan den være hostet i f.eks. Azure cloud. Webserveren er den eneste kommunikation til databasen, hvilket sikre en datasikkerhed, så der mindskes muligheden for korrupt data.



[4]

Teknisk gennemgang

- Nikolaj Bang & Nicolai Dreier

Vi har valgt at kommentere vores Save metode, da det har den største relevans for denne iteration og use case.

I begyndelsen af metoden sætter vi et ID på den der bruger appen, hvis venstresiden af '??' er null, bliver user bare sat til et _guestID, som i databasen er 3. Vi har valgt at løse vores problem, hvor vi overfører mere end en valgt interesse, ved hjælp af datastrukturen Dictionary. Her fravælger vi en liste, da vi laver hver af vores formål om til keys, så vi ikke får flere formål af den samme type. Denne Key svarer til interessens Primary Key i databasen, og Value'en svarer til om interessen er valgt, det vil sige true er valgt og false er ikke valgt. I det Save metoden bliver kørt, bliver vores dictionary initialiseret med om interesserne er blevet valgt. Her kunne vi også have gjort brug af et SortedSet af interesse Enum, men grundet manglende erfaring med Enums, er problemet løst med en dictionary.

```
public async void Save() {
      try {
            var user = AppViewModel.CurrentUserSingleton.CurrentUser?.ID ?? _guestID;
```

[4] https://msdn.microsoft.com/en-us/sync/bb887608.aspx 27-05-2016

```
Dictionary<int, bool> bools = new Dictionary<int, bool>() {
    {1, AppViewModel.BraetspilValgt}, {2, AppViewModel.Figurspilvalgt}
    {3, AppViewModel.PenAndPaperValgt}, , {4, AppViewModel.KortspilValgt},
    {5, AppViewModel.LiveRollespilValgt}, {6, AppViewModel.EventsValgt}, };
```

Vi opretter et Data Transfer Object, dette StatisticsDTO tager en Dictionary med key Integer og value af Boolean, dette objekt bruger sender giver vi til vores PersistencyClass's metode PostUserReviews.

```
StatisticsDTO dto = new StatisticsDTO() { BooleanDic = bools, ID = user };
```

PostUserReview melder tilbage om det lykkedes, eller om der opstod en fejl i en HttpResponseMessage. Hvis responsen melder om fejl, kommer der en popup der fortæller brugeren, om der eventuelt mangler en internetforbindelse og at de skal prøve igen, eller kontakte en medarbejder, dog hvis det lykkedes navigeres der til frontpage view'et, samt et popup vindue med "tak". Metoden er allerede optimeret til brug af andre brugere end en gæstebruger, dette ses på linjen hvor vi sætter et bruger id, hvis der er en der er logget ind, ellers bruger den et gæste id. Derudover er den sidste popup besked, også lavet klar til hvis der er en bruger logget ind, så den informere om at brugeren bliver logget ud, som sker på linjen efter.

Hele metoden er inde i en try catch, for at scopet kan tale sammen, samt hvis enten oprettelsen af vores DTO smider en exception, eller hvis der sker en fejl ved kaldet til PersistencyClass metoden PostUserReviews. En alternativ til en app som kun virker online, kunne være at den midlertidigt gemmer svarene, hvorefter den sender dem igen, når der kommer en internetforbindelse der virker.

Vores StatisticsDTO, tjekker om ID er positiv og den dictionary, den får tildelt indeholder mindst et KeyValuePair er true, ellers kaster de en exception, som bliver vist i en popup fra catchen.

```
var httpResponseMessage = await PersistencyClass.PostUserReviews(dto);
if (httpResponseMessage.StatusCode == HttpStatusCode.NotFound) {
   await new MessageDialog("Noget gik galt.\nKontakt medarbejder").ShowAsync(); }
if (httpResponseMessage.IsSuccessStatusCode) {
   AppViewModel.NavigationHandler.ToFrontPage();
   await new MessageDialog("Tak for dit svar!" +
     (AppViewModel.CurrentUserSingleton.CurrentUser != null
      ? "\nDu bliver automatisk logget ud" : "")).ShowAsync();
        AppViewModel.CurrentUserSingleton.CurrentUser = null;
```

```
            }
```

Vores StatisticsDTO, tjekker om ID er positiv og den dictionary, den får tildelt indeholder mindst et KeyValuePair er true, ellers kaster de en exception, som bliver vist i en popup fra catchen.

```
        }
        catch (Exception e){
            await new MessageDialog(e.Message).ShowAsync();
        }
}
```

Udover denne funktion, som indebærer den funktionelle fokus i iterationen, har vi opsat projektet efter MVVM og en Client/server arkitektur. Dette gør at vi opnår en high cohesion og low coupling, samt har vi en creator, som er Save metoden, der opretter og sender et DTO ved hjælp af PersistencyClass. PersistencyClass er kodet til lige det formål, vores 1 iterations app har, at sende og oprette nye elementer ved brug af POST, på vores web api.

Vores AppViewModel agere som controlleren for view'et, denne controller indeholder properties som alle er databinded til xaml controls i view'sne. Når der skal ske noget funktionallitet, bliver serviceprovideren "ServiceHandler" brugt, denne klasse indeholder metoder, som AppViewModel gør brug af. For at navigere imellem vores views, gør vi brug af en NavigationHandler, den gør brug af en xaml control, der bliver gemt som en Frame, der gør man kan navigere med lavere coupling end codebehind.

