Indholdsfortegnelse

Introduktion

Forord

Denne rapport er udarbejdet som en redegørelse til eksamensprojektet, Casablanca Hotel Casen, på CPH Business Academy, i forbindelse med den endelige eksamen på 2. semester. Projektet består af et program udviklet med en agil arbejdesmetode, Scrum, og er designet til at kunne blive brugt på Casablanca Holiday Centre; til styring af centrets indlogeringer og kunde administration. Projektet begyndte Tirsdag d. 18/3 - 2014 og sluttede i slutning af maj 2014. Rapporten er udfærdiget af gruppen bestående af: Simon Grønborg, Thomas Hedegaard, Robert Elving, Kasper Hald og Christopher Mortensen. Formålet med denne rapport er først og fremmest at den skal fungere som eksamensmateriale for gruppens medlemmer. Derudover er den rettet mod alle, som er interesseret i vores program, og vil vide hvordan vi fik det udviklet med Scrum, Git og andre værktøjer. Rapporten beskriver de vigtigste dele af det endelige produkt samt arbejdsprocessen og herunder hvilke tanker, overvejelser, beslutninger og antagelser der er gjort. Rapporten fremlægger gruppens konklusion på både arbejdsprocessen og produktet.
Til slut vil vi gerne sige tak til en række mennesker, som har bidraget til projektet. Vi vil gerne sige tak til Jette Kreiner og Henrik Hauge, for deres vejledning, ideer og faglig viden. Vi vil også gerne sige tak til Palle Beck, for hans gode og konstruktive arbejde som product owner, og som vejleder til de metoder vi har anvendt i programudviklingen.

Indledning

> “One dosen’t discover new lands without loosing sight of the shore!”
>
> - Andre Gide

“Man skal kravle før man kan gå”, er ikke et ordsprog som altid er rigtigt. I denne opgave skal vi hjælpe et ferieresort i Marocco, med at komme ind i det 21. århundrede, ved at lave et topmoderne IT-system, som kan håndtere alle de reservationer de for fra kunder. Vi skal lave et booking system som receptionisten skal bruge, til at oprette reservationer for kunder. I dag ville det være smartere med en hjemmeside hvor kunder selv kan klare deres reservationer, men det er en opgave som først skal klares i 2015. Når det er sagt, bryder vi disse bånd og designer vores program med det i tankerne. Så programmets arkitektur skal være designet med en web-udbygning i tankerne. Programmet skal kunne skabe et stort overblik og visualisere disse reservationer.

Case beskrivelse

Casablanca holiday Centre er et ferieresort i Marocco, der tilbyder eksklusive ferier, til folk som godt kan lide aktive sportsferier. Centret ligger 5 km. fra Casablanca’s centrum og 1 km. fra stranden. Casablanca Holiday Centre blev indviet og stod færdigt i 2010, 4 år efter det blev

udtænkt. Resortet har 104 lejligheder, delt op i tre kategorier; Single, Double og Family, som hver har sit at tilbyde til gæsterne. Casablanca holiday Centres mål er at give de besøgende den absolut bedste oplevelse, og det skal de 52 ansatte, sportsfaciliteterne, pool- og golf-områderne sørge for. Casablanca Holiday Resort bruger i øjeblikket et excel-system til at organisere alle de reservationer, de for fra kunder og rejsebureauer. Hele arbejdesprocessen med reservationer bliver, alt efter om det er fra bureauer eller private, behandlet af sekretæren og manageren eller receptionisten og manageren. Centret kører godt, men for at det kan kaldes en komplet succes, skal der laves et nyt IT-system da det nuværende er til skade for kundernes oplevelse og de ansattes effektivitet.

Funktionnele krav til IT-løsningen:

Reservér en lejlighed

- En booking skal kunne skrives direkte ind i Systemet, med alt relevant data. Telefon nummer, navn, adresse, email, lejlighedsnummer osv. (Se bilag 1 - Casablanca Case)
- Løsningens datalager skal køre på en Oracle server.
- Programmet skal kunne køres på minimum 4 computere samtidigt, uden at det foresager samtidighedsproblemer.
- Computerne med programmet skal placeres i receptionen.
- Programmet skal udvikles med henblik på, at det i fremtiden skal blive til en web-løsning, hvor kunder selv foretager deres reservation.

Sidste ovennævnte kriterie er ikke fastslået i casen, men det er nævnt det bliver til en web-løsning i 2015, så det antages som et reelt krav til produktet.

Reserver en sportsfacilitet

- Systemet skal gøre op med chancen for dobbelte reservationer og udnytte den maksimale kapacitet for alle centrets sportsfaciliteter.
- Alle kunder skal have deres eget login til sportsfaciliteterne, som kan reserveres op til en uge frem, eller til en time før det skal bruges.
- Ved travle aktiviteteter skal der være en venteliste, kunderne kan skrive sig op på.
- En reservation skal kunne annulleres op til to timer før.
- Receptionisten skal også kunne reservere sportsfaciliteter til kunderne hvis, de ønsker det.

Ikke funktionelle krav til IT-løsningen:

- Systemets brugergrænseflade skal være simpel og informativ.

Til selve opgaven er en række tekniske krav som skal overholdes. Dette er en del af pensum og det materiale som vi skal igennem når produktet og opgaven skal laves.

- Programmet skal laves så det bruger en logisk lagdeling, som skal skrives i Java, alle kørerne på Klienten. Altså en “Fat Client”.

- Vi skal bruge designmønstre som er beskrevet i “Fowler’s Patterns of Enterprise Architecture”.
- Det skal kunne håndtere samtidighedsproblemer.
- Projektet skal udvikles med Scrum.

Problemanalyse

Til dette projekt skal der udarbejdes et program som kan håndtere alle de opgaver, der er nævnt i casebeskrivelsen. Alle kravene til produktet skal opfyldes, men helt specielt har vi valgt at bearbejde to primære problemer. Første problem er ineffektivitet. Det tager for lang tid at reservere en lejlighed. Lige nu bruger resortet et excel system til behandling af reservationer fra kunder og rejsebureauer. Arbejdesgangen for at oprette en reservation er lang. En reservation bliver modtaget af en receptionist som, ser om denne kan lade sig gøre. Derefter skal sekretæren skrive den ind i excel systemet. Hvis det er et rejsebureau skal manageren ind og håndtere reservationerne, og derefter kan sekretæren skrive det ind i systemet.
For at Casablanca kan leve op til sit gode ry, skal der laves et nyt IT-system, som kan erstatte den nuværende excel-løsning. Systemet skal gøre hele processen mere enkel og økonomisk besparende, så ikke der skal lægges så mange timer i at oprette reservationer. Vores program skal designes med effektivtet i tankerne, så de “ikke funktionelle” krav om simplicitet bliver vægtet højt.
Det næste problem som opstår er, det nye system kun er midlertidigt. I 2015 vil en web-løsning blive integreret, så programmet skal være fleksibelt og nemt at udbygge til en web-løsning

Hvordan skal det gribes an

Der er flere måder at gribe større programmeringsprojekter an på. I vores projekt vil vi bruge Scrum, til at organiserer vores arbejdsproces og opgaverne. Med Scrum kan vi opdele og uddelegere arbejdet uden at låse os fast. (Se afsnittet omkring scrum)
Måden vi vil gribe det an på, er først ved at skabe et overblik over projektet, så vi kan få en ide over størrelsen af det program der skal laves. Til det laver vi et klassediagram, indrettet i de klasser programmet skal bestå af. Sideløbende med klassediagrammet skal relevante designmønstre udvælges til brug i programmet.
Det teksniske overblik:

- Git til versionsstyring.
- Branches til bugs, experimentelle funktioner og hotfixes.
- Google drive til resource lagging og distribution.
- Github som program host.
- Whiteboards til Scrum resourcer, tasks, user stories go burndown charts
- Lo-Fi modelering til udvikling af den grafiske brugergrænseflade.

