Webbaseret CRM-system
EqualSums

Peter L. Thomsen Gregers Boye-Jacobsen

Januar 2012

Indhold

0.1 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 1

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.

1.1 Product Backlog

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

1.2 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.

1.2.1 Sprint backlog

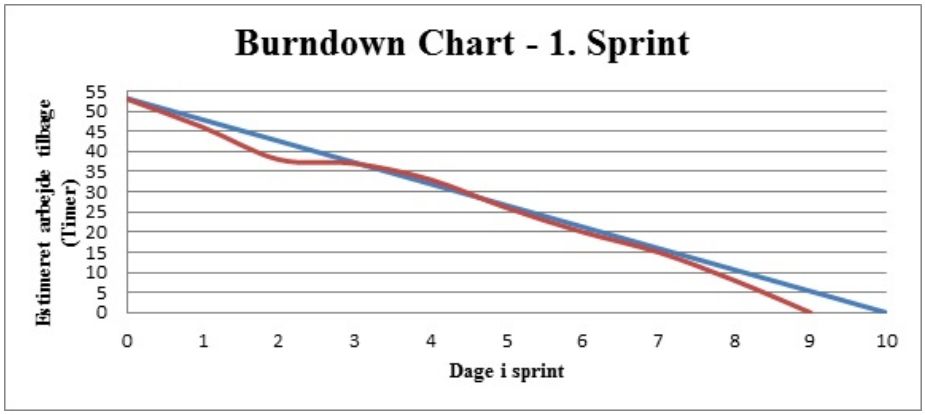
Der blev ikke sat så mange user stories ind i dette sprint for få en god buffer til uventet problemer med arkitekturen og opstart af projektet. Der var også lidt usikkerhed omkring hvor meget der kunne laves på et sprint. Kolonnen "Effort"er det antal timer der er givet hver User Story. Der kan maximalt 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 1.1: Sprint 1 backlog

1.2.2 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 1.2: Sprint 1 Burndown Chart

1.2.3 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

1.2.4 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.

1.3 Sprint 2

1.4 Sprint 3

1.5 Sprint 4

1.6 Sprint 5

Kapitel 2

Teori

2.1 Patterns

2.1.1 Hvad er patterns

Som nævnt i forordet på s. **??**, 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.

2.1.2 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.

2.1.3 Forskellige patterns

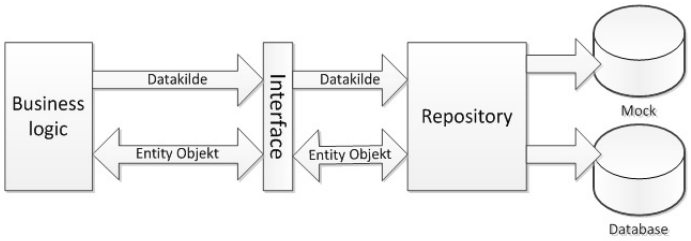
Repository

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 2.1: 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 }
```

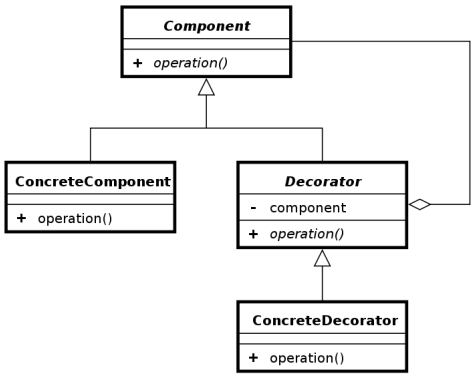
Testen er en ret standard unit test der følger den typiske "arrange, act, assert". Der startes med at lave et array af Tickets som her vil blive datakilden. Dernæst lavers et mock af ITicketRepository. Derudover laves en instans af TicketController hvor der bruges den contructor der tager imod et ITicketRepository 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 2.2: Decorator Model

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

Tabel 2.1: 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 2.1 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.

Litteratur