Webbaseret CRM-system
EqualSums

Peter L. Thomsen Gregers Boye-Jacobsen

Januar 2012

Indhold

Indhold **2**

1 Indledning **3**
1.1 Forord . . . 3
1.2 Problemformulering . . . 4
1.3 Virksomhedsbeskrivelse . . . 5

2 Metode **6**
2.1 Udviklingsmetode . . . 6
2.1.1 Anbefalet metode valg . . . 7
2.1.2 Scrum . . . 8
2.1.3 Tests . . . 9
2.2 Teknologier . . . 10
2.2.1 MVC . . . 10
2.2.2 Database . . . 10
2.2.3 Frontend . . . 11

3 Produkt **15**
3.1 Kravspecifikation . . . 15
3.2 Design . . . 17
3.2.1 Domænemodel . . . 17
3.2.2 Domænebeskrivelse . . . 18
3.2.3 Databasestruktur . . . 18

4 Projekt Forløb **21**
4.1 Product Backlog . . . 21
4.2 Sprint 1 . . . 21
4.2.1 Sprint backlog . . . 22
4.2.2 Burndown Chart . . . 22
4.2.3 Udførelse af Sprint 1 . . . 22
4.2.4 Retrospective . . . 27
4.3 Sprint 2 . . . 27
4.4 Sprint 3 . . . 27
4.5 Sprint 4 . . . 27

4.6 Sprint 5 27

5 Teori **28**
5.1 Patterns 28
5.1.1 Hvad er patterns 28
5.1.2 Formål 28
5.1.3 Forskellige patterns 28

A UserStories **I**

B Databasediagram **II**

Litteratur **III**

Kapitel 1

Indledning

1.1 Forord

Allerede på 1. semester stiftede vi bekendtskab med vores første patterns, først var det et søge pattern, men senere lærte vi også til singleton patternet. Samtidig blev vi undervist i brugen af Model-View-Controller mønsteret, der var et arkitekturpattern.

Under vores praktiktid på Equalsums blev vi stillet over for en opgave, hvor det var nødvendigt at bruge flere patterns, nogen kendte (singleton og observer) og også ukendte (f.eks. Repository-patter). Da vi blev stillet overfor opgaven til vores hovedopgave, indså vi, at vi blev nødt til at sætte os ind i flere patterns. Dette ledte frem til en interesse for patterns generelt.

Larman beskriver patterns med følgende ord:

> "Experienced OO developers (and other software developers) build up a repertoire of both general principles and idiomatic solutions that guide them in the creation of software."

[2, p.278]

Med andre ord, over tid samler udviklere en række generelle værktøjer der med ganske lette ændringer passer på mange forskellige problemstillinger - det er disse værktøjer der er kendt som patterns.

Vi tænkte derfor, at det kunne være interessant at se nærmere på patterns og hvordan de kan bruges i et stykke software. I denne rapport vil vi dels dokumnentere vores arbejde med at udvikle et webbaseret crm-værktøj til EqualSums. Samtidig vil vi, ved at tage udgangspunkt i dette software, kigge nærmere på de patterns vi bruger, og se på patterns generelt. Vores håb er, derved at få en

større forståelse for patterns, og hvordan de kan benyttes som "byggeklodser"i et stykke software.

Vi har fra Equalsums side fået til opgave at lave en webbaseret løsning, der kan håndtere kontakter fra før første kontakt, over mulige kunder til de er etablerede kunder. Der skal holdes styr på kontaktoplysninger, hvad man har foretaget sig af opkald, mails og møder og endelig skal man kunne planlægge aftaler med kunden. Kort sagt skal der konstureres et CRM-system.

Customer **R**elationship **M**anager

Samtidig skal produktet udvikles, så det er nemt at arbejde videre på for et andet team. Vi har indset, at vi på knap 10 uger ikke kan levere et kompelet CRM-system. Derfor vil vi fokusere på at udvikle i moduler der er prioriteret efter kundens (Equalsums) ønsker.

Rapporten vil blive delt op i 3 dele, første del, hvor vi beskriver vores arbejdsmetode. Andet afsnit, hvor vi dokumenterer produktet fra kravsspecifikation til færdigt produkt og endelig (men absolut ikke mindst) teori-afsnittet, hvor vi vil gå i dybden med forskellige patterns, hvad de kan, hvad de ikke kan, og hvorfor/hvorfor ikke vi har valgt at bruge dem.

Det hele vil blive rundet af med en konklusion, hvor vi opsummerer rapportens indhold, og de svar vi er kommet frem til gennem processen.

1.2 Problemformulering

I vores hovedopgave er den overordnede problemstilling:

> „At udvikle et webbaseret CRM-system der benytter patterns, og som er let at udvikle for tilkommende udviklere."

Det er klart, at det er et meget bredt problem, der skal afgrænses. Vi vil fokusere på patterns som det er relevant at anvende til den stillede opgave, gennemgå dem, forklare hvordan de virker generelt, herunder hvilke problemer de løser og endeligt hvordan vi benytter dem.

Under vores praktikperiode benyttede vi både singleton- og observer-pattern, men ud over disse to relativt simple patterns, benyttede vi også Repository-pattern, for at gøre det nemmere at bruge Mocks til test.

Når den endelige kravsspecifikation er på plads, vil vi få virksomheden til at prioritere user-storiesn'e. Herefter vil vi lave estimates og ud fra disse estimates vil vi vurdere hvilke user-stories det er realistisk at implementere i løbet af 4 sprints á to ugers varighed.

1.3 Virksomhedsbeskrivelse

Equalsums er en relativt lille virksomhed der blev stiftet af Mikkel Elliott og Martin Skov Kristensen. Virksomheden beskæftiger sig med udvikling af online softwareløsninger til bogholderi og bogføring. Udover Mikkel og Martin er der en fast udviklerstab på 4 backend-udviklere og én frontendudvikler. Udover den faste stab er der pt. en konsulent, tre praktikanter og en studentermedarbejder. Udviklerstaben styres af en projektleder, der sørger for den interne koordination i udviklergruppen.

Af øvrige ansatte er der tre sælgere hvoraf en også fungerer som administrativ medarbejder.

Ledelsen fremmer bevidst en uhøjtidelig tone i virksomheden, hvilket bl.a. ses ved, at hele virksomheden spiser frokost sammen, og oftest går en kortere tur herefter. Medarbejderne har også sækkestole og boksebold til rådighed, ligesom der også er kaffe ad libitum. Herudover er man til enhver tid velkommen til at medbringe kage.

Der er mere eller mindre frie mødetider, så længe man opnår 37 timer ugentligt, hvilket giver medarbejderne stor frihed til selv at vælge deres mødetider.

Den oprindelige idé var, at virksomheden skulle tilbyde en hel vifte af softwareløsninger. Da man for alvor kom i gang med den første løsning, blev det dog vurderet, at dette produkt i sig selv kunne holde hele virksomheden beskæftiget. Derfor skiftede fokus til dette enkelte produkt, der idag er det Equalsums udvikler.