Versionsstyring håndteres af Git; til dem der ikke kender Git, er det et revisions værktøj som SVN og Mercurial. Git, i sammenhæng med Github fungerer som back up af projektet og distribution. Git's branch funktionalitet bruger vi til at organiserer udviklingen af programkoden. En branch indeholder projektets sidst virkende version, de andre branches bruges til at lave eksperimentelle ændringer i programmet og bug fixing. Til at designe brugergrænsefladen benytter vi os af lo-fi modelering. Når opgaver løses følges denne procedure:

1. Udvælg en relevant opgave på baggrund af programmets stadie
2. Opret en branch til udvikling af den nye funktionalitet
3. Lav tilføjelsen på klassediagrammet.
4. Lav strukturen i koden så alle andre dele af programmet vil få adgang til de nye metoder.
5. Tilføj logikken til metoderne.
6. Test logikken.
7. Få koden gennemset af et andet gruppemedlem
8. Få flettet den færdige feature sammen med den virkende version.

Læsevejledning

Denne rapports indhold afspejler de valg vi har taget om hvad vi synes de vigtigste elementer i vores produkt og vores arbejde er, i forhold til hvad der er blevet lagt vægt på i projektoplægget. Indholdet i rapporten er udvalgt efter en liste i projektoplægget og valg taget på baggrund af de emner vi kan komme op i til eksamen. Indholdet om produkt og process har vi overvejende fordelt ligeligt, så både process og produkt er i fokus. I afsnittet om process ligger fokus udelukkende på Scrum, roller og vores brug af Git, da det var de absolut centrale dele af hele projekt forløbet. Der vil også blive nævnt noget om roller og vores samarbejde, men det står skrevet sammen med vores brug af Scrum.

Afsnittet om produktet er valgt ud fra et design synspunkt. Gang of Four og Martin Fowler mønstre, har vi i højere grad lagt vægt på at dokumentere, da det har valgene af disse har de største indslag på koden.

BEMÆRK: Vi beder læseren om at starte vores program, det skulle gerne være ligetil, da det er *en* eksekverbar fil. Skulle der være problemer, se da bilag 21 .

Produkt rapport

Product Backlog

Vores Product Backlog (Se bilag 2 - Product backlog) blev stykket sammen i fællesskab, efter at have nærlæst vores opgavebeskrivelse. Alle krav blev noteret, og vi udarbejdede en Product Backlog ud fra de krav vi fandt frem til. Vi formulerede dem, så de var meget adskillelige og af fin størrelse. Disse produkter brugte vi senere som userstories. Hvis vi endte med at lave en user story der var for stor, prøvede vi at splitte den op i mindre dele. En user story som f.eks. ”Receptionisten kan oprette bookings for kunder og manipulere med dem”, blev vi enige om var en alt for stor og overordentlig user story, at kunne arbejde med. Det lykkedes at splitte den op i; ”Receptionisten kan oprette bookings for kunder, og gemme dem til senere brug.”, der fokuserede på oprettelsen, samt ”Receptionisten skal kunne redigere en booking”, der fokuserede på redigeringsdelen. Med disse to userstories, ville det være nemmere at finde hovede og hale i dem. Hver user story i vores Product Backlog havde følgende attributter:

ID	Description	Importance	Estimate	How To Demo

Det er vigtigt at man giver hver user story et unikt ID, så man ikke blander tingene sammen. Specielt når vi skal til at klippe dem ud i papir, er det meget funktionelt at give hver user story et ID. En nøgtern beskrivelse er vigtig, så vi kan forstå hvad en user story går ud på. Desuden skal vores product owner altid have rådighed over den færdige product backlog, så sproget skal være let læseligt og kontant. Importance beskriver hvor vigtig vores user story betyder for programmet. Vi stillede spørgsmålet; Hvor stor en rolle for det færdige program spiller den pågældende user story og hvad skal vi derfor lave først?
Vi fandt det først og fremmest vigtigst, at receptionisten skal kunne oprette bookings for kunder, samt gemme dem til senere brug. Vi prioriterede derfor denne user story højest på listen, da vi syntes, at det var den mest essentielle feature i programmet. Når man skal beskrive hvor vigtig en user story er, bruger vi meget simpelt tal til at repræsentere deres værdi. Jo højere en værdi, desto vigtigere er en user story. Det ligegyldigt hvor stor differencen imellem hver user story ligger på, da det ikke siger noget om hvor vigtig den er. Det er hierarkiet af dem som er betydende for deres vigtighed. Derfor er der ingen user story der kan have den samme vigtighed. Når vi skal bruge userstories til at placere på vores sprint backlog, er man ikke tvunget til at tage dem med højeste prioritet, men vigtigheden kan give et indblik i, hvad vi skal lave før vi starter på andre komponenter til vores program.
Vi har derefter estimeret hvert enkel user story i, hvor mange mandedage gruppen skal afsætte til hver. Hver user story skal så splittes op i mindre tasks når sprint backloggen skal laves.

Vi har også inddraget en How To Demo-del til vores userstories. Med et par linjer, skal der stå hvordan man via det færdige program skal navigere rundt, for at demonstrere, at en user story er færdig.

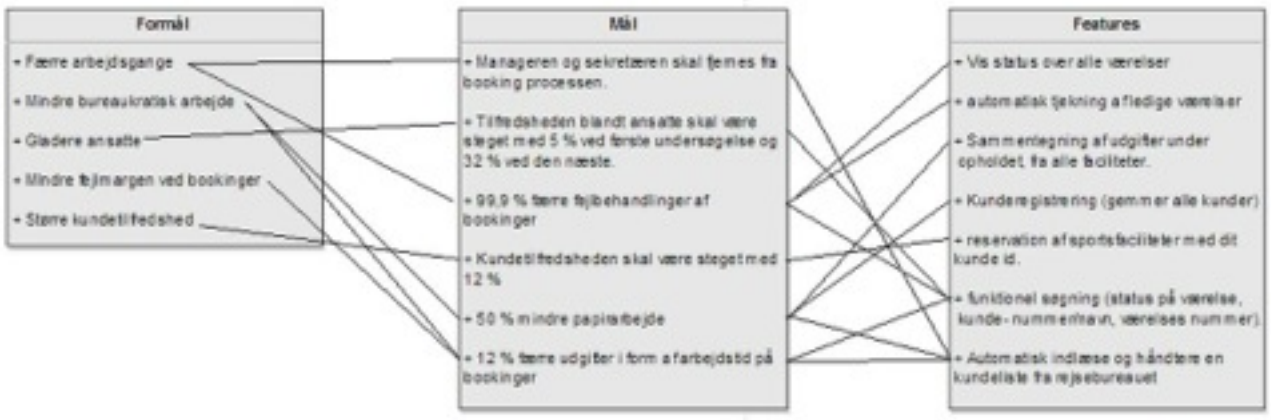
Hvor mange user stories nåede vi?

Under de tre sprints vi havde til rådighed, fik vi færdiggjort seks userstories, hvoraf to af dem var lavet spontant, fordi vi havde underestimeret for lidt tid til testning og refaktorering af programmet. Den tilhørende rapport, stjal også hurtigt billede i den sidste rapport.
Første sprint gik ud på at få udarbejdet de to øverste userstories på vores Product Backlog, netop dem med den højeste prioritet; En receptionist skal kunne oprette og gemme bookings. En receptionist skal kunne redigere og slette en booking.
Vi mødte en del komplikationer i det første sprint, derfor vi ikke nåede at færdiggøre den sidste user story, så andet sprint havde følgende userstories; En receptionist skal kunne redigere og slette en booking. Hent info fra tidligere kunder ved oprettelse af ny booking. Man kan få en liste frem med alle værelserne på nuværende dato.
Det lykkedes os at færdiggøre alle tre userstories, men med et sprint tilbage, hvor vi ikke har sat tid af til refaktorering, testning og rapportskrivning, lagde vi de resterende userstories bag sig og konstruerede to til vores tredje sprint; Refaktorering, test og Rapport.