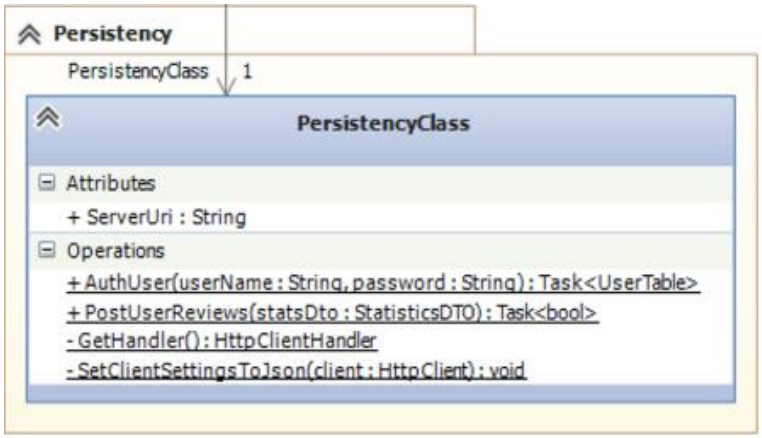
Persistens implementering

- Af Nikolaj, Andreas & Daniel

Som et led i vores program har appen en database, hvor al data ligger i. For at denne database skal kunne tilgås, har vi oprettet nogle forskellige metoder, som kan bruges til sende dataen. Til dette har vi en persistens klasse, som indeholder fire metoder. I denne iteration er det metoden "PostUserReviewsDto" som er spændende at kigge på. For at lagre kundens interesser lagrer vi dataen i et "Data Transfer Objekt" og bruger objektet som argument når "PostUserReviewsDto" bliver kaldt. "PostUserReviewsDto" gør brug af de to private hjælper metoder i klassen for at kunne kommunikere med webservicen. Først benytter den "GetHandler" metoden til at oprette en HttpClientHandler, med default bruger information. Efter clienten er oprettet definerer vi at den skal operere med Json format ved hjælp af metoden "SetClientSettingsToJson".

Alle Http kald bliver sendt til vores Webservice, som har en RESTful Api, vores RESTful Api består af en routed sti. Den fortæller hvilken controller der skal blive kaldt på serversiden, samt hvilken metode. Inde i controllerne har vi en eller flere metoder for GET, PUT, POST og DELETE, disse metoder bliver udpeget ud fra headers og stien på det indkommende Http kald.

Webservicen modtager et Http kald, hvori den finder ud af hvilken controller der skal behandle det. Vores UserStatisticsController modtager et kald, behandler det, og bruger Entity Framework til at kommunikere med vores database.




```
private const string ServerUri = "http://fanaticprojekt.azurewebsites.net/";
public static async Task<HttpResponseMessage> PostUserReviews(StatisticsDTO statsDto) {
    using (var client = new HttpClient(GetHandler())) {
        try {
            SetClientSettingsToJson(client);
            var response = await client.PostAsJsonAsync("api/UserStatistics", statsDto);
            return response;
            }
            catch (Exception e) {
               return new HttpResponseMessage(HttpStatusCode.NotFound) {ReasonPhrase = e.Message};
            }
    }
}
private static void SetClientSettingsToJson(HttpClient client) {
```

```
    client.BaseAddress = new Uri(ServerUri);
    client.DefaultRequestHeaders.Clear();
    client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
}
private static HttpClientHandler GetHandler() {
    return new HttpClientHandler() { UseDefaultCredentials = true };
}
```

UnitTests

- Af Nikolaj Bang

Vi har opstillet 10 test cases, hvor vi har valgt at fokusere på det Data Transfer Object "StatisticsDto", som bliver oprettet, når man vælger sine interesser i appen. Derudover har vi også valgt at, teste hvilke scenarier der kan forekomme, når vi sender dette objekt til webservicen.

Vi har valgt at fokusere på Blackbox-testing via Unit testing, da vi har en deadline den 31. maj 2016, hvor vi gerne vil være sikker på at metoderne virker korrekt.

I vores Blackbox-testing, bliver testene udført af en der kender til implementationen, dette sikre at testene runder de forventede fejlmeddelelser. Vi laver også en integrationstest, da vi tester vores Post af statisticsDto til vores Webservice i Azure Cloud.

Vi har valgt at undgå at risiko-testing via Whitebox, da dette ville være for stor en ting for dette projekts tidsramme, fordi Whitebox-testing er meget mere omfattende.

Test case #	Description of test case	Expected value	Passed successfully
1	StatisticsDto.ID = 0	0	yes
2	StatisticsDto.ID = -1	ArgumentException	yes
3	StatisticsDto.ID = 1	1	yes
4	StatisticsDto.ID = Int32.MaxValue	2147483647	yes
5	StatisticsDto.ID = Int32.MinValue	ArgumentException	yes
6	Unchecked { StatisticsDto.ID = Int32.MaxValue + 1 }	ArgumentException	yes
7	All BooleanDic Values are false	ArgumentException	yes

8	BooleanDic = null	NullReferenceException	yes
9	Post a single review to the webservice	HttpStatusCode.Created	yes
10	Try posting with non existing ID	HttpStatusCode.NotFound	yes

UnitTests code

Initialize af test cases

```
public StatisticsDTO StatisticsDto { get; set; }
[TestInitialize]
public void BeforeEachTest() {
    StatisticsDto = new StatisticsDTO();
    StatisticsDto.ID = 3; // GuestID i databasen
}
```

Test case #1

```
[TestMethod]
public void UserIdZero() {
    StatisticsDto.ID = 0;
    Assert.AreEqual(0, StatisticsDto.ID);
}
```

Test case #2

```
[TestMethod]
public void UserIdBelowZero() {
    Assert.ThrowsException<ArgumentException>(() => {
       StatisticsDto.ID = -1; });
}
```

Test case #3

```
[TestMethod]
public void UserIdAboveZero() {
    StatisticsDto.ID = 1;
    Assert.AreEqual(1, StatisticsDto.ID);
```

```
}
```

Test case #4

```
[TestMethod]
public void UserIdMaxPositiveInt() {
    StatisticsDto.ID = Int32.MaxValue;
    Assert.AreEqual(Int32.MaxValue, StatisticsDto.ID);
}
```

Test case #5

```
[TestMethod]
public void UserIdMaxNegativeInt() {
    Assert.ThrowsException<ArgumentException>(() => {
        StatisticsDto.ID = Int32.MinValue; });
}
```

Test case #6

```
[TestMethod]
public void UserIdBorderTest(){
    unchecked { Assert.ThrowsException<ArgumentException>(() => {
          StatisticsDto.ID = Int32.MaxValue + 1;
          Assert.AreEqual(Int32.MinValue, StatisticsDto.ID);}); }
}
```

Test case #7

```
[TestMethod]
public void BooleanDicAllFalse() {
    Assert.ThrowsException<ArgumentException>(() => {
        StatisticsDto.BooleanDic = new Dictionary<int, bool>()
        { {1, false }, {2, false }, {3, false },{4, false },{5, false },{6, false },};});
}
```

Test case #8

```
[TestMethod]
public void BooleanDicNull() {
    Assert.ThrowsException<NullReferenceException>(() => {
        StatisticsDto.BooleanDic = null; });
```

```
}
```