Kapitel 2

Metode

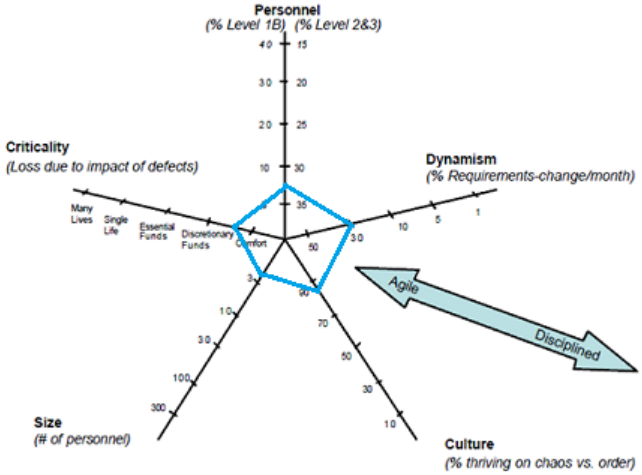
2.1 Udviklingsmetode

Her vil vi beskrive den udviklingsmetode vi har valgt at bruge til vores projekt og hvorfor vi ikke har valgt nogen af de andre. Som det første har vi undersøgt hvilken metode der vil passe bedst til vores projekt, og til det har vi gjort brug af Berry Boehm's stjernemodel (fig 2.1. Modellen bruges til at vurdere om man bør bruge en agil, plandreven eller noget midt i mellem til sit projekt. Det gør den ud fra 5 faktorer som man skal svare på i forhold til det projekt man skal til at starte på. Des tættere midten af diagrammet man sætter sine krydser, des mere en agil metode har man brug for. Det samme gør sig gældende den anden vej, hvor, jo længere man kommer ud af akserne, jo mere bør man gøre brug af en mere plandreven metode.

Personnel: På denne akse vises hvor erfarent et udviklerteam er. Da ingen i gruppen har nogen særlig stor erhvervsmæssig erfaring på trods af vi har lidt kendskab til teknologierne fra skolen er vores vurdering at den vil ligge mellem 0-35 og 10-30. Hvilket peger hen imod en agil metode.

Dynamism: Her bestemmes hvor dynamisk man har behov for at være. Efter de første par møder med vores virksomhed blev det ret hurtigt klart for os at der ville komme mange ændringer undervejs. Mange kunder har svært ved at sætte ord på hvad de egentlig gerne vil have til at starte med. Derudover er der flere forskellige vi snakker med som står for hvert deres område. Det betyder vi ligger omkring de 30 som peger på en rimelig agil metode.

Culture: Her ser man på hvordan teamet trives bedst. Skal der være regler og procedure for alle, eller har man mere frihed til selv at vælge. Vores team har det godt med at forholdsvist frit arbejdsmiljø hvor vi begge trives rigtig godt. Når vi ser på virksomheden hvor vi laver programmet er der også meget stor



Figur 2.1: Stjernemodellen

frihed og folkene i virksomheden trives godt herunder. Så det er kun naturligt for os at ligger os tæt på de 90 som igen peger på en agil metode.

Size: Her kigger man simpelt på hvor stort ens team er. Da vi arbejder i en lille to mandsgruppe er den nem at sætte på. Hvilket igen vil sige det ligger sig op af en agil metode. Store grupper kan blandt andet have svært ved at kommunikere og koordinere et stort projekt hvor imod et lille hold nemmere kan styre hvem der laver hvad.

Criticality: Her kigger man på hvor store konsekvenser fejl har i systemet. I vores system kommer fejl ikke til at have den store betydning. De kommer hverken til at koste menneskeliv eller vanvittige økonomiske nedture for firmaet. Derfor ligger vi os ind mod midten igen og dermed en agil metode.

2.1.1 Anbefalet metode valg

Ud fra vores analyse på forgående side ved hjælp af Berry Boehm's stjernemodel, er det tydligt at se vi ligger os op af en agil metode. Som metode har vi valgt at bruge Scrum. Vi har allerede erfaring med Scrum fra vores andre projekter samt fra vores praktikophold. Vi syntes derfor den er et oplagt valg vores projekt. Dog vil vi ikke kun benytte Scrum da vi mener en del af de agile metoder handler om at fintune dem så de passer perfekt til netop dit projekt. Vi har derfor valgt at tage nogle rutiner med fra nogen af de andre metoder. Fra UP har vi valgt at gøre brug af Domæne modellen, samt vores database diagram. Vi syntes det giver os et godt fundament, især som mindre erfarne. Derudover har vi også lånt værktøjerne, refactoring og testing som de primære fra XP, men også pair programmering, fælles kode og kode standarder vil blive brugt. En af

grundene til vi ikke har valgt at arbejde ud fra en UP metode, er blandt andet Berry Boehm's stjernemodel, men i høj grad også vores korte tidsfrist. Da vi har et meget begrænset tidsrum til at få lavet programmet har vi simpelthen ikke tid til at analysere så meget. Derudover ligger det også klart at der kommer mange udvidelser til systemet da vi igen på grund af den begrænset tid er blevet nødsaget til at afgrænse vores projekt væsentligt. Det vil sige der er mange opgaver vi slet ikke har med i dette oplæg men som skal laves bagefter.

2.1.2 Scrum

Scrum er en agil udviklingsmetode skabt tilbage i 90'eren. Det er et interaktivt og inkremental framework som hurtigt er blevet meget populært blandt udviklere, da det giver en stor frihed, men samtidig sætter nogle fast rammer for arbejdsgangen.

Roller

I Scrum vil man støde på rollerne Scrum Master, Produkt Owner og Scrum Team. Vores produkt Owner hedder Martin Kristensen og er salgschef hos Equalsums. Det er ham der står for projektet sammen med os, så det var naturligt at give ham den rolle, som blandt andet indebære at prioritere user stories og stå til rådighed når der er spørgsmål omkring produktet. Som Scrum master har vi valgt Peter Thomsen. Valget faldt på Peter fordi han havde erfaring med programmet ScrumDesk som vi bruger til at styre vores backlog, sprints, osv. Gregers Boye-Jacobsen er en del af selve Scrum teamet som udvikler. Peter indgår selvfølgelig også som en del af teamet og udvikler sammen med Gregers.

Strukturen

Vi arbejder med 14 dags(10 arbejdsdage) sprints hvor vi har $2\frac{1}{2}$ arbejdsdag til udvikling på systemet. Grunden til vi ikke har mere er selvfølgelig rapport skrivning. Vi har en velocity på 0,7 hvilket betyder vi har 53,2 effektive timer til hvert sprint.

Da vi altid sidder ude i virksomheden når vi udvikler på produktet, indgår vi også i deres Scrum holds dagligt Scrum møder, og hvis der ellers er nogen fælles møder. Internt i vi vores eget projekt holder vi også et dagligt Scrum møde, hvor vi kort fortæller hvad vi har lavet og hvad vi har tænkt os at gå i gang med næste gang. Derudover holder vi også et sprint review møde med Martin (produktowneren) hvor vi viser det frem vi har lavet i sprintet. Det gør vi for at sikre at det vi har lavet lever op til Martins forventninger. Scrum Retrospective holder vi løbende dels fordi vi kun er to på holdet og hele tiden snakker sammen under forløbet, og dels fordi man har en tendens til at glemme hvad der gik dårlig og især hvad gik godt.

2.1.3 Tests

For at vores system lever op til vores krav personlige krav om god kodeetik og standard har vi valgt at bruge en forskellig række test. Det er med til at sikre en høj kvalitet i produktet, og er selvfølgelig med til at test om vores system gør som det skal. På den måde får virksomheden også et produkt der er testet igennem, og i højere kvalitet.

Unittests

Vi har valgt at bruge Unit test til at teste dele af vores program. Unit tests virker på den måde at man tester så simpel en del af programmet om muligt. Det vil ofte være en enkelt metode af gangen. Det gøres for at sikre at være lille del af programmet virker korrekt. For at kunne gøre dette uafhængigt af andre metoder og for at få de samme test resultater hver gang laver man nogle fake data, kaldet mocks. En af de store fordele ved at lave unit tests er at man kan refactor sin kode, og hurtigt teste om den stadig virker som den skal. Med det metodevalg vi har valgt passer netop dette perfekt.

Ofte vil man høre om test-driven udvikling især i forbindelse med agile metoder og XP bruger det som en af deres store værdier. Det vil sige man skriver sine tests først, og derefter laver man en simpel metode der kan få denne tests til at "pass". Man kan også vælge at teste eksisterende kode for at sikre sig den gør som man den skal, og hvis man skulle ændre i koden senere vil det igen være nemt at teste.