1. O - Opret booking
2. H - Rediger en booking og Customer
3. Q - Hent info fra tidligere kunder
4. F - Overskuelig grafisk brugergrænseflade til reservations visning

(Se bilag 2 - Product backlog)

Sporbarhedsmodel



Figur 1: Traceability model

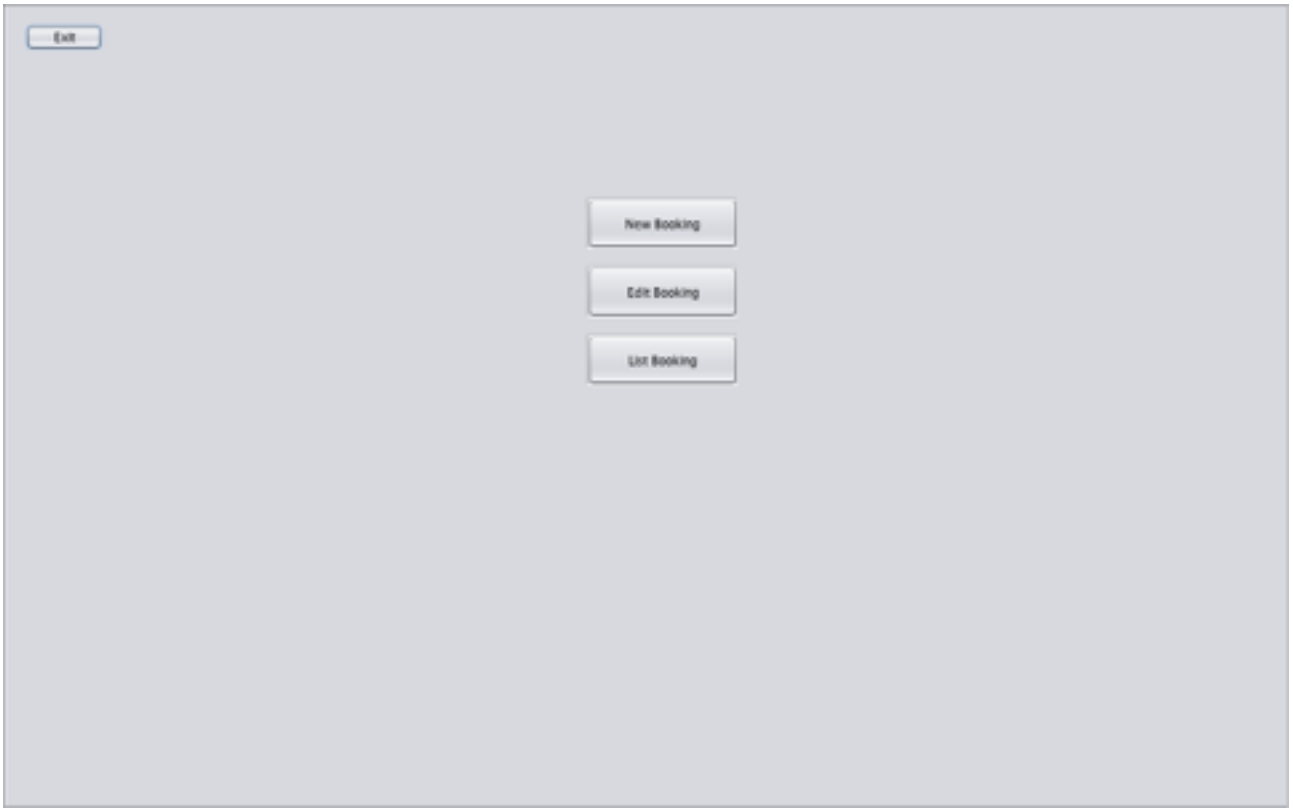
Centret vil gerne have at personalet i fremtiden bruger IT-systemet til at udføre reservationer. Formålet med dette er forhåbentligt at, det resulterer i færre arbejdsgange, da IT-systemet skal laves for at automatisere en del af de manuelle opgaver personalet bruger meget tid på. Dette resulterer i reduceret bureaukratisk arbejde og forhåbentligt gladere ansatte.
IT-systemet skal kunne administrere mange reservationer og sørge for at forhindre dobbeltreservationer, hvilket betyder mindre margin for fejl.
Systemet skal medføre at manageren og sekretæren skal fjernes helt fra reservations processen, så de har mere tid til andre opgaver. Tilfredsheden blandt de ansatte skal være steget, med et målbart tal lavet ud fra undersøgelser. 5% ved første undersøgelse og 32% ved den næste.
Da man ikke kan være 100% sikker på at ingen fejlbehandlinger af reservationer opstår, går vi ud fra at 99,9% af alle reservationer ikke er fejlbehandlet.
Via en tilfredshedsundersøgelser vil vi sammenligne kundetilfredsheden før og efter indførelsen af IT-systemet. IT-systemet vil også spare forbruget af papir og derfor sigter vi efter en reduktion på over 50% . Da vi i vores formål havde nævnt færre arbejdsgange for personalet er vi sikre på at centret vil have 12% færre udgifter i form af arbejdstid, da processen er blevet kortere.
Med henblik på features, der skal kunne sørge for at målene bliver realiseret ville et godt overblik over værelserne, i form af en overskuelig liste med status på værelserne, være en meget vigtig feature. Et automatisk tjek af ledige værelse er brugbart for receptionisterne, når de hurtigt skal udføre en ny reservation. Personalet skal have muligheden for at få en sammenfatning af diverse udgifter, så et overskueligt regnskab skal konstrueres. Når kunder reservere et værelse, skal de registreres, så de hurtigt kan blive søgt efter i IT-systemet, via den funktionelle søgemaskine, der derudover også kan søge efter informationer på værelserne.
Hvilke features blev nået? Det lykkedes os at vise status over værelserne via vores listevisning, så denne feature er opfyldt. En feature som et automatisk tjek af ledige værelser er en feature der foregår i koden, hvilket var noget vi implementerede som det første, da vi skulle oprette nye bookings. Kunderegistrering og sammentegning af aktive kunders udgifter fra opholdet var også features, begge til at formindske papirarbejdet for receptionisterne, som blev nået at få implementeret. En funktionel søgning er også implementeret inde i programmet. Reservation af sportsfaciliteter og automatisk indlæsning af kundeliste fra et rejsebureau er undladt, da vi ikke igangsatte arbejdet på de sidste user stories.

Beskrivelse af programmet

Ideologi

Programmet er blevet designet efter en simpel ideologi; “Kvalitet frem for kvantitet”. Programmet skulle være nemt og simpelt at bruge. Vi har mere end en gang været nødt til at tage store dele af den kode vi har skrevet og skrive den helt om, for at forbedre den, men også for at gøre den mere kompatibel med funktioner der skulle implementeres senere.