Test case #9

```
[TestMethod]
public void TestPostReviewsSingle() {
    StatisticsDto.BooleanDic = new Dictionary<int, bool>() {
    {1, false }, {2, false }, {3, true }, {4, false }, {5, true }, {6, true }, };
    var t1 = PersistencyClass.PostUserReviews(StatisticsDto).Result;
    if (t1.IsSuccessStatusCode) {
        Assert.AreEqual(HttpStatusCode.Created, t1.StatusCode);
    } else {
        Assert.Fail();
    }
}
```

Test case #10

```
[TestMethod]
public void TestPostReviewsNonExistingId() {
    StatisticsDto.BooleanDic = new Dictionary<int, bool>() {
    {1, true }, {2, true }, {3, false }, {4, false }, {5, false }, {6, false },};
    StatisticsDto.ID = 333; // ID that doesnt exist
    var t1 = PersistencyClass.PostUserReviews(StatisticsDto).Result;
    Assert.AreEqual(HttpStatusCode.NotFound, t1.StatusCode);
}
```

Systemtest review & beta testing

- Af Nicolai Dreier

Beta Test

Da vores app skal benyttes af både nybegyndere og øvede kunder, går vi meget op i usabillity. Derfor har vi valgt at få 4 grupper til at afprøve vores app og se om vores krav er realistiske. Fra hver gruppe blev der valgt 1 person til at afprøve appen. Vi fortalte personen at de skulle forestille sig at gå ind i butikken Fanatic og så stod der en computer med touch (som test) som appen kørte på.

Appen kørte uden fejl ved test af alle 4 grupper.

1 gruppes tid: 13,99 sekunder

2 gruppes tid: 29,13 sekunder

3 gruppes tid: 7,91 sekunder

4 gruppes tid: 14,07 sekunder

Gennemsnitstiden for alle grupper er: 16,275 sekunder. Dermed opfylder det vores krav om use casen til maximum 30 sekunder.

Fælles for tiderne som trækker gennemsnittet op er at brugeren går i stå ved valg af interesser. Feedback fra testpersonerne viser at designet af view'et ikke viser tydeligt nok hvad der ønskes af dem. Derfor har vi gjort teksten større så det er lettere at få øje på.

System Test

Hvis kunden ikke vælger et formål på "FormaalSelection" siden kaster vi en exception som håndtere dette:

```
public Dictionary<int, bool> BooleanDic {
     get { return _booleanDic; }
     set { if (value.Count(x => !x.Value) == value.Count) {
               throw new ArgumentException("Vælg venligst et formål.\nEller tryk på tilbage knappen.");}
         _booleanDic = value;
     }
}
```

Messages boksen giver kunden en mulighed for at vælge et formål, eller trykke på tilbage knappen hvis kunden kom ind på siden ved en fejl.

Hvis vores database, webservice eller internet er nede vil kunden blive bedt om at kontakte en medarbejder. Der bliver sørget for at kunden bliver fortalt, hvad der er galt og hvad der skal gøres for at fikse det.

```
{
        await new MessageDialog("Noget gik galt.\n Kontakt medarbejder").ShowAsync();
}
```

Revurdering af risikoanalyse

- Af Nicolai Johansen

Risikolisten er blevet opdateret igennem hele projeket. I risk listen har vi valgt Magnitude på en skala fra 1-10, hvor 10 er højest og 1 er mindst.

Type	*Beskrivelse*	*Magnitude*	*Impact*	*Indicator*	*Mitigation*
Technical	Får lavet for meget uden planlægning (waterfall)	7 -> 2	High	Udfører arbejde ud fra løbende ideer	Gruppemøde for at planlægge forløbet.
Technical	Manglende erfaring med versionsstyring kan forårsage tab af data	7 -> 4	Critical	Mistet arbejde	Planlægge brugen af Github og undervise hinanden.
Communication	Hvis vi laver det anderledes end det kunden havde visualiseret det.	6 -> 3	medium	Kunde kan ikke nikke genkendende til det udførte arbejde	Bedre kommunikation med kunde evt. I form af mock ups.

Technical - Får lavet for meget uden planlægning

Dette har ikke været et problem i første iteration, da vi har håndteret det med et online scrumboard på trello. Der er kun blevet produceret de produkter og opgaver som er blevet planlagt. Gruppen har holdt evaluerings møder hver morgen, hvor der er blevet kigget på hvilke opgaver der skulle lægges for dagen.

Technical - Manglende erfaring med versionsstyring kan forårsage tab af data

Gruppen har ikke været udsat for datatab, men der har været problemer med GitHub, som var det planlagte program der skulle bruges til projektet håndtering. Filen ".gitignore", ignorerer filer fra Visual Studio når man downloader projektet derfra. Det medførte at projektet og koden og alle views ikke fungerede korrekt. Det færdige projekt blev samlet på ét USB stick og givet til de andre når der kom opdateringer i projektet. Sent i projektet blev dette fikset på GitHub, så vi kunne dele, gemme og downloade derfra.

Communication - Hvis vi laver det anderledes end det kunden havde visualiseret det

Eftersom Fanatic er placeret i Roskilde, og vi har en kontaktperson i gruppen, har kommunikationen med ejeren været god. Gruppen har designet appen efter de mock ups der er blevet vist og accepteret af kunden. Vi har sørget for at kunden altid har været med på sidelinjen, når der blev taget beslutninger om design og visualisering af appen, så kunden fik det bedste resultat.

1. Iteration afrunding

- Af Andreas

Som gruppe har vi igennem første iteration været velforberedte på hvilke udfordringer vi kunne løbe ind i undervejs. Det har vi gjort blandt andet med vores Risk List, som er med til at gøre den første iteration en succes. Vi har sørget for at opretholde en god kontakt med Fanatic, for at være sikre på at vi møder deres forventninger og budt ind med egne ideer, så vi samtidig kan guide til et godt resultat. I første iteration har vi også brugt meget tid på design, da Temaet for appen og farvevalg skulle klarlægges. Det lykkedes os at lave et simpelt og brugervenligt design, og vores arkitektur fungerer som ønsket.

Vi har benyttet Trello som projektstyring og det har givet os en god visualisering af hvor langt vi har været i processen.

Efter denne iteration går vi i gang med næste use case, som omhandler ønsket om at kunne trække data ud af databasen.

Elaboration - 2. iteration

- af Andreas

I anden iteration vil vi arbejde med use casen "Trække data ud af databasen". Denne use case tager udgangspunkt i at Fanatic skal kunne trække data ud af databasen, og bruge dette til statistik.
Vi ønsker at bibeholde og arbejde videre på den gode arkitektur og være tro mod det design, som gruppen har udarbejdet, og som Fanatic har været imponeret over.