Vi har valgt ikke at bruge test-driven udvikling da det kræver en del erfaring at skrive tests som vi ikke har endnu. Da vi har en stram tidsramme har vi valgt kun at teste dele af vores program, fordi det andet vil tage for lang tid for os.

Integrationstests

Vi har også valgt at benytte Integrations tests til vores projekt. En integration test minder lidt om en unit test bortset fra den tester flere dele af systemet på en gang. Derudover bruger den også rigtige data, man skal altså ikke mocke data. Ved at de tester flere dele af programmet sikre man sig at de forskellige dele af programmet arbejder rigtigt sammen. Vi har valgt at bruge integrations tests for at kunne teste vores framework ordenligt for at sikre vores program gør det rigtige igennem de forskellige lag.

Da vi laver meget sortering via LINQ i vores db-lag er det vigtig for os at kunne teste på om vi får det præcis ønsket resultat tilbage, og om det bliver vist ordenligt i vores GUI.

Acceptancetests

Vi har valgt at køre User Acceptance test på vores system. Det vil sige vi vil have vores egentlig bruger til at teste systemet, da de ved bedst hvordan det præcist skal være. Vi forstiller os Martin (Produkt owneren) skal systemet for fejl. Vi vælger at teste det på denne måde for at sikre os kundens tilfredshed mens det hjælper os med at finde de fejl der måtte være i systemet. Som udvikler på et system, har man stor indsigt i hvordan systemet er bygget op og har en tendens til at klikke udenom fejlen. Det dur ikke da det netop går ud på at finde fejl.

2.2 Teknologier

Når der skal udvikles web-applikationer, er der utroligt mange valg at træffe rent teknologimæssigt. Der skal træffes valg for både backend, frontend, database og alt derimellem. Hvis der skal være fancy animationer skal der så bruges html5 eller flash - eller silverlight? Skal backenden være php eller asp.net? Skal databasen være en postgresql, mssql eller mysql-database? Hvordan snakker backend og frontend sammen? xml, json eller noget helt tredje? Der er mange muligheder og i det følgende kapitel bliver de valg vi har truffet forklaret og begrundet.

2.2.1 MVC

Overordnet set har vi valgt at arbejde med *Microsofts ASP.net mvc3 framework*. Dette giver en logisk opbygning i projektet, der bliver delt op i *Model*, *View* og *Controller*. Udover, at det holder sig tæt op af den 3-lags arkitektur som vi har brugt i uddannelsen, giver det også øget overblik. Samtidig giver det mulighed for at bruge *Microsofts Razor View Engine* (Herefter blot Razor) der i forhold til tidligere løsninger som f.eks. web forms giver øget læsbarhed og enklere syntaks.

2.2.2 Database

EntityFramework

I Equalsums bruges ADO.NET entity framework (Herefter EF) til at forestå kommunikation mellem database og backend. Da EF ud fra databasen selv konstruerer modeller er det en væsentlig forenkling i forhold til at bruge sql-sætninger. I al sin enkelthed fortæller man EF, at man f.eks. gerne vil have fat i en kunde-objekt fra databasen, og herefter vil EF selv finde ud af, hvilke relationer der findes i databasen, og sørge for at hente al relevant data ud. Det samme gør sig gældende ved indsætning, hvor man laver et objekt med de

ønskede attributter, og herefter fortæller EF at det skal gemmes i databasen, hvorefter de enkelte værdier bliver indsat i de forskellige tabeller med de rette relationer.

LINQ

LINQ er som sådan ikke specifikt til brug for databaser, men da det er til at lave vores database-queries vi bruger LINQ, vil det blive nævnt i database-afsnittet.

Language **IN**tegrated **Q**uery

LINQ udmærker sig ved at lave sql-lignende kald. Forskellen er, at man kan bruge LINQ på andre datakilder end databaser, f.eks. arrays. LINQ gør brug af *anonyme typer* og *lambda-expressions* - disse er typer og funktioner der oprettes ad hoc, det vil sige, at de kun eksisterer så længe de bruges, og man undgår derved, at oprette en mængde klasser og metoder. I det følgende eksempel vises et linq-udtryk, hvor man i en graph vil have alle ikke-besøgte nodes:

```
1 List<Node> unvisitedNoteList = (select from n in nodeList
2                                 where n.visited == false
3                                 select n).toList();
```

Hvis man ikke havde brugt toList() ville man have fået en IEnumerable¡Node¿

2.2.3 Frontend

Det har altid været forbundet med stort besvær at udvikle hjemmesider der ser ens ud, og opfører sig ens i alle browsere. Dette skyldes at der findes en lang række standarter, og at alle browsere understøtter disse standarter i større eller mindre grad. Det har derfor været nødvendigt at benytte sig af workarounds og hacks, der udnytter de forskellige browseres svagheder og styrker.

I vores arbejde slipper vi for denne problemstilling, da systemet der udvikles, kun skal bruges internt. Eftersom EqualSums udelukkende bruger Google Chrome, er det kun nødvendigt at udvikle til en enkelt browser. I det efterfølgende vil brugen af forskellige front-end teknologier blive forklaret.

RazorViewEngine

Som tidligere nævnt bruger vi Microsofts Razor View Engine, der er en ny view engine til asp.net. Fordelen ved at bruge Razor er, at der er en kraftigt forenklet syntaks. Samtidig er Razor smart nok til, selv at vide, hvornår der er tale om et @der starter en blok, og hvornår det er @ i en e-mail adresse. Dette betyder også, at man ikke behøver markere når en razor-blok afsluttes, til forskel fra asp hvor man bruger %> og php hvor man bruger ?>. Når man starter en Razor kodeblok skriver man ganske enkelt @ efterfulgt at koden. Det er her vigtigt at huske, at der ikke skal semikolon efter blokken. Brugen af Type inference gør

dog, at Razor selv kan vurdere om det er en kodeblok der kommer, eller om man bare skriver en mail-adresse.

Dette øger læsbarheden af koden, og hastigheden hvormed man udvikler, da det er langt mere intuitivt.

jQuery

jQuery er et framework til javascript der virker over alle moderne browsere, og man skal som udvikler ikke bekymre sig om browser kompatibilitet. Grundideen bag jQuery er "Write less, do more". Som eksempel kan nævnes, at man har mulighed for at bruge css-selectorer (og pseudoselectorer) til at vælge elementer. Man slipper derfor for at bruge document.getElementById og andre mærkelige konstruktioner, da man kan gribe direkte ned i DOM-træet. Man kan f.eks. vælge den første række i tabellen med id "customers"ved at skrive

```
1 \$("\#customers:first-child")
```

Udover det er meget nemmer at vælge et bestemt element, er det også nemt at manipulere elementerne hvadenten man vil ændre, tilføje eller fjerne indhold, tekst, css-egenskaber eller værdier.

Denne fleksibilitet sammen med en lang række indbyggede funktioner til animationer gør jQuery til et virkelig effektivt redskab, når der skal bruges javascript på en side. Desuden gør jQuerys popularitet, at der findes en lang række plugins, som tilføjer næsten enhver funktionalitet man kan tænke sig, her kan især nævnes jQuery UI (http://jqueryui.com/)

AJAX

I forbindelse med kommunikation mellem front-end og backend vil vi i høj grad benytte AJAX, som muliggør asynkron kommunikation mellem klient og server. Dette betyder, at der kan hentes og sendes data uden, at siden skal refreshes. På den måde får brugeren en langt smidigere oplevelse, end hvis der skulle refreshes hver gang klienten skulle sende eller forespørge data fra serveren.