Frontpage

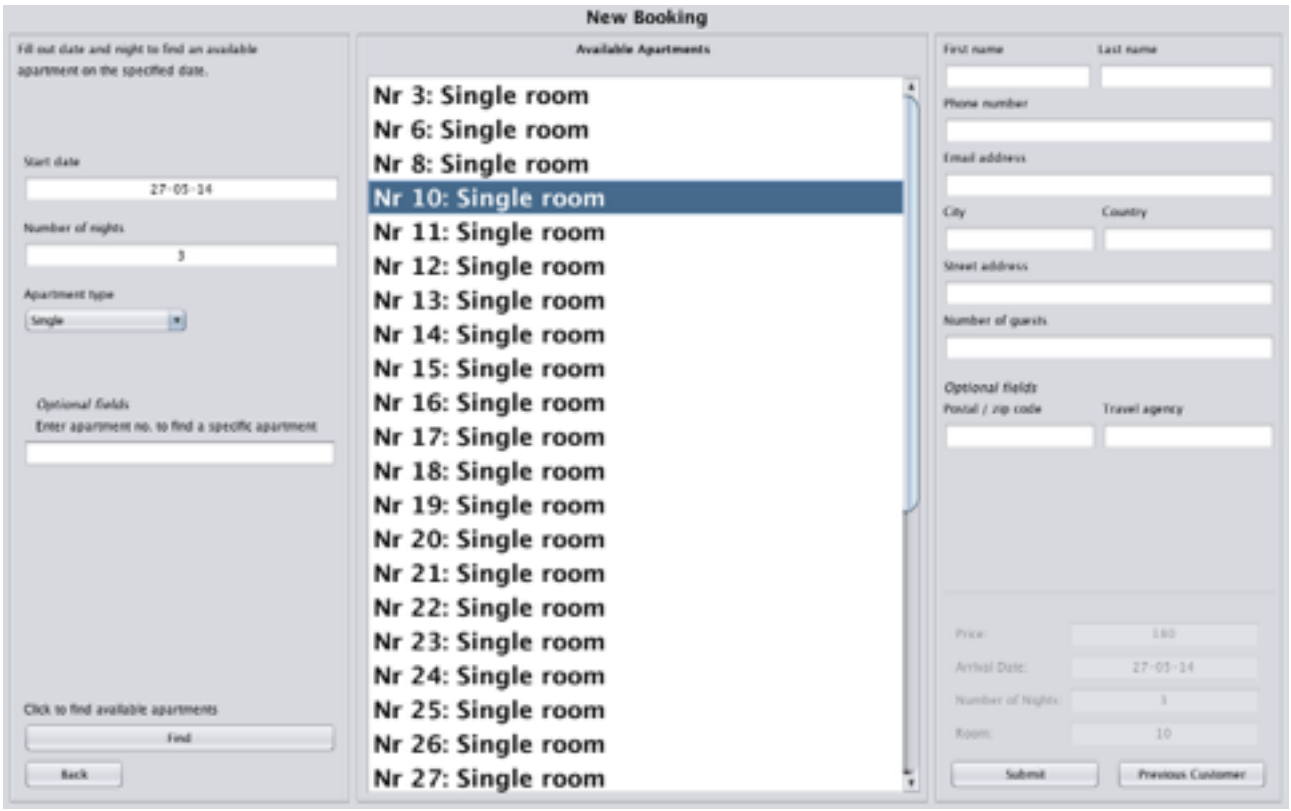


Figur 2: program - Front page

Forsiden af programmet afspejler den måde hvorpå programmet er designet. Funktionaliteter er opbygget af moduler som er blevet implementeret med tiden. De 3 knapper til hver af de tre funktionaliteter afspejler dette. Det giver også et enkelt overblik.

- New Booking - navigerer til “New booking” siden.
- Edit Booking - navigerer til “Edit booking” siden.
- List Booking - navigerer til “List booking” siden.

New booking



Figur 3: program - new booking page

New booking siden er dedikeret til at oprette en reservation og er opdelt i tre kolonner. Det venstre til søgning efter ledige lejligheder, på en given dato og varighed. I midten er listen med de ledige lejligheder som søgningen returnerer. Højre kolonne er formular udfyldning.

Booking information

Det første skridt for at oprette en booking er at udfylde informationerne i reservations formularen.

- Start date - Hvilken dato en booking skal starte på.
- Number of nights - Hvor mange nætter en booking skal varer.
- Apartment type - Hvilken type lejlighed som skal bookes.
- Apartment no. - Hvis det ønskes en bestemt type lejlighed, kan man søge på lejligheds nummeret her. Dette er valgfrit.

Apartment search

Når man har udfyldt booking informationen, skal man vælge en ledig lejlighed på listen. Hvis der ingen lejligheder kommer frem, er der ingen ledige lejligheder. Lejlighederne findes ved at trykke på “Find knappen”

- Find - Finder alle ledige lejligheder baseret på søgekriterierne.

Customer information

Her udfyldes til sidst relevant info om en kunde.

- First name - En kundes fornavn.
- Last name - En kundes efternavn.

- Phone number - En kundes telefonummer.
- Email Adress - En kundes email aresse.
- City - En kundes hjemby.
- Country - En kundes hjemland.
- Street Adress - En kundes adresse.
- Number of guests - Antallet af besøgende, inklusiv kunden, som skal bo i lejligheden..
- Zipcode - En kundes postnummer. Dette felt er valgfrit.
- Travel agency - Hvis bookingen kom igennem et rejsebureau, skrives bureauets navn her, ellers efterlades det blankt.

De tre efterfølgende felter er baseret på bookingen og lejligheden, og bliver automatisk udfyldt.

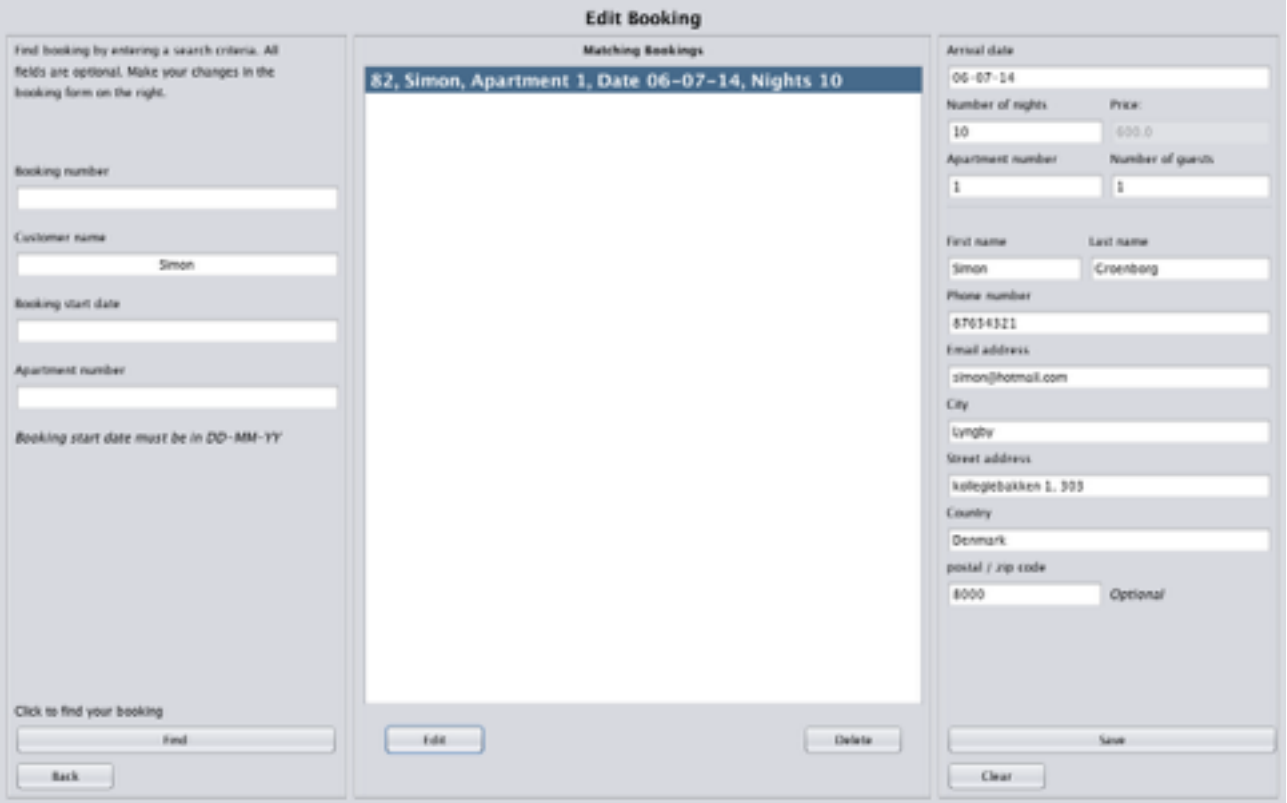
- Submit - Tilføjer bookingen, og gemmer ændringerne i databasen.
- Previous customer - Går til et seperat vindue for at hente information fra en tidligere kunde.