Second use case

- Af Daniel

Use case name: Trække data ud af databasen

Scope: Fanatic Statistik App

Level: User goal

Primary Actor: Fanatic

Stakeholders and Interests:

Fanatic vil gerne trække data ud af databasen, fordi de gerne vil føre statistik over deres kunders formål.

Preconditions:

Der skal være kunder der har brugt Appen, så der er data i databasen.

Success Guarantee: Fanatic trækker data ud af databasen.

Main Success Scenario (Basic Flow):

1. Fanatic trækker data ud af databasen.
3. Fanatic bruger data til statistik.

Extensions (eller alternativ flow)

1. A Ingen internetforbindelse
   Hvis dette er tilfældet håndterer vi det med en try/catch, hvor der bliver sendt en besked til brugeren i form af en pop up boks med teksten: "*Noget gik galt! Tjek evt. om der er internetforbindelse, eller kontakt medarbejder i butikken!"*

Special Requirements

Systemet skal køre på tablet.

I forbindelse med FURPS er det vigtigt for os at Usability er implementeret i vores app. Den skal være let og hurtig at gå til.

Technology and Data Variations List

Det bliver lavet som en windows app.

Fanatics tablet skal have internetadgang for at kunne sende data.

Internetforbindelse på 10/1 minimum for en god forbindelse og respons tid.

Frequency of Occurrence

Fanatic trækker data fra databasen 10 - 30 gange om måneden.

Miscellaneous

Det skal skrives i C#.

Mål med prototype

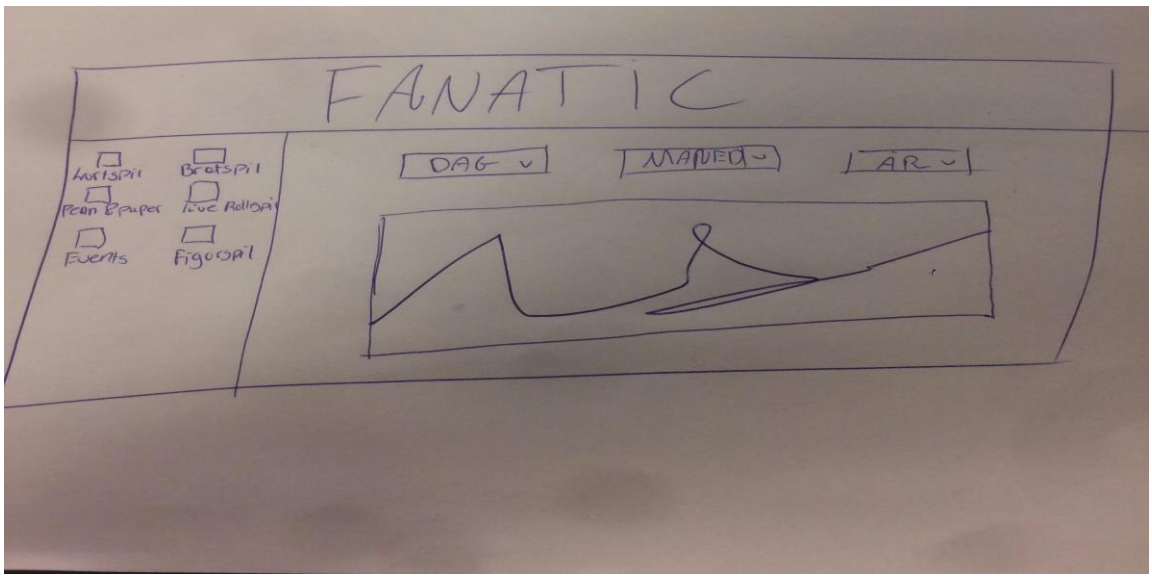
- Af Nicolai Johansen

Gruppens mål med Admin Appen, er at Fanatic får mulighed for at trække den ønskede statistik ud på deres kunders interesser.

Beskrivelse af "eksperimentet" / prototypen

- Af Nicolai Johansen

Prototypen af admin appen er blevet udarbejdet fra vores allerede eksisterende design fra kunde appen. AdminAppens inspiration kommer igen fra simplicitet, da det skal være nemt og hurtigt for Fanatic at se deres statistikker. Imens Fanatic har Kundeappen til at stå ude i butikken, skal Daniel have en anden app, AdminAppen. Denne app skal køre på udelukkende Fanatics egne computere eller telefoner. Når dagen, måneden eller året så er gået, kan Daniel trække statistikker ud på hvad hans kunder kommer efter i butikken.



GUI & Grafer

- Af Søren og Nicolai

Til denne Use case skulle vi kunne trække data ud af databasen, og bruge det til statistik og få lavet en graf ud af det. Men da vi ikke kan udarbejde grafer i Visual studios som udgangspunkt, fandt vi frem til et framework på nettet, fra et firma som hedder Syncfusion[5]. De laver en masse extensions til Visual Studio, som blandt andet gør at man kan lave grafer.

Selve graf feltet bliver lavet i Xaml og bliver skrevet på samme måde som f.eks. en button.

[5] https://www.syncfusion.com/products/uwp 30-05-2016

Vores Graf tager Info fra en Collection og forskellige properties fra de objekter som ligger i Collectionen er blevet bindet på de 2 akser.