Asynchronous
Javascript
And
XML

Et sted hvor man bruger AJAX, er ved autofuldførelse, som f.eks. kendt fra google-søgninger. Har man en debugger der overvåger serverrequests, som f.eks. firebug mens man skriver sin søgning, vil man se, at der bliver sendt en request for hvert tastetryk. Dette er AJAX-requests der bliver sendt.

Netop brugen af jQuery gør det langt nemmere at benytte sig af asynkrone kald, idet der i jQuery findes en indbygget funktion til at sende asynkrone forespørgsler til serveren, der forenkler processen, som det ses i figur 2.2

Figur 2.2: Asynkront jQuery-kald

```
var data = {userId: 4}
var url = {"CRMsystem/feedDetails/getUserName"}

$.getJSON{url, data, function(data){
  $("#customerName").text(data);
}
```

JSON

Java**S**cript **O**bject **N**otation

Vi har dog valgt at benytte JSON istedet for XML som dataformat til dataudveksling. Først og fremmest skyldes dette, at JSON er lavet specielt til Javascript, men samtidig er der langt mindre data der skal sendes frem og tilbage, da data ikke skal markeres på samme måde som xml hvorved der er langt mindre overhead:

Figur 2.3: XML-eksempel

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person>
    <firstName>Thomas</firstName>
    <lastName>Hansen</lastName>
    <address>Guldvej 23</address>
    <city>Aalborg</city>
    <zipCode>9000</zipCode>
    <email>thomas@gmail.com</email>
  </person>
</persons>
```

Figur 2.4: JSON-eksempel