Previous customer

Figur 4: program - previous customer page

Previous customer siden gør en i stand til at hente information fra tidligere kunder. I søgefeltet indtastes der hvad man vil søge på, hvor efter den vil søge efter alle kunder med et fornavn, efternavn eller telefonnummer, der passer til søgekriterierne. Hvis den ønskede kunde er fundet, trykker man på kunden, og vælger den. Når man kommer tilbage til 'New Booking' siden, vil de relevante felter være fyldt ud med kundens info. Ellers vil et tryk på cancel returnere uden at foretage ændringer.

Edit booking



Figur 5: program - edit booking page

Edit booking giver en muligheden for at ændre på reservationer allerede oprettet i systemet. Siden er delt i tre dele, én for søgekriterier, én for søgte reservationer, og én for booking- og kundeinformation. Når man søger, er alle søgefelter valgfrie. Man indtaster søgekriterier baseret på hvad man vil søge på.

- Booking number - Det automatisk tilføjede nummer til bookingen, da den blev oprettet.
- Customer name - Søger på personens fornavn.
- Booking start date - Start datoen for bookingen.
- Apartment number - Nummeret på den lejlighed der indgår i bookingen.

Find - Søger i databasen efter reservationer, baseret på de indtastede kriterier.
Når man har søgt efter reservationer kommer der en liste frem med reservationer, og deres generelle information. Man vælger en booking ved at klikke på den, hvorefter man kan ændre den i informations kollonen.

Save - Gemmer ændringerne som er blevet foretaget. Hvis ændringerne vil gøre reservationen ugyldig, vil det blive skrevet en fejlmeddelelse på skærmen, og reservationen vil ikke blive gemt. Alt data skal være gyldig, før en reservation kan gemmes.

Cancel - Går tilbage til forsiden uden at gemme ændringer.

List booking

List booking gør en i stand til at få et overblik over de reservationer der ligger i databasen. List booking har 3 overordnede funktioner, et overblik over dagens vigtige reservationer, et overblik over reservationer for en bestemt måned, og et overblik over reservationer for en lejlighed.

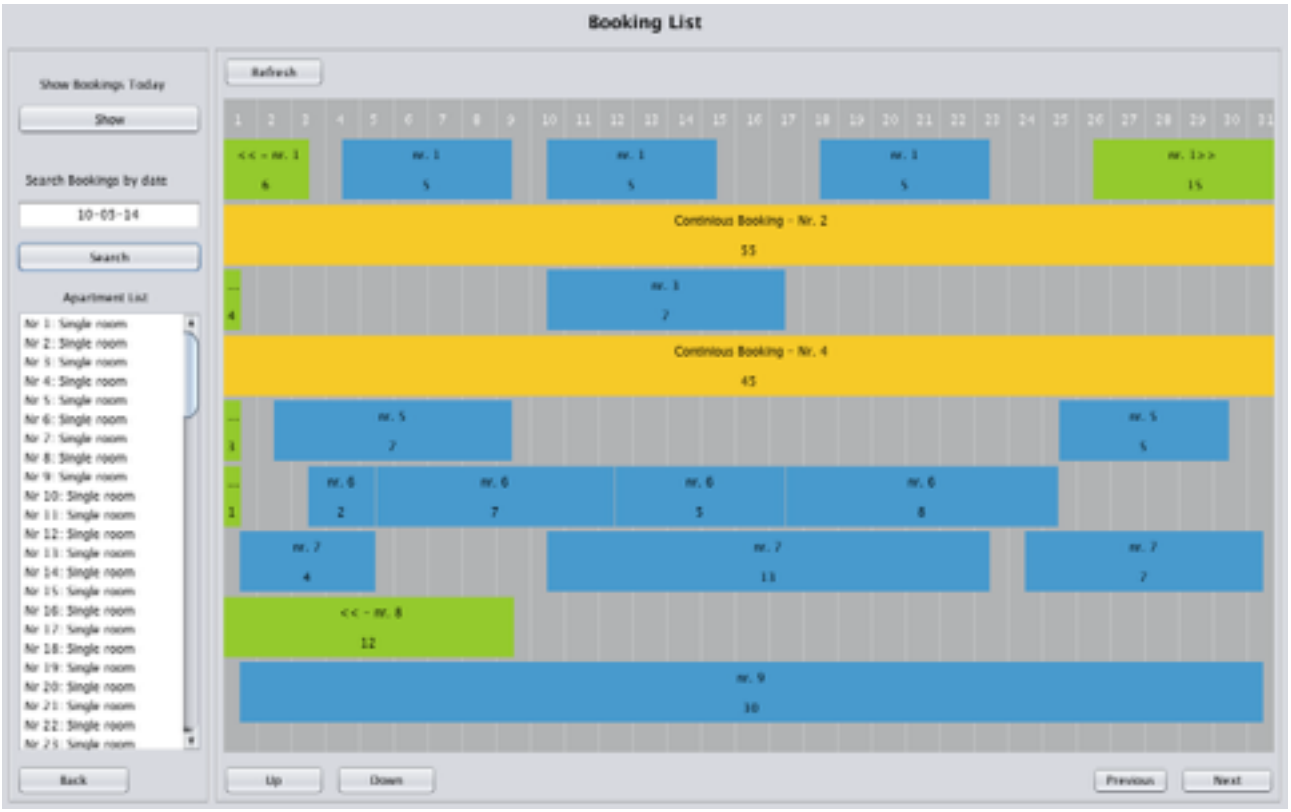
Show today

Viser en liste over alle relevante reservationer for den nuværende dag dag. Kun reservationer som starter eller slutter den pågældende dag, bliver vist. Er der flere reservationer en der kan vises på siden, kan man bladre op og ned ved at klikke på knapperne i bunden til venstre. Man kan klikke på en reservation, og få et popup-vindue op med mere information om reservationen.

Figur 6: program - List booking - show today

Show month

Viser en liste over alle reservationer for en måned. Man indtaster en dato i søgefeltet, hvorefter listen viser alle reservationerne for den måned. Listen er sorteret efter lejligheds nummeret, og man kan se reservationer for flere af lejlighederne, ved at klikke på knapperne i bunden af vinduet, hvor der også er knapper for at skifte hvilken måned der vises. Klikker man på en reservation, får man et vindue op med yderligere information om reservationen. (Billedet på næste side)



Figur 7: program - List booking - show month

Show apartment

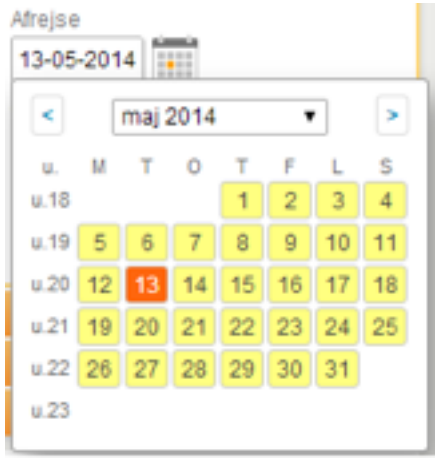
Viser en liste over reservationer for en bestemt lejlighed. Man vælger lejligheden i listen, nederst i venstre hjørne af vinduet, hvorefter reservationer bliver vist. Reservationerne ligger i kronologisk rækkefølge, og der vises minimum 3 måneder frem.