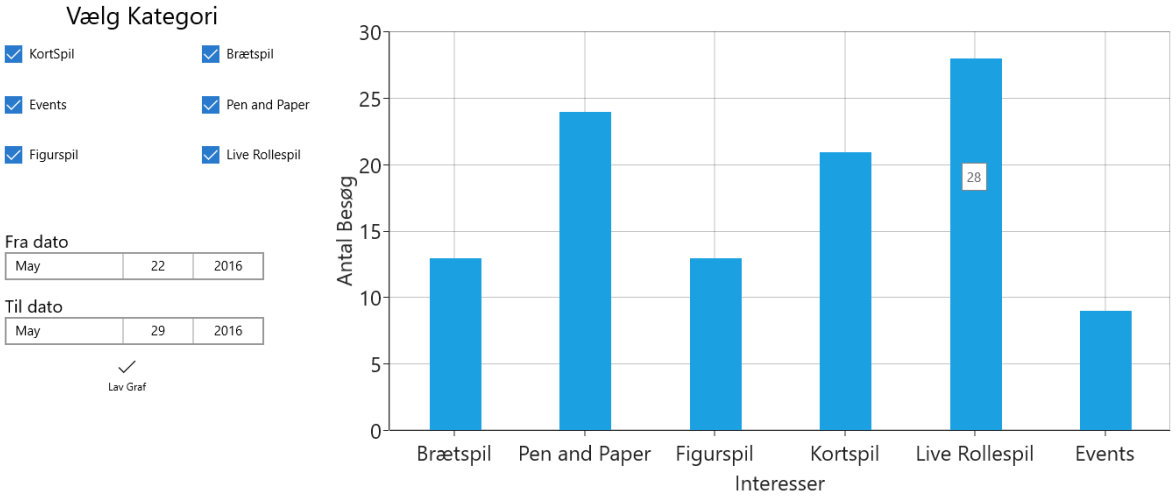
"PrimaryAxis" og "SecondaryAxis" er de 2 akser som ligger på grafen, man kan lave forskellige typer akser, som kan tage forskellige informationer og bruge det til at lave grafen ud fra. På vores "PrimaryAxis" bruger vi det der hedder en en "CategoryAxis", da den kan tage en "String" og bruge som info på en af akserne, som vi har gjort i vores tilfælde.

På vores "SecondaryAxis" har vi lavet en "NumericalAxis", da denne blandt andet kan tage "ints" og "Double" som parameter.

Vi kan igennem frameworket lave forskellige typer grafer, som Piecharts, linecharts eller ColumnCharts. Vi har valgt at lave ColumnCharts til denne iteration, da vi mener at det er den bedste måde at visualisere vores data lige nu. Man kan se i Xaml koden at der er forskellige properties fra de klasser der ligger i vores collection, på vores "PrimaryAxis" er name propertien blevet bindet og på vores "SecondaryAxis" er deres Count blevet bindet, og ud fra de data vil den så kunne lave den graf som vi bruger i billedet.

```
<Grid x:Name="GraphGrid" HorizontalAlignment="Left" Height="530" Margin="415,290,0,0" VerticalAlignment="Top" Width="960">
    <charts:SfChart>
        <charts:SfChart.PrimaryAxis>
            <charts:CategoryAxis Header="Interesser" FontSize="24"/>
        </charts:SfChart.PrimaryAxis>
        <charts:SfChart.SecondaryAxis>
            <charts:NumericalAxis Header="Antal Besøg" FontSize="24"/>
        </charts:SfChart.SecondaryAxis>
        <charts:ColumnSeries ItemsSource="{Binding DataSingleton.CurrentGraph}"
                             XBindingPath="Name" YBindingPath="Count" ShowTooltip="True" Spacing="0.6"/>
    </charts:SfChart>
</Grid>
```

FANATIC



Adminappen er inspireret af vores kundeapp, med simpelt design, hvor Fanatic nemt og hurtigt kan trække data ud. SyncFusion er et Framework vi har benyttet for at få det til at blive mere visuelt. Da det er en backend app, går vi ikke meget op i at det skal se "flot" ud, men mere at det skal være brugbart.

Error Prevention

Hvis Daniel åbner Adminappen og ikke vælger nogle interesser, så kommer der en messagedialog box op og siger: "*Vælg venligst en eller flere kategorier".* Hvis der ikke er internet, håndterer vi også dette med en messagedialog box som fortæller: *"Tjek evt. om der er internetforbindelse"*.

Teknisk gennemgang

- Af Nikolaj Bang

Metoden GraphInit sørger for at hente data fra vores webservice, og fylde vores liste i DataSingleton klassen op. Den gør brug af SmartClient, som returnere en liste med objekter af typen, man specificere. Metoden er en Task, fordi vi gerne vil kunne awaite den i GraphSort funktionen, hvis webservicen ikke svarer indenfor få sekunder, da vi kører funktionen i constructoren for denne ServiceHandler. I denne iteration har vi implementeret et View i vores database, som hedder StatsJoinedIr, deraf typen vi henter, denne type er et join imellem IRTYPER og UserStatistic[6].

```
public async Task GraphInit() {
```

[6] Se Elaboration - 2. iteration, Database modellering diagram i bilag

```
  try {
    AppViewModel.DataSingleton.Statistics = await SmartClient.Get<StatsJoinedIr>(ServerUri, StatsApi);
     } catch (Exception e) {
       await new MessageDialog(e.Message).ShowAsync();
}
```

Metoden GraphSort er i lange træk, den der bliver udført, når man i view'et klikker på knappen "Lav graf". Metoden tjekker først om der er en internet forbindelse, dette er for at undgå et infinite loop senere i koden, fordi vi gør brug af et rekursivt kald til funktionen selv, når data'en er blevet loadet.

```
public async void GraphSort() {
  try {
    if (!SmartClient.CheckForInternetConnection())
       throw new EndOfStreamException("Check om der evt er en internet forbindelse");
    if (AppViewModel.DataSingleton.Statistics != null) {
       AppViewModel.DataSingleton.CurrentGraph.Clear();
       var dateFrom = AppViewModel.DateTimeOffsetFrom.Date;
       var dateTo = AppViewModel.DateTimeOffsetTo.Date;
       Dictionary<int, bool> bools = new Dictionary<int, bool>() {
       {1, AppViewModel.Braetspil}, {2, AppViewModel.FigurSpil},
       {3, AppViewModel.PenAndPaper}, {4, AppViewModel.Kortspil},
        {5,  AppViewModel.LiveRollespil}, {6, AppViewModel.Events}, };
         if (bools.Count(x => !x.Value) == bools.Count)
            throw new ArgumentException("Vælg venligst en eller flere kategorier!");
```

Metoden kigger på AppViewModel, for at finde ud af hvordan sorteringen skal foregå, så den ønskede graf kan blive vist. Derudover kigger den på om der overhovedet er valgt en kategori, for at fortælle brugeren at, der skal vælges mindst én kategori.

```
         var valgteInteresser = bools.Where(x => x.Value).Select(x => x.Key).ToList();
         List<List<StatsJoinedIr>> mergedList = new List<List<StatsJoinedIr>>();
         foreach (var interesse in valgteInteresser) {
```

```
            var gp = AppViewModel.DataSingleton.Statistics.Where(x => x.IR_TYPE ==          interesse &&
x.DATE.Date <= dateTo && x.DATE.Date >= dateFrom).ToList();
             mergedList.Add(gp);
           }
```