```
person[
  {
    firstName: "Thomas",
    lastName:"Hansen",
    address: "Guldvej 23",
    city: "Aalborg",
    zipCode: 9000,
    email:"thomas@gmail.com"
  }
]
```

Som det ses af figur 2.3 og 2.4 er json-objekter langt enklere end xml-objekter, i det viste eksempel er der tale om et array (omgivet af []) med et enkelt objekt. Et objekt er omkranset af og .

CSS3

Brugen af CSS3 giver en lang række muligheder for at lave langt mere elegant design, her kan bl.a. nævnes brug af gradienter og afrundede hjørner. CSS3 er endnu ikke understøttet af alle browsere, og ved visse egenskaber (som f.eks. border-radius), skal der benyttes en egenskab for hver af de 3 browser-engines (webkit, mozilla og IE). Som tidligere nævnt bliver der ikke problemer med cross-browser kompatibilitet.

Kapitel 3

Produkt

3.1 Kravspecifikation

EqualSums har stille os en opgave der omhandler udvikling af et CRM system. Det de primært var interesseret i at få udviklet var deres håndtering af kunder. Hvor fokus ville være den måde salgsafdelingen håndtere kunderne på. Da det kun var en lille del af programmet vi skulle lave, blev der stillet krav om at programmet nemt skulle kunne udvides og vedligeholdes. Selve produktet skulle kunne vise og håndtere alle firmaets kunder, og potentielle kunder. Der skal kunne laves aftaler med de forskellige leads, som også bliver noteret i sælgernes Google-kalender. Derudover skal der være mulighed for at skrive noter på en lang række af de data der er i systemet. Der blev også stillet krav om at kunne sende mails direkte fra programmet, samt hente gamle mails frem. Det de i bund og grund ønskede, var en effektiviseret kopi af det Excel ark de bruger nu, hvor man samtidig har styr over hvad man præcist har aftalt og har haft aftalt med kunderne.

På "Min Side"siden ønskes følgende features:

- Der skal være en oversigt over dagens opgaver, for at gøre det hurtigt og nemt at overskue de opgaver man har for dagen
- Mulighed for at fjerne en opgave når den er løst, så man ikke længere skal se og tænke på den
- Medarbejderens kalender skal vises på siden så man hurtigt og nemt kan se hvilken aftaler man har i løbet af måneden, og dermed også nemmere kan planlægge nye aftaler
- Man skal kunne se detaljer for dagens opgaver så man kan få en mere præcis beskrivelse af hvad der skal laves

- Sortere dagens så man bedre kan styre hvilken rækkefølge man vil lave dem i

På "Leads Oversigt"siden ønskes følgende features:

- Der skal være en oversigt over alle Leads, så man nemt kan finde de leads man har i systemet og læse deres vigtigste data
- Mulighed for at sortere leads efter de viste data så man kan få en liste over præcis de leads man ønsker at arbejde med
- Man skal kunne opdatere data på leads, så de stemmer overens med virkeligheden
- Der skal være mulighed for at afmærke flere leads og opdatere samme felt på dem på én gang, så man slipper for at skulle rette det samme data på mange forskellige leads
- Der skal kunne uploades en CSV fil, med en masse leads så man slipper for at skulle indtaste disse en ad gangen. Der skal samtidig tjekkes for dubletter så man ikke får de samme leads flere gange
- Der skal være en blank plads i bunden så man manuelt kan oprette et enkelt lead
- Skal kunne tilføje en ny kolonne dynamisk i oversigten, så man dynamisk kan tilføje flere data at sortere efter

På "Lead Detaljer"siden ønskes følgende features:

- Man skal kunne sortere efter lead status, samt ES-medarbejder så man nemt kan få en præcis liste af leads man vil se dejtalerne på
- Se alle stamdata på et lead, samt en liste af dets kontakter(Kontaktpersoner) så man kan få en dejtaleret information omkring Leadet.
- Man skal kunne se stamdata på en valgt kontakt indenfor det valgte lead, så man nemt kan se de oplysninger der er gemt omkring kontakten
- Man skal kunne rette de stam data der er på både Lead og kontakt, så de stemmer overens med virkeligheden
- Der skal være mulighed for at kunne oprette en ny kontakt på leadet, så man har en mere mulighed for at kontakte firmaet
- Upload CSV fil med kontakter til et bestemt lead, så man ikke skal indtaste dem manuelt, her tjekkes også for dubletter så man ikke oprette den samme kontakt to gange
- Der skal kunne oprettes noter på leads og kontakter, så man senere kan slå op og læse noten omkring leadet eller kontakten. Derudover skal der også være noter på de Actions(aftaler) man laver med kontakten

- Finde gamle noter frem på både leads og kontakter, så man kan se hvad der er blevet skrevet omkring leadet eller kontakten før i tiden
- Oprette en aftale med et lead eller kontakt(aftale kan være møder, telefon opkald, kurser osv.). aftaleren oprettes også i medarbejderens Google-kalender
- Der skal kunne oprettes noter på aftaler, hvis man skal huske noget specielt omkring aftalen kan man skrive en note omkring det så man er sikker på at huske det
- Find gamle aftaler frem igen samt tilhørende noter, så kan man hurtigt se hvad man egentlig har lovet og aftalt med en kunde, især hvis der opstår uenighed omkring aftaler
- Skal kunne sende en e-mail til den valgte kontakt, samt vedhæft dokumenter i denne. Så slipper man for at skulle åbne et 3. parts program for at skrive en e-mail
- Man skal også kunne vedhæfte filer til sin e-mail
- Der skal være en historik over sendt og modtaget post mellem ES-medarbejderen og kontakten, så man igen kan finde historikken frem og bekrafte hvad der egentlig er blevet aftalt

3.2 Design

3.2.1 Domænemodel

Vi har lavet en domænemodel for at give et bedre overblik over det problemområde vores projekt omhandler. Modellen hjælper os og andre med at forstå hvordan systemet er bygget op. Den viser sammenhængen mellem de forskellige klasser. Navnene på klasserne fortæller hvad vi har med at gøre, mens attributterne fortæller noget om klassens egenskaber.

Det kan ses på modellen at en Employee ikke har direkte forbindelse til Contact, men i stedet har forbindelsen til Company. Dette er fordi de altid vil have alle firmaets kontakter. Action og Course er sat til Contact i stedet for Company for at en ansvarlig kontaktperson til netop den Action eller Kursus. Derudover er Paricipants sat ind for sig selv da det kan være ansatte i firmaet som man ellers ikke har nogen kontakt til. Support-System er resten af det system der blev lavet under vores praktikophold. Modellen bag det har vi valgt at lave på denne måde for at holde de to projekter adskilt.

3.2.2 Domænebeskrivelse

Her vil de forskellige klasser og deres associationer kort blive forklaret. Først beskrives klassen, og derefter forklares hvordan associationerne hører til denne klasse.

Action Svarer til en aftale der er aftalt mellem en employee og contact. Kunne være et møde.
Contact: Der kan være tilknyttet en eller flere kontakter til en action. Der vil altid mindst deltage 1 men nogen gange flere.
Employee: Her vil der også være mulighed for at der er flere der deltager. Igen vil der ikke være en action uden en Employee.

Notes En Note er en lille besked tilknyttet de klasser den har asscociationer til. Meget simpel klasse.

Employee Indeholder de vigtige informationer der kræves af en medarbejder i virksomheden.
Company: Viser hvilken medarbejder der ejer dette firma(Lead). En ansat kan eje flere virksomheder, men ikke eje et som allerede er ejet af en anden.
Course: Viser om denne medarbejder deltager i dette kursus.

Company Company indeholder stamdata omkring firmaet(Leadet)
Contacts: Viser hvilken kontakter der tilknyttet dette firma. Der kan være flere kontakter men de vil naturligt kun være tilknyttet et firma.
Employee: Viser hvilken ansat der ejer virksomheden. Her vil altid kun være en ejer.

Contact Indeholder de vigtige informationer omkring en kontakt.
Company: Viser hvilket firma kontakten er en del af.
Course: Viser om denne kontakt har forbindelse til et kursus.

Course Indeholder data omkring et kursus.
Contact: Viser hvilke kontakter der er tilknyttet dette kursus. Ofte vil det være en hvor resten vil være deltagere.
Employee: Viser hvilke medarbejdere der er tilknyttet dette kursus.
Participants: Viser hvilke kursister der er tilknyttet dette kursus.

Participants Virker som en klasseliste for et kursus. En liste over deltagere som ikke nødvendigvis er kontakter.
Course: Viser hvilket kursus listen er tilknyttet. Der vil kun være et kursus pr. liste samt der selvfølgelig kan deltage mange til hvert kursus.

3.2.3 Databasestruktur

Database designet er lavet ud fra domæne modellen. Vi har forsøgt at holde tabellerne så simple som muligt, det vil sige vi ikke blander data sammen.

Derudover forsøger vi at undgå null værdier samt redundans. Sidst vil vi følge de 3 første normalformer for at give os et bedre design, hvor vi minimere redundans og opdater, indsæt og slet uregelmæssigheder.

Som det ses på domæne modellen er der flere steder mange-til-mange relationer. For at styre dette i database sammenhæng sættes en tabel ind imellem. Her vil de to fremmed nøgler til sammen lave en primær nøgle. Det kan blandt andet ses ved Employee og Action. Her vil der blive sat en ny tabel ind, som indeholder de to fremmed nøgler, som dens primære nøgle.

De steder hvor der er 1-til-mange relationer vil der blive sat en fremmednøgle på den tabel der er mange af. Eksempelvis vil Company have mange kontakter, der sættes en fremmednøgle på kontakten så den altid kan se hvilket firma den hører til. Hvis der havde været 1-til-1 relationer vælges en af tabellerne og lægger en fremmed nøgle på den.

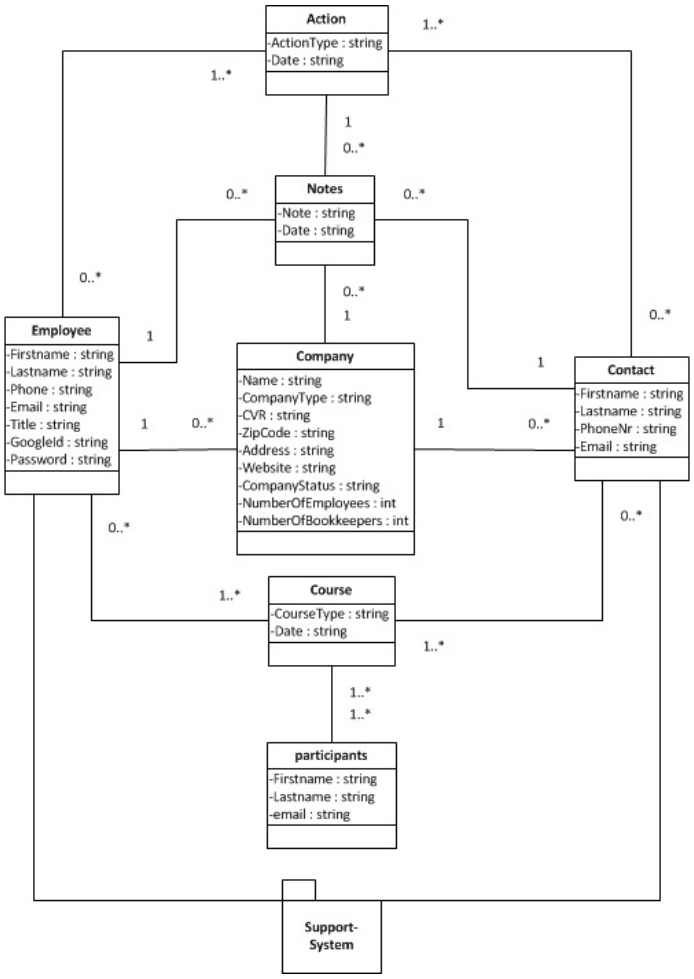
1. NF kan blandt andet ses i vores Kontakt tabel hvor navnet på kontakten er delt op i fornavn og efternavn, altså atomare data.

2. NF kan ses flere steder i database da vi ikke har nogen attributter der er afhængig af dele af en nøgle.

3. NF ses i det klassiske eksempel med post nummer og by. Da disse er afhængige af hinanden, og den primære nøgle flytter vi dem ud i en tabel for sig selv. [1, p.348-361]

Beskrivelse af databasemodel

Vi vil her beskrive sammenhængen mellem nogen af de tabeller hvor man ikke umiddelbart kan se hvad der menes. Company: Denne tabel har flere tabeller knyttet til sig som kun indeholder en primær nøgle samt en enkelt attribut.(CompanyStatus, CycleState og CompanyTypes) Vi har lagt dem ud i tabeller for sig, fordi vi skal kunne søge og sortere på dem. Det er lister med fastsatte værdier som kunden skal kunne vælge ud fra. Titles: Titles har to mange til mange tabeller, det er fordi en Employee kan have flere titler, samt mange titler kan have adgang til de samme steder i programmet. Det bruges til at styre hvor de forskellige Employees har adgang i programmet. Notes: For at undgå at få en masse null værdier har vi valgt at oprette en række nye tabeller som ligner det man gør ved mange til mange relationer. Her skal man dog være opmærksom på det faktisk kun er en, en til mange relation. ValueList: Kunden vil være i stand til at oprette dynamiske kolonner af bestemte typer. Derfor har vi lavet denne tabel som indeholder data samt en fremmed nøgle til infoList som fortæller hvilken type data vi skal sorterer efter og hvad kolonnens navn er. Disse dynamiske kolonner er ens for alle Companies.



Figur 3.1: Domænemodel

Kapitel 4

Projekt Forløb

Her vil blive forklaret hvordan projektet er planlagt, og gå i mere i detaljer omkring de enkelt spændende områder i hvert sprint. Sprintet vil kort blive gennemået hvorefter der vil blive kigget nærmere på nogle kodeeksempler.

I de 5 sprints som forløbet strækker sig over, er hvert sprint delt op i 2 uger hvor af ca. $2\frac{1}{2}$ dag om ugen går med programmering mens de sidste uger af forløbet går over i ren rapportskrivning. Denne struktur er valgt, for at få skrevet lidt ned, mens det man har lavet stadig er nyt og friskt i hukommelsen. Ved ikke at lave for lange sprints får man oftere feedback på produktet hvilket betyder, at man hurtigere kan rette til når eller hvis der kommer ændringer.

4.1 Product Backlog

Bla bla bla Bla bla bla Bla bla bla Bla bla bla Bla bla bla Og et billed af backloggen.

4.2 Sprint 1

I det første sprint vil der altid gå tid med at få sat arbejdsmiljø op, men da der skulle arbejdes videre på et projekt som allerede havde sat op, var meget af dette allerede på plads. Der skulle dog laves lidt da der skulle omstruktureres på opbygning af projektet ved hjælp af Areas i MVC 3. Derudover blev der også en del refactoring af gamle klasser fordi der blev lavet en hel del om i database designet, så systemet kunne understøtte al den nye funktionalitet.

4.2.1 Sprint backlog

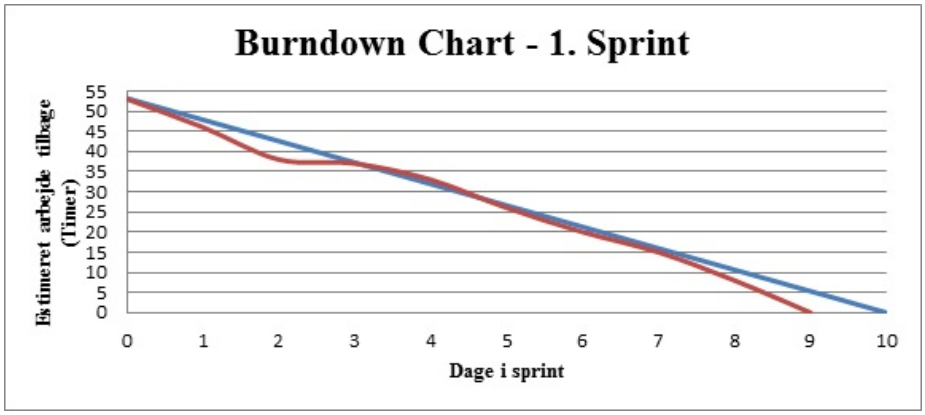
Der blev ikke sat så mange user stories ind i dette sprint for få en god buffer til uventet problemer med arkitekturen og opstart af projektet. Der var også lidt usikkerhed omkring hvor meget der kunne laves på et sprint. Kolonnen "Effort"er det antal timer der er givet hver User Story. Der kan maximalt klares 53,2 i et sprint, og det viste sig at passe ret godt i første sprint.

Subject	Story Type	Description	Importance	Effort
ListeView sorterSubject: ListeView sorteringing	User Story	Som Sælger vil jeg kunne Sortere og filtrere data i listen over firmaer/leads si	100	20
Upload data	User Story	Som Sælger vil jeg kunne uploade hele dokumenter (cvs) med leads, så jeg	100	14
Klare gøre arbejdsmiljø	Technical Story	Gøre Visualstudio projektet klar til at begynde arbejdet.	100	8

Figur 4.1: Sprint 1 backlog

4.2.2 Burndown Chart

Som burndown chartet viser, blev de første opgaver løst hurtigere end forventet, men det udlignede sig senere. Bufferen som bevidst var blevet sat ind i dette sprint for at få en god start, blev der heldigvis ikke brug for. Det ses også i form af, at alle opgaver var løst en dag før sprintets deadline.



Figur 4.2: Sprint 1 Burndown Chart

4.2.3 Udførelse af Sprint 1

Vi startede med at gøre arbejdsmiljøet klar til kunne håndtere den del af programmet der skulle håndtere salgsafdelingens område. MVC 3 er heldigvis utrolig fleksibelt og vedligeholdsvenligt så det gik nemt. Der opstod ikke nogen særlige problemer med hensyn til opsættelse af arbejdsmiljøet.

User story - Upload data

„Som Sælger vil jeg kunne uploade hele dokumenter (cvs) med leads, så jeg kan tilføje mange kontakter på én gang. Så jeg hurtigt kan importere mange kontakter i stedet for at taste dem enkeltvis.“

Som user story'en beskrive skal der kunne uploades en kommasepareret fil med leads, som skal skrives ind i databasen. Her skal der selvfølgelig tjekkes om der er dubletter, så der ikke uploades de samme leads flere gange, eller får fejl fordi man prøver at oprette flere leads med samme cvr-nummer som er gjort unik i databasen.

Det er blevet lavet så man som bruger skal vælge den CSV fil man vil uploade. Her er det vigtigt at dataet står i den rigtige rækkefølge fordi metoden ikke kan skelne mellem hvad data det er den læser.