Figur 8: program - List booking - Show apartment

De vigtigste features

Hurtig dato indskrivning

Da det at oprette en ny reservation, vil være en opgave receptionisterne kommer til at udføre, mange gange i løbet af en dag. Vil en lavere ekspeditionstid betyde, at flere kunder kan serviceres. Derfor er det altafgørende at indtastningen af data i formularene sker hurtigt. Ved indtastning af datoer i vores formularer overvejede vi at implementere en lille “drop down” kalender hvor receptionisten kunne vælge datoer.(se nedenstående billede)



Figur 9: Spies kalendar - spies.dk

Men en garvet receptionist der skal lave mange reservationer vil meget hurtigt blive træt af at muse rundt i kalenderen for at finde startdato og slutdato. Derfor har vi valgt at når brugeren taster datoen ind, skal de kun indtaste tal for dagen, måneden og året. Dato formatet i vores booking system er DD-MM-YY. Det er nemt at se hvilken dato der er tale om og kræver færre tegn at indtaste end f.eks. DD/MM/YY(shift+7 for /) eller formater med YYYY eller MON.
For at gøre indtastningen endnu hurtigere udfylder programmet bindestregerne mellem tallene og sørger for at dimensioner ikke bliver brudt. Vores implementation kan ses i nedenstående kode eksempel.

```
JTextField selectedTextField = this.listBookingDatejTextField1;
String candidates = "1234567890";
boolean hit = candidates.indexOf(evt.getKeyChar()) >= 0;
      if (hit) {
            int textLength = selectedTextField.getText().
      if (textLength == 2 || (textLength % 2 == 1 && textLength > 1) &&
textLength < 7) {
         selectedTextField.setText(selectedTextField.getText() + "-");
```

```
          }
            if (textLength > 7) {
char[] firstChars = selectedTextField.getText().toCharArray();
selectedTextField.setText("");
                for (int i = 0; i < 7; i++) {
selectedTextField.setText(selectedTextField.getText() + firstChars[i]);
                }
            }
        }
    }
```

Kodeudklip 1 - lag: presentation, klasse: CasablancaGUI, Linje: 2075-2092

Der checkes efter hvert tastaturtryk, for om det indtastede tegn er en numerisk værdi og hvor langt henne i indtastningen brugeren. Med denne løsning skal brugeren, for at skrive “30-04-14” taste 300414. Det vil hurtigt spare en del tid at gøre det sådan, i forhold til den lille pop up kalender.

Cronjob

Navnet Cron kommer fra UNIX & Linux og er en baggrundsproces, hvis funktion er at udfører handlinger på bestemte tidspunkter med bestemte intervaller.
Da det er lovpligtigt at en forretning som Casablanca holiday Resort gemmer deres gamle reservationer. Vi er ikke er interesseret i at “Booking” tabellen i databasen flyder med gamle reservationer, har vi valgt at de automatisk flyttes over i en anden tabel, “History”. Da vi ønsker at selve flytningen af reservationerne sker under så kontrollerede forhold, så vi er sikre på ingen data går tabt, giver det mest mening at lægge flytningen på et tidspunkt på døgnet, hvor systemet ikke er i brug af receptionisterne, altså uden for receptionens åbningstid. Man kunne vælge at ligge proceduren enten ved programmets opstart, eller nedlukning. Forbi der er fire terminaler er der potentiale for at reservationerne vil blive forsøgt rykket, adskillige gange i løbet af dagen, når receptionister går til pauser, eller på anden måde gå fra deres terminal. Derfor ligger proceduren på et tidspunkt med mindst aktivitet. Vi besluttede os for at proceduren skulle ske klokken 01:00 om natten.
Vi har implementeret denne løsning i vores `CronThreader` klasse.
Når en ny facade instantieres(se nedenstående kodeudklip), som kun sker ved programstart, instantieres også en `CronThreader` der i sin egen tråd kører en stored procedure på databasen, mellem klokken 01.00 og 01.59 alt efter hvad minuttallet er ved programstart.

```
private DbFacade() throws ConnectionException {
        this.chatty = new Chatty();
        con = DBConnector.getInstance().getConnection();
```

```
        CronThreader c = new CronThreader(con);
        new Thread(c).start();
    }
}
```
Kodeudklip 2 - lag: dataSource, klasse: DbFacade, Linje: 24-30

Tidspunktet proceduren på databasen skal eksekveres, udregnes ved at finde ud af hvor mange timer der er til klokken 01.00 og derefter vente det antal timer før første eksekvering. Der tages ikke højde for minutter, så hvis programmet opstartes klokken 18.13 vil flytningen ske klokken 01.13. Proceduren ligger som en stored procedure på databasen og ser ud således:

```
create or replace PROCEDURE movebooking
IS
  cd DATE;
BEGIN
  cd := TO_DATE(sysdate,'DD-MM-YY');

  INSERT INTO history (b_id, cust_id, a_num, date_from, number_of_nights,
travel_agency, number_of_guests, price)
    SELECT b_id, cust_id, a_num, date_from, number_of_nights, travel_agency,
number_of_guests, price FROM booking
  WHERE date_from + number_of_nights < cd;

  DELETE FROM booking
  WHERE date_from + number_of_nights < cd;

END;
/
```
Kodeudklip 3 - Stored procedure “move_booking”

Enkelt sagt flyttes alle de reservationer fra booking tabellen over i history tabellen hvor slutdatoen ligger før dags dato.

GUI - Month View

En brugervenlig, strømlinet og flot brugergrænseflade var nogle af de krav vores product owner prioriterede rigtigt højt. Derfor var det vigtigt at samle så meget relevant information på skærmen på en gang og gøre det let for en receptionist, at lokalisere de eftersøgte informationer. Ved at implementere designdelen af `ListBooking ShowMonth` som i Lo Fi modellen(Se bilag 3 - LoFi show

month), med grafisk blok repræsentation af reservationers placering i en valgt måned, opnår vi dette. Vores implementation af designet stemmer overens med Lo - Fi modellen. Meget af brugervenligheden i systemet, navigerings letheden i brugergrænsefladen.
Ønsker man at se for næste eller forrige måned, kan brugeren trykke *next* eller *previous*. Ønskes der at bladre gennem lejligheder for den viste måned trykkes *up* eller *down*, når *Apartment list* er markeret.

Figur 10: program - List booking - show month

En lille genistreg i `DrawMonth` metoden er måden hvorpå vi sammenligner datoer. Ved at omdanne datoerne til tal, kan vi nemt sammenligne to datoer og finde ud af om en dato ligger mellem to andre datoer. Måden vi gør dette på er ved at gange årstallet med 100 og lægge resultatet sammen med måneden, eks:

```
year = 14;
month = 4;
date = (year * 100) + month = 1404;

year 2 = 14;
month2 = 5;
date2 = (year2 * 100) + month = 1405;
```

dato checkene kunne med denne metode blive sammenlignet på denne måde:

```
date > date2;
date < date2;
date == date2;
```

Vi kunne have inkluderet dagene også men det var ikke nødvendigt, fordi vi skulle bare holde måneder og år op mod hinanden. for at finde ud af om en reservation lå inde i en måned, gik ind eller ud af, eller overlappede måneden helt.

Transaktioner

En vigtig feature i vores program er hvordan vi vælger at håndtere kommunikationen mellem databasen og programmet. Vi har valgt at foretage mange små transaktioner(microtransactions) og have meget kommunikation mellem databasen og koden. Man kan sige vores program er snaksaligt (chatty) hvilket gør at ved fejl mister vi kun en transaktion i forhold til et program der er mere chunky. Et chunky program vil køre efter alt eller intet princippet, hvilket resulterer i at større databidder kan gå tabt ved fejl. Da programmet skal håndtere rejsereservationerne, er det vigtigt der er styr på hvem der har en reservation hvornår, så receptionisten ikke bliver nødt til at sende kunder væk igen, hvis der er sket en fejl eller dobbeltbooking.

Design & arkitektur

I De følgende afsnit vil vi komme ind på hvilke mønstre vi har brugt, hvordan vi har brugt dem og hvorfor. Derudover vil vi redegøre for fordele/ulemper ved brugen af de forskellige mønstre hver især, hvorved vi også vil redegøre og argumentere for hvordan vi har modificeret og tilpasset nogle af mønstrene til netop vores behov.