Metoden udvælger hvilke interesser der er valgt, og laver en liste af integers. Dernæst opretter vi en liste af lister med StatsJoinedIr, hvor vi i foreach statementet looper igennem valgteInteresser, og finder de StatsJoinedIr's der opfylder brugerens krav. Den laver en gp(graphpoint) liste for hver af de valgte interesser, hvorefter den tilføjer dem til mergedList.

```
           foreach (var liste in mergedList) {
             AppViewModel.DataSingleton.CurrentGraph.Add(new GraphData()
             { Count = liste.Count, Name = liste.FirstOrDefault()?.TYPE });
           }
       } else {
         await GraphInit();
         GraphSort();
       }
    } catch (Exception e) {
       await new MessageDialog(e.Message).ShowAsync();
      }
}
```

Når metoden looper igennem mergedList, opretter den GraphData, der består af hvor mange af interesse typen der er og navn på interessen. Dette tilføjer den til en observablecollection, som bruges i view'et.

Else delen er om listen af StatsJoinedIr er tom eller ej, her køres GraphInit igen, for at hente data, hvorefter vi kalder GraphSort rekursivt, så sorteringen kan ske på at der rent faktisk er data, eller brugeren kan få at vide om der ikke længere er en internetforbindelse.

Konklusion

- Fælles

"How can Unified Process (UP), C#-Programming and Relational Databases be used for development and implementation of a minor IT system?"

Igennem dette forløb har vi arbejdet med Unified Process (UP) til at udvikle en app for rollespils butikken Fanatic. Unified Process har givet os redskaber som har hjulpet os til at afdække krav omkring systemet, og skabt overblik over projektforløbet og programmet. Med sparring med Fanatic har vi udviklet 2 apper, Kundeappen, som hjælper med at få overblik over hvad kundens interesser er, og admin appen, som giver mulighed for at Daniel (ejeren) kan trække statistikker ud på kundernes interesser.
Prototypen er udviklet ved hjælp af C#, modelleret af design patterns og programmerings principper. Prototypen indeholder en app der er tilkoblet en web service som kommunikerer sammen med en database som ligger på Azure. Databasen er modelleret efter de 3 normalformer som gør at vi har et solidt design og gør det nemt at trække data ud.
Da simplicitet har været den røde tråd igennem udviklingen af appen, ville vi benytte grafer til visning af udtræk fra databasen. Derfor har vi benyttet et framework som Visual Studio ikke havde, til at få det vist mere visuelt.
Overordnet har vi udviklet en prototype der opfylder Fanatics behov om at måle statistikker på deres kunder, og udtrække data'en fra deres database.
trello visualisering af arbejdsprocessen

Evaluering af arbejdsprocess

- Fælles

Unified Process har været godt til at holde et overblik over vores system. I forhold til størrelsen på vores projekt har der været brugt meget tid på design og udarbejdelse af artefakter for at demonstrere at vi ved hvordan de skal bruges. Hvis vi arbejdede færdig med vores apper ville det naturligt blive større, og derfor ville UP være essentielt for at få et godt produkt. Her ville eksempelvis Design klasse diagrammet og sekvens diagrammet være gode til, at give et visuelt overblik over klassernes associationer, og hvordan metoderne bliver eksekveret i systemet.
Projektet er blevet styret ved hjælp af en roterende projektleder rolle, som har uddelegeret opgaver igennem Trello. Da projektet kun dækker to iterationer har det været svært at holde de roterende roller. Dette har ikke

været et problem for gruppens måde at arbejde på, da vi har haft gode projektstyringsværktøjer som eksempelvis Trello, som gav en god visualisering af hvor langt vi var i projektet. Udfordringen kunne være undgået ved faste roller i projektet.
Vi har benyttet C# og XAML kode, som har været godt og simpelt at programmere i. XAML kode var lidt en udfordring da vi skulle lære hvordan SyncFusion fungerede, så graferne kunne fungere.
Til projektdeling og projektstyring har vi benyttet GitHub, hvor der har været problemer med ".gitignore" filen, som visual studio havde genereret. Dette kunne være blevet forebygget med en kursus video på eksempelvis Lynda.com omkring GitHub.
Database modelleringen har været en udfordring, fordi vi har arbejdet ud fra en allerede eksisterende database. Dette har været et problem fordi navne konventionen har været påtvunget i tabellen "UsersTable". Det er lykkedes os at lave et godt database design på trods af udfordringerne.
Der har været problemer med vores WebService, da den "går i dvale", dvs. når en kunde benytter den efter en lang pause, vil den være meget langsom første gang man bruger den efter pausen.

Perspektivering

- Fælles

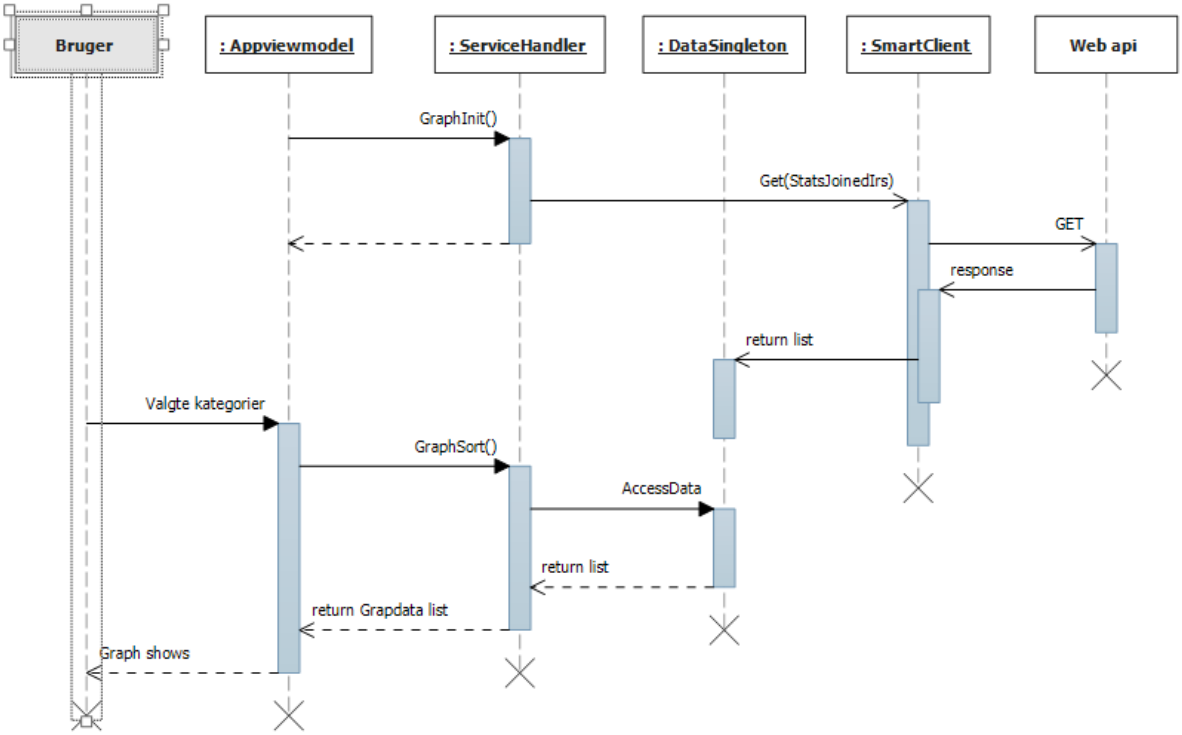
Hvis apperne skulle implementeres i Fanatics system, ville vi mangle en del iterationer som f. eks Log Ind og Opret Bruger. Daniel vil rigtig gerne have Kundeappen over på en Ipad, så det kunne der arbejdes videre med. Hibernate tilstanden som webservicen går i, kunne også blive repareret, så den ikke ville være langsom på noget tidspunkt.