```
1  [HttpPost]
2  public ActionResult UploadCSV()
3  {
4  List<List<CSVCompanyWrap>> tempList = new List<List<CSVCompanyWrap
       >>();
5              CSVList cvsList = new CSVList();
6              try
7              {
8                  HttpPostedFileBase file = Request.Files["
                       uploadCSVContacts"];
9                  if (file != null && file.ContentLength > 0)
10                 {
11                     string fileName = file.FileName;
12                     string path = System.IO.Path.Combine(Server.
                           MapPath("~/Uploads"), fileName);
13                     file.SaveAs(path);
14
15                     tempList = CSVReader(path);
16
17                     cvsList.Companies = tempList[0];
18                     cvsList.Doubles = tempList[1];
19                 }
20             }
21             catch(Exception e)
22             {...}
23
24             return View(cvsList);
25         }
```

Det første metoden gør, er at tjekke om filen overhovedet er blevet uploadet korrekt. Såfremt alt går godt, bliver filen kopieret over på serveren. Herefter bliver stien til filen sendt videre til metoden CSVReader(), som er metoden der læser filen.

CSVCompanyWrap er en wrapper klasse til company objekter da der skal være mulighed for at kunne vælge hvilken leads der skal sorteres fra ved hjælp af nogle checkboxes. Det er to simple Properties, en boolean til at styre om checkboxen er markeret eller ej(true/false) og en property til det givne Company objekt.

CSVList indeholder to lister med CSVCompanyWrap objekter, dette er også den model vores View forventer at modtage. De to lister svare til en række leads og en række dubletter.

```
1  private List<List<CSVCompanyWrap>> CSVReader(string filePath)
2  {
3    List<List<CSVCompanyWrap>> fullList = new List<List<
         CSVCompanyWrap>>();
4    List<CSVCompanyWrap> companies = new List<CSVCompanyWrap>();
5    List<CSVCompanyWrap> dublets = new List<CSVCompanyWrap>();
6    try
7    {
8      using (StreamReader reader = new StreamReader(filePath))
9      {
10       string line;
11       while ((line = reader.ReadLine()) != null)
12       {
13         string[] lines = line.Split(';');
14         short number;
15         CSVCompanyWrap cw = new CSVCompanyWrap()
16         {
17           Company = new Company()
18           {
19             ...
20           },
21           Choosen = false
22         };
23
24         if (companies.Where(x => x.Company.Cvr.Equals(cw.Company.
               Cvr)).FirstOrDefault() != null)
25         {
26           dublets.Add(cw);
27           dublets.Add(companies.Where(x => x.Company.Cvr.Equals(cw.
                 Company.Cvr)).Single());
28           companies.Remove(companies.Where(x => x.Company.Cvr.
                 Equals(cw.Company.Cvr)).Single());
29         }
30         else if (dublets.Where(c => c.Company.Cvr.Equals(cw.Company
               .Cvr)).FirstOrDefault() != null)
31         {
32           dublets.Add(cw);
33         }
34         else
35         {
36           companies.Add(cw);
37         }
38       }
39     }
40   }
41   catch (Exception e)
42   {...}
43   fullList.Add(companies);
44   fullList.Add(dublets);
45   return fullList;
46 }
```

Metoden her læser den uploadede CSV fil og splitter den op hver gang den støder på et ";". For hver linje i filen bliver der lavet et CSVCompanyWarp objekt. Det objekt tjekkes der så på om eksistere i en af de to lister, companies eller dubletter. Hvis der findes en dublet i companies listen fjernes de begge og flyttes ind i listen med dubletter. Til sidst sendes de to lister tilbage hvor de bliver vist for brugeren, som herefter skal tage stilling til hvilke af dubletterne han vil bruge.