"Three principal layers"

Da der efter dette princip som vi følger, eksisterer tre overordnede lag, har vi delt vores program ind i tre overordnede pakker. Dette er for at dele programmets klasser ind under hvert af de dertilhørende lag. I Præsentationslaget befinder alle klasser der er relaterede til Brugergrænsefalde, i domænelaget ligger entitetsklasserne; Booking, Guest, Aparment og History, samt en Controller. I det tredje lag, dataSource, ligger alle klasser der er relaterede til kontakten med databasen, samt klassen Chatty, der har til formål at styre transaktionerne. Ved at overholde lagenes ansvarsområder, bibeholder vi en høj lag samhørighed.
Man kan bemærke, at vores domæne lag ikke er inddelt i et service og logik lag. Da vores program ikke er større, valgte vi at fravælge den mulighed.

Hvad er formålet?

Et af formålene ved at lagdele programmet, er overblikket. Ved at lagdele programmet, kan man hurtigt få en idé om hvilke klasser der har ansvaret for hvilke områder.

Figur 11: Henrik Hauge - Archicture and layering

En fordel, vi tog i betragtning da vi designede programmet, var muligheden for at udvide, opdatere eller på anden vis ændre programmet i fremtiden. Lagdelingen muliggør groft sagt, at tage ét lag ud, og udskifte det med et andet. Dette kommer også an på hvordan klasserne kender hinanden. Implementationen af den nye/ændrede funktionalitet simplificeres i høj grad, hvis programmet er delt ind i lag. For at give et eksempel på denne påstand, kan nævnes et af kravene fra vores case. Her bliver det bl.a beskrevet, hvordan man i fremtiden ønsker en webløsning, der skal bygges ovenpå det allerede eksisterende IT-system. Vores nuværende GUI følger med programmet, dvs. man skal installere programmet på sin egen computer for at bruge det. Da man fra ferie resortets side ønsker, at privatpersoner fra deres egen computer skal kunne oprette bookings m.m, skal måden hvorpå man kommunikerer med IT-systemet ændres. Skal man lave en web løsning, kræves en GUI i browseren, men programmets

funktionalitet vil være den samme. Man kan altså tage programmet som det ser ud nu, og udelukkende udskifte præsentations-laget med den web-baserede GUI, og man har stadig den nødvendige funktionalitet; at oprette en reservation. Hvis et lag kan udskiftes, som nævnt ovenfor, vidner det om løs kobling; du har funktionaliteten fra de samme klasser i lagene under dig, uden at du skal ned og ændre i hvordan de kommunikere med det nye lag du selv sidder og arbejder i. Dette påviser, at lagene er selvstændige og uafhængige af hinanden[1].

Facade

Måden hvorpå lagene kommunikerer med hinanden, er designet efter facademønstret. Lagene skal bruge af hinandens funktionalitet. Denne kommunikation sker gennem facadeklasserne; ét lag har én facadeklasse. Denne klasse fungerer som en "træstamme" for det pågældende lag. Den samler funktionaliteten fra andre klasser i det samme lag, og gør den tilgængelig for andre lag. For at give et eksempel, kan nævnes måden hvorpå vores domænelag kommunikerer med vores dataSource-lag. I dataSource-laget er facaden-klassen navngivet `DbFacade`. `DbFacade` er den eneste klasse der bliver instantieret fra det ovenstående lag, og er dermed klassen der giver adgang til funktionaliteten, som udregnes/ligger i andre klasser.
Endvidere skal det bemærkes, at metoder i bl.a. dataSource-laget's klasser er "protected"(Se kodeeksempel nedenfor), med undtagelse af `DbFacade's`.

```
protected ArrayList<Booking> getAllBookings(Connection con) {}
```
Kodeeksempel 4 - lag: dataSource, klasse: BookingMapper, Linje: 13

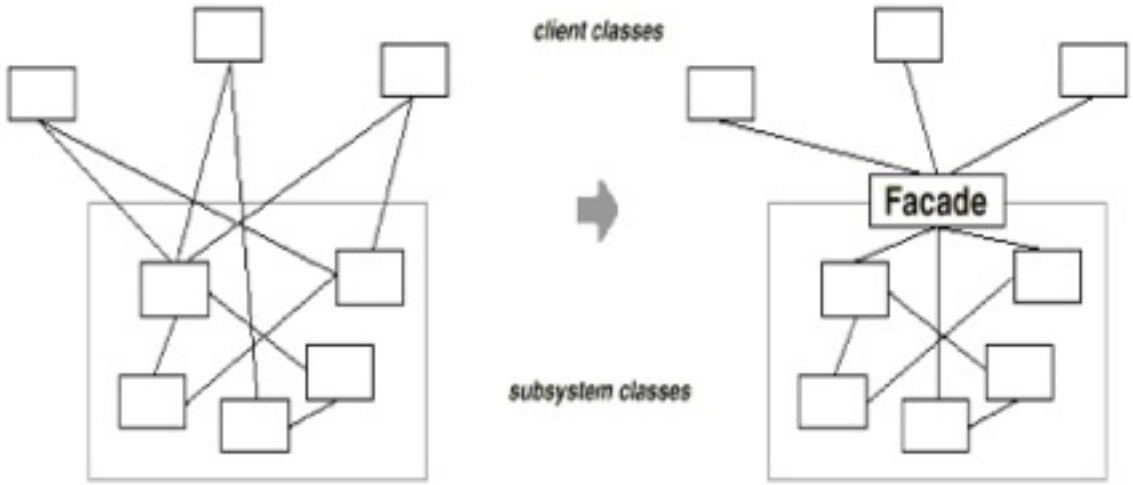
Grunden til f.eks. Mapper-klassernes metoder er deklareret med en protected modifier er, at de udelukkende bliver brugt i dataSource-laget, og derfor ikke skal ses af nogen klasser uden for dette denne pakke. Klasser placeret uden for denne pakke, skal udelukkende kende til facade-klassen. Dette har vi valgt at gøre, for at overholde konceptet med "Encapsulation", og for at forhindre direkte brug af metoderne i andre lag. Vi kunne også have givet de pågældende klasser, i dette tilfælde Mapper-klasserne, en "default-modifier" i klasse-deklarationen. Dette ville have forhindret en mulig instantiering i et andet lag. Vi valgte dog at lade være, grundet det fælles overblik af programmet.

Facadens formål

Tag udgangspunkt i eksemplet nævnt i forrige afsnit, vedr. en webløsning af programmet. Hvis præsentationslaget skulle udskiftes, og man skulle kalde de samme funktionaliteter som før i domænelaget, men domænelaget ikke havde en klasse der samlede lagets funktionaliteter, ville

[1] Lethbridge & Laganiére - p. 347 - 349, "The Multi-Layer architectural pattern"

processen være mere tidskrævende end nødvendigt. I stedet for man skulle checke flere klasser igennem, for at finde den ønskede metode du har brug for at kalde, behøves man udelukkende at kalde på domænelagets facade. `Controller` fungerer som facade for domænelaget i programmet, og al lagets funktionalitet tilgås gennem denne klasse. Derudover giver facaden også adgang til det nedenstående lag's funktionalitet. I programmet skal man f.eks. gennem `Controller`-klassen i domænelaget, for at få adgang til dataSource laget's funktioner.
Endvidere kan det siges, at vi som udviklere ikke ved, om det er os selv eller andre udviklere, der kommer til at stå for webløsningen i fremtiden, eller om man på et tidspunkt ønsker at redesigne programmet's GUI. Her vil brugen af facade-mønstret være en fordel. Du får som udvikler hurtigt et overblik over programmets kode, da al kommunikation til laget under, sker gennem én klasse. Én af mange fordele ved facade-mønstret er netop, at det er nemmere at bruge underliggende lag's funktionalitet: "*It hides the implementation of the subsystem from clients, making the subsystem easier to use*"[2]. Derudover slipper man for at have flere instantieringer i præsentationslaget, af klasser i domænelaget, hvilket resulterer i en løsere kobling.