Litteraturliste

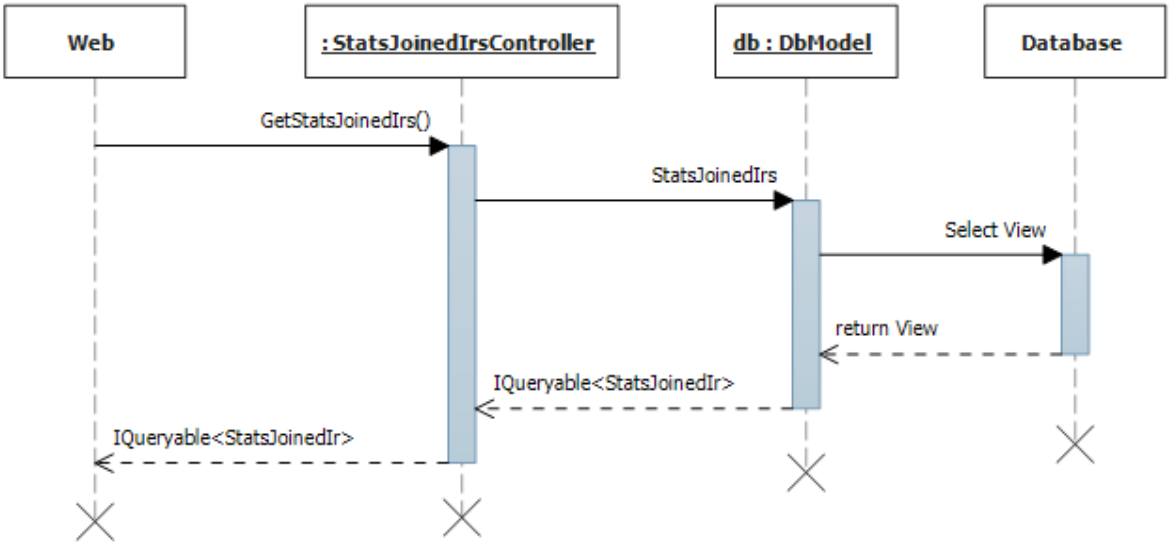
https://msdn.microsoft.com/en-us/sync/bb887608.aspx 27-05-2016
https://www.syncfusion.com/products/uwp 30-05-2016
https://support.microsoft.com/da-dk/kb/283878 30-05-2016

Bilag

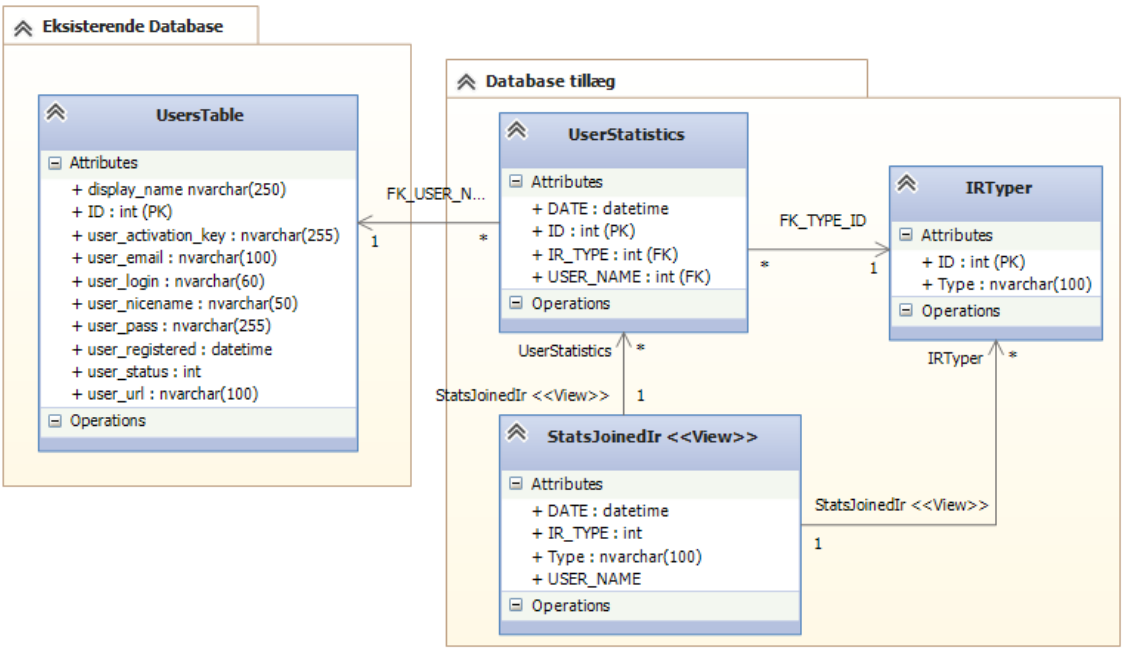
Elaboration - 2. iteration, Design sekvensdiagram