```
[HttpPost]
public ActionResult SortDublets(CSVList csv)
{
    CSVList cvsList = new CSVList();

    foreach(var item in csv.Doubles)
    {
        if(item.Choosen)
        {
            item.Choosen = false;
            csv.Companies.Add(item);
        }
    }
    cvsList.Companies = csv.Companies;

    return View("ConfirmCSV", cvsList);
}
```

En simpel metode der flytter dubletter over i companies listen ud fra hvilken lead man valgte på hjemmesiden. Herefter bliver man flytter over til en ny side hvor man skal bekræfte en sidste gang om man har valgt de rigtige leads. Er dette tilfældet, kan man uploade sine data til database hvor der igen vil blive undersøgt om der er dubletter. Hvis man skulle vælge et lead som man ikke vil have med, kan man vælge den fra her.

```
public ActionResult SubmitCSV(CSVList csv)
{
    List<CSVCompanyWrap> tempList = csv.Companies;

    foreach(var item in tempList.ToList())
    {
        if (item.Choosen)
        {
            csv.Companies.Remove(item);
        }
    }
    csv.Doubles = new LeadDetailsController(_db).
        LeadDetailsRepository.AddCompaniesFromCSV(csv);

    return View("ChooseDbDoubles", csv);
}
```

Her løber metoden listen igennem for at se om brugeren skulle have fravalgt nogen leads, hvorefter de vil blive fjernet af listen. Til sidst sendes listen til repositoriet hvor de vil blive tilføjet til databasen.

```
public List<CSVCompanyWrap> AddCompaniesFromCSV(CSVList csv)
```

```
2  {
3      List<CSVCompanyWrap> companyDoubles = new List<CSVCompanyWrap
           >();
4
5      foreach(var item in csv.Companies.ToList())
6      {
7          CSVCompanyWrap comp = new CSVCompanyWrap()
8          {
9               Company = _db.Companies.Where(x => x.Cvr.Equals(item.
                    Company.Cvr)).SingleOrDefault(),
10                  Choosen = false
11         };
12
13         if (comp.Company != null)
14         {
15             companyDoubles.Add(item);
16             companyDoubles.Add(comp);
17         }
18         else
19         {
20              _db.Companies.AddObject(item.Company);
21              _db.SaveChanges();
22         }
23     }
24
25     return companyDoubles;
26 }
```

Inden de endeligt bliver tilføjet til databasen, undersøges listen om der også er dubletter i databasen. Det gøres ved at trække et Company objekt ud fra database via CVR nummeret i den sorteret CSV liste. CVR-nummeret bruges fordi det er unikt og derfor ikke indeholder nulls eller ens værdier. SingleOrDefault() returnere enten et objekt eller null. Det vil sige, hvis metoden finder et objekt vil if-sætning være true og så vil de to dubletter blive skrevet i en liste som bliver sendt tilbage til brugeren. Hvis den ikke finder nogen objekter vil leadet fra CSV dokumentet bliver skrevet i databasen med det samme.

De dubletter der bliver fundet i CSV dokumentet og databasen skal sorteres på samme måde som tidligere af brugeren. herefter bliver de sendt ned til repositoriet og overskriver objektet i databasen.

User Story - ListeView sortering

„Som Sælger vil jeg kunne Sortere og filtrere data i listen over firmaer/leads så jeg kan få leads frem i den rækkefølge jeg ønsker“

bla bla bla bla bla bla bla blabla bla bla blabla bla bla blabla bla bla blabla bla bla blabla bla bla blabla bla bla blabla bla bla bla

4.2.4 Retrospective

Det første sprint gik utroligt godt, vi fik overraskende få problemer med at klargøre miljøet, hvor vi ellers havde regnet med der kunne opstå flere problemer. Vi fik og se en af MVC's stærke sider, ved at man nemt kan vedligeholde det, og tilføje store ændringer til det. Samtidig fik vi også set hvor nemt det bliver at lave selv forholdsvis store ændringer i databasen når man bruger entity frameworket. Der blev selvfølgelig en del refactoring i vores gamle projekt, men på trods af de store ændringer var det hurtigt og nemt at få til at virke igen.

4.3 Sprint 2

4.4 Sprint 3

4.5 Sprint 4

4.6 Sprint 5

Kapitel 5

Teori

5.1 Patterns

5.1.1 Hvad er patterns

Som nævnt i forordet på s. 3, kan patterns beskrives som generelle løsninger, der har vidst sig at virke på mange problemer. Patterns er altså en form for "altmuligværktøjer"som en programmør kan benytte sig af.

5.1.2 Formål

Formålet, eller ideen med patterns er, at det er løsninger, der kan bruges i mange tilfælde. På denne måde undgår programmører at skulle opfinde den dybe tallerken igen og igen. Samtidig er patterns den bevist bedste måde at løse problemerne på, så man undgår fejl, såfremt patterns'ne implementeres korrekt.

I det følgende vil vi forevise en række patterns, som vi har brugt i arbejdet med CRM-systemet. Vi vil gennemgå hvilke former for problemer de løser, hvordan de implementeres, og hvordan og hvorfor vi har benyttet os af dem. Vi har dog valgt at se bort fra helt simple patterns som f.eks. singleton.

5.1.3 Forskellige patterns

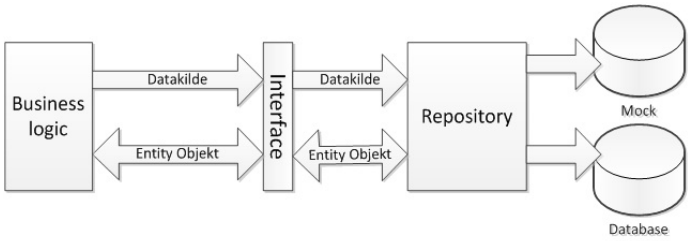
Repository

Repository pattern er en måde at adskille kommunikationen mellem databasen og resten af programmet. Det er her vi skriver og læser til og fra databasen. Det betyder blandt andet, at en masse kodeduplikation undgåes, idet der kaldes

ned til repositoriet hver gang og genbruger på den måde de forskellige metoder. Ved de mere simple operationer som CRUD, kunne man vælge at lave det om til et generisk repository. Udover mere abstraktion i vores design, giver det også mulighed for bedre at kunne unit teste systemet bedre. Det hjælper med at isolere de metoder der skal testes, ved at gøre det langt nemmere at forfalske (Mock) vores data.

Implementering af Repository

Repositoriet er implementeret med henblik på at kunne unit teste med et testing framework kaldet Moq, hvilket samtidig også giver os den ønskede abstraktion. For at hjælpe med at danne et overblik over vores implementering har vi lavet følgende diagram.



Figur 5.1: Respositorymodel

Som det ses her, er forrentningslogikken placeret først. Her bestemmes hvilken datakilde der skal bruges. Det vil være forskelligt om repositoriet tilgåes fra f.eks. en unit test med Mocked data, eller hvis det er en Integrations test som vil bruge de virkelige data til at teste på, eller hvis programmet bruges normalt. Dernæst er interfacet, som fører videre ned i repositoriet som så henter den ønskede data fra den ønskede datakilde.

Vi startede med at lave et repository lag i projektet. Her oprettes alle repositories samt deres interface. Næsten alle MVC controlleres har hvert sit repository da controllerne næsten altid dækker et bestemt område i databasen. Hvis dette ikke er tilfældet, genbruges det repository som har de nødvendige metoder. På den måde genbruges mange af metoderne i repositories'ne.

Vores repository implementerer det interface der hører sig til. Derudover har vi en constructor hvor vi sætter hvilken datakilde den skal bruge. Da vi bruger Moq kan vi mock dette så den eksempelvis opfatter en simpel tabel som datakilde. Dybere forklaring omkring dette kommer senere i afsnittet.