Figur 12 - Tarr Bob, p. 3 - "*Design Patterns in Java*"

Singleton

Der kan opstå situationer, hvor man ønsker at eliminere muligheden for, at have mere end én instans af en given klasse. Dette kan opnås ved at implementere Singleton-mønstret. For at tage udgangspunkt i vores program, har vi valgt at implementerer Singleton i to af vores klasser; `DbConnector` og `DbFacade` i dataSource-laget. Ideen er, at mens klienten kører programmet, arbejder vi altid med den samme forbindelse, til database. Derfor har vi kun brug for én instans af `DbConnector`, da det er denne klasse der sørger for, at oprette forbindelse til databasen. Derfor mente vi, at det var hensigtsmæssigt, at gøre netop en klasse med denne type ansvar til Singleton. Som konkret argument for brug af mønstret Singleton, er klassen `Chatty` vores eksempel. `Chatty` indeholder ArrayLister navngivet: bookings og customers. Disse lister er kopi af data fra

[2] Bob Tarr - p. 6, "*Design Patterns in Java*"

databasen, og bruges hver gang man i programmet skal søge efter reservationer. Du søger dermed op mod de nævnte ArrayLister på java niveau, i stedet for at søge på databasen hver gang. Hvorfor vi har valgt at gøre dette, uddybes i afsnittet “Chatty, Chunky eller Unit of Work”. Ved multi instantiering vil der komme risiko for dubletter af data, hentet fra databasen, som kan være anledning til en række fejl. Dette umuliggør vi netop ved, at gøre klassen `DbFacade` til singleton, fordi `Chatty` udelukkende instantieres gennem `DbFacade's` konstruktør.

```
private DbFacade() throws ConnectionException {
        this.chatty = new Chatty();
        con = DBConnector.getInstance().getConnection();

        CronThreader c = new CronThreader(con);
        new Thread(c).start();
    }
}
```

Kodeudklip 5 - lag: dataSource, klasse: DbFacade, Linje: 24-30

Kigger man i programmet's klasser kan man se, at `Dbfacade` kun instantieres i `Controller`-klassen i domænelaget. Dog sker DbFacade's instantiering i `Controller's` konstruktør, men `Controller`-klassen instantieres flere gange, og havde DbFacade ikke været Singleton, ville vi have flere instanser af `DbFacade` og `Chatty` liggende i heap'en(dermed flere lister). Dette ønsker vi netop ikke, da vores entiteter fra databasen ligger i `Chatty`.

Måden hvorpå vi implementerer Singleton, sker på klasseniveau. Det er den samme teknik vi bruger, i både `DbConnector` og `Dbfacade`. Kodeeksemplet nedenfor viser hvordan vi gør klassen `DbFacade` til en Singleton:

```
private static DbFacade instance;
```

Kodeeksempel 6 - lag: dataSource, klasse: DbFacade, Linje: 22

```
public static DbFacade getInstance() throws ConnectionException {
        if (instance == null) {
            instance = new DbFacade();
        }
        return instance;
    }
}
```

Kodeeksempel 7 - lag: dataSource, klasse: DbFacade, Linje: 32-37

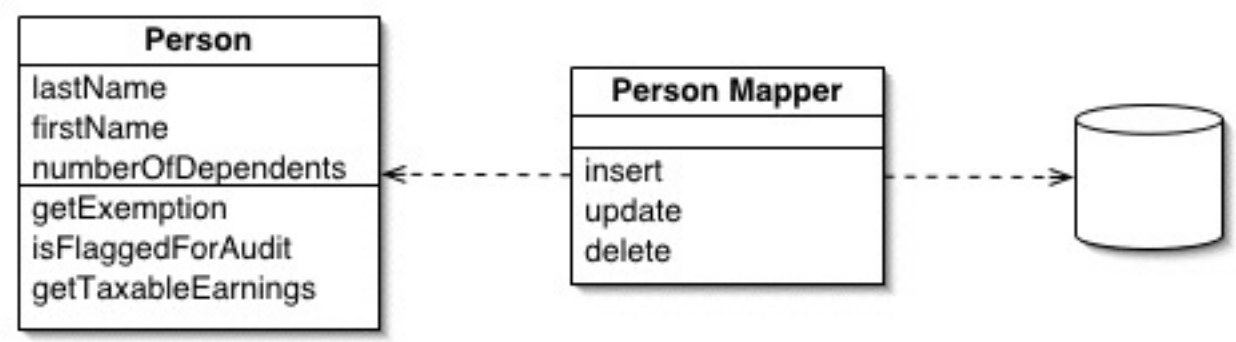
I kodeudklip 7 på forrige side kan det ses, at `DbFacade's` konstruktør bliver sat til private. Man kan dermed ikke instantiere klassen uden for klassen selv. Derfor laver vi en metode, der hedder getInstance, som vi sætter til at være static. Static fordi, vi dermed kan tilgå metoden på klassen, uden at instantiere den. Bemærk at vores objekt der returneres fra getInstance, ligger som attribut, er af typen `DbFacade` og er kaldet "instance". Når man vil instantiere klassen, skal man tilgå metoden, og det er op til metoden at kalde klassens private konstruktør, og returnere objektet. Hvis instance er null, betyder det, at klassen ikke er instantieret endnu. Den konstruerer dermed objektet, og returnerer det. Hvis objektet derimod findes på heap'en, returnerer den `DbFacade`-objektet, uden at kalde kontruktøren igen. Dermed vil der altid forekomme maks én instans af denne klasse.

Data Mapper

Programmer der snakker med en database, skal kunne flytte data fra programmet selv, ned på databasen og hente det igen. Denne proces skal håndteres hensigtsmæssigt. En database gemmer/behandler f.eks. data helt anderledes, end et program skrevet i java gør. For at udføre denne process på en måde der giver mening for os, har vi valgt at implementere "Data Mapper"-mønstret.
Dette mønster er klasse placeret i et lag kaldet datasource, som klarer al kommunikation med databasen, dvs. redigere, indsætte, slette og anden manipulation med databasen data. Da det er to vidt forskellige fremgangsmåder at arbejde med data på, forsøger vi med dette mønster at separere vores objekter i programmet fra databasen, og dermed isolere programdelen og databasen fra hinanden:
"*The Data-Mapper is a layer of software that seperates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other*"[3].
Klasser der i vores program vedrører dette mønster er klasserne; `BookingMapper`, `GuestMapper`, `ApartmentMapper` og `HistoryMapper`. Der er en mapperklasse per entitetsklasse.

Alle de måder hvorpå vi ønsker at arbejde med den pågældende data på databasen, sker gennem disse klasser. Normalt hvis du vil oprette en tuple i en tabel på en database, vil du i SQL, bruge en insert-kommando. Denne kommando vil ifølge dette mønster, ligge i en mapper-klasse, tilhørende den gældende entitet (se figur 12, på næste side).

[3] Fowler Martin - p. 165 "*Patterns of Enterprise Application Architechture*"



Figur 13 - Fowler Martin - p. 165, “Patterns of Enterprise Application Architecture”

Vores program følger den samme ideologi, som vist i figur 2. F.eks. har BookingMapper-klassen flg. metoder: getAllBookings, insertNewBooking, updateBooking og deleteBooking. Klassen indeholder alle de metoder der tillader os, at tage vores objekt, gemme det på databasen, ændre det og/eller slette det. Vi kan altså manipulere med dataen på databasen og indsætte ny data. Modsat tillader klassen samtidig, at vi kan få det returneret igen. I dette tilfælde ville man kalde metoden getAllBookings, for at få alle reservationer tilbage som objekter.
Et konkret eksempel på hvordan