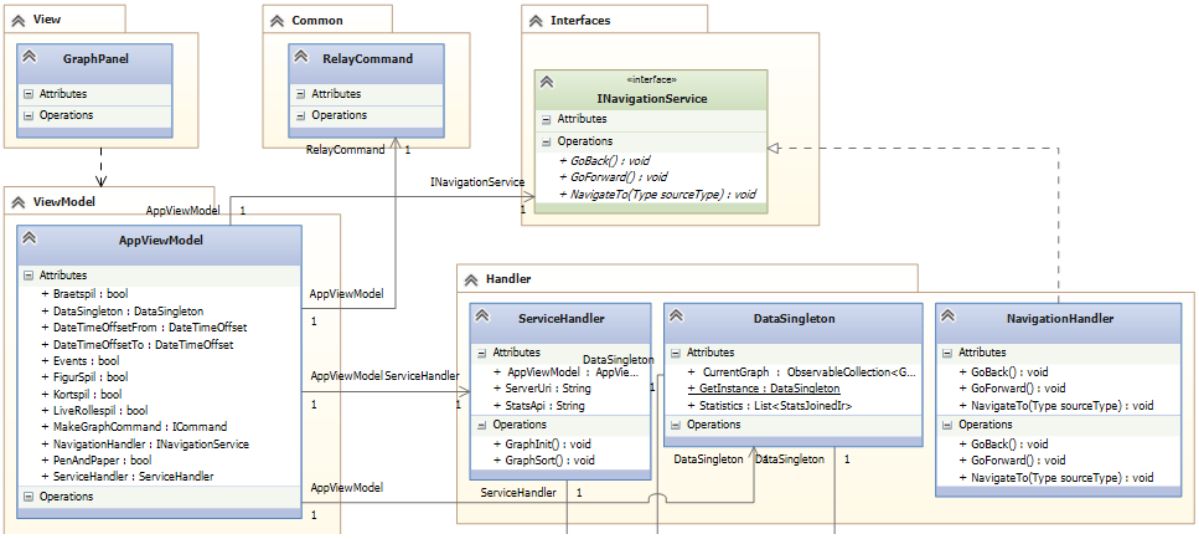
Elaboration 2. iteration, Webservice sekvensdiagram

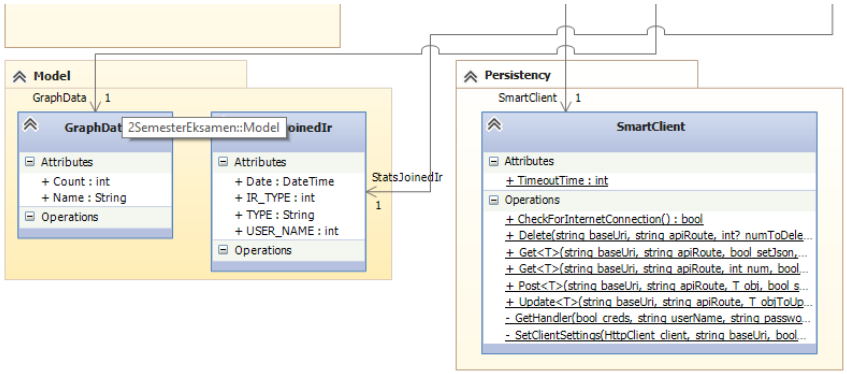


Elaboration - 2. iteration, Database modellering

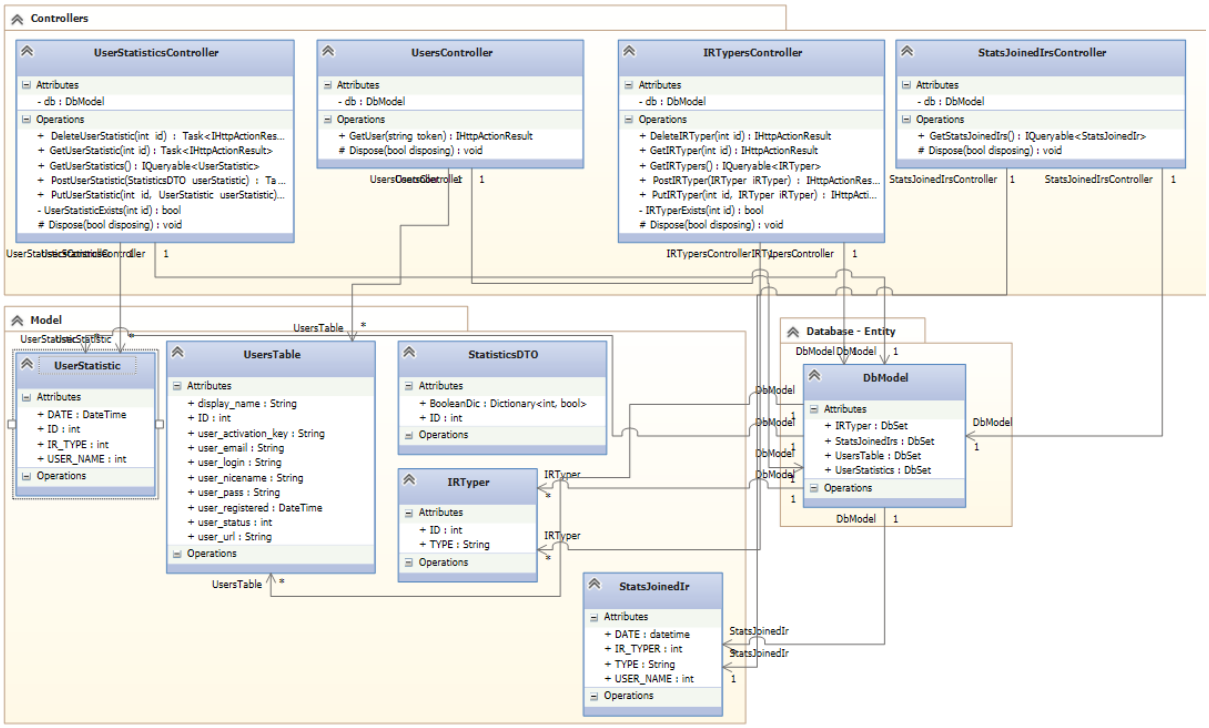


Elaboration 2. Iteration, Admin app klasse diagram





Elaboration 2. Iteration, Webservice klasse diagram



Versionstyringssti

https://github.com/Bang0123/2semprojekt