```
1 public class ContactRepository : IContactRepository
2 {
3     private CRMSystemEntities _db;
4     public ContactRepository(CRMSystemEntities repository)
```

```
5       {
6           _db = repository;
7       }
8   }
```

I selve respositoriet vil der selvfølgelig være en implamention af hver enkel metode angivet i interfacet, de er undladt her da de ikke er relevante.

Da det er vores controllere der står for at styre hvilken datakilde der skal gøres brug af, har vi oprettet en række contructors til netop at styre dette.

```
1   public class LeadDetailsController : Controller
2       {
3           private IContactRepository _repo;
4
5           public LeadDetailsController()
6           {
7               this._repo = new ContactRepository(new
                    CRMSystemEntities());
8           }
9
10          public LeadDetailsController(IContactRepository repo)
11          {
12              this._repo = repo;
13          }
14
15          public LeadDetailsController(CRMSystemEntities db)
16          {
17              this._repo = new ContactRepository(db);
18          }
19
20          public IContactRepository LeadDetailsRepository
21          {
22              get { return _repo; }
23          }
24      }
```

Vi har vores standard constructor uden parameter som bliver brugt af når vi bruger programmet normalt. Det vil sige, så snart et View bliver loadet denne kaldt. Den bruger bare vores entity framework som standard.

Den anden constructor bruges når der skal mockes data til test. Som det kan ses tager den et interface som parameter. Det er det object der mockes ved unit tests. Den sidste bruges hvis dette repository skal bruge fra en anden controller. I Entity frameworket er objekter bundet til en bestemt context, så for at kunne sende objekterne igennem controlleren skal den context med, som de er bundet til. Det er dette denne constructor gør. Derudover er der en simpel property for at kunne få fat i repositoriet fra andre controllere.

Unit Test med repository

(Denne unit test hører til en anden MVC controller og repository end de forgående eksempler) Her vil vi kort beskrive en test metode, for at vise hvordan

vi benytter repository pattern i forbindelse med unit testing.

```
1  [TestMethod()]
2  public void BugDetailsTest()
3  {
4      Ticket[] ticketss = new Ticket[] {
5          new Ticket() {Ticketsid = 1, Headline = "test1", Content =
               "bla",... },
6          new Ticket() {Ticketsid = 2, Headline = "test2", Content =
               "blabla",... },
7          new Ticket() {Ticketsid = 3, Headline = "test3", Content =
               "blablabla",... }
8      };
9      int id = 2;
10     Mock<ITicketRepository> mock = new Mock<ITicketRepository>();
11     mock.Setup(m => m.GetDetails(id)).Returns(ticketss[id - 1]);
12
13     TicketController target = new TicketController(mock.Object);
14     var result2 = target.BugDetails(id) as ViewResult;
15     Ticket ticket = (Ticket)result2.ViewData.Model;
16
17     Assert.AreEqual("blabla", ticket.Content);
18 }
```

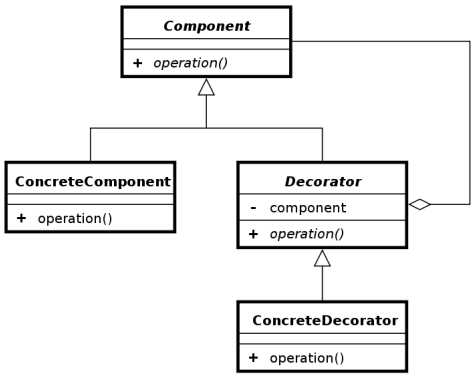
Testen er en ret standard unit test der følger den typiske "arrange, act, assert". Der startes med at lave et array af Tickets som her vil blive datakilden. Dernæst lavers et mock af ITicketRepository. Derudover laves en instans af TicketController hvor der bruges den contructor der tager imod et ITicketRepository objekt som parameter. På den måde ved testen, at vi vil benytte vores array af Tickets som datakilde. Til sidst tjekkes det om det er den forventede data der returneres.

Decorator

I og med, at der skulle laves et dynamisk filter, hvor det på forhånd ikke var vidst hvilke parametre der skulle filtreres på, var vi nødt til at finde en løsning der gav os en hvis grad af fleksibilitet.

Problemet med at bruge Entity framework og LINQ er, at det ikke er muligt at lave dynamiske queries på samme måde som i sql.

Normalt bruges factory pattern til at tilføje funktionalitet til en klasse. Et decorator pattern fungerer ved, at man har en ConcreteComponent, som decoratorklassen arver fra. Decoratorklassen er den klasse som skal dekoreres. Den initialiseere, hvorefter man giver den videre til en anden klasse som parameter. Denne nye klasse tilføjer så funktionalitet, og den nye klasse bliver så igen givet videre til en anden klasse osv. Denne senden klasser videre, bliver gjort mulig via et factory pattern. Et decorator pattern vil især typisk blive brugt ved grafik, hvor man f.eks. kan tilføje funktionalitet som scrollable, resizable osv til et vindue, hvor vinduet et grundklassen.



Figur 5.2: Decorator Model

FilterItem	
Values	List<string>
Criteria	List<int>
CompanyList	List<Company>
Type	string
Category	string

Tabel 5.1: Attributter for FilterItem-klassen

På modellen kan ses "grundklassen"(concreteComponent) og dekoratøren, som er den klasse der skal dekoreres. Øverst ses interfacet Component, som er det interface decorator'en arver fra.

Måden vi implementerede decorator-pattern'et fra, var ved at først lave en generel abstrakt klasse "ConcreteQuery". Denne klasse indeholder ganske simpelt en liste over companies samt en metode "DoQuery()". Desuden lavede vi en klasse for hver kolonne i oversigten, som så kan kaldes og filtrere resultaterne.

Hver klasse har en constructor der tager imod det FilterItem der hører til dem, og samtidig tager imod Company-listen som den ser ud pt. Herefter bliver filteret påført listen, og listen returneres.

Som det ses af kodeeksemplet i figur blah, kan man se, at vi har en fælles constructor der bliver arvet fra ConcreteConstructor, denne constructor bliver kaldt og modtager kriterie og værdi fra det respektive FilterItem-objekt. Disse lægges i en Herefter tager query-klassen over, og behandler data hvis det er nødvendigt, f.eks. hvis der er tale om tal-data, bliver de konverteret fra en string til integer.

Det er forholdsvist simpelt når vi benytter filteret. Filtercontrolleren får tilsendt en liste med filterItems. Som det ses på figur 5.1 er FilterItem ikke andet end

en beholder der holder værdier der er relevate for et enkelt filter.

Herefter løbes listen af filtre igennem, og den dertilhørende query hæftes på concreteQuery-klassen via klassen QueryFactory. Resultatet heraf returneres til controlleren, controlleren pakker serialiserer herefter listen til JSON og sender den tilbage til klienten.

Bilag A

UserStories

Bilag B

Databasediagram

Litteratur

[1] Ramez Elmasri. *Fundamentals of database systems*. Pearson Addison Wesley, Boston, 2007. ISBN 032141506X.

[2] Craig Larman. *Applying UML and patterns*. Prentice Hall, Upper Saddle River, NJ, 2005. ISBN 0131489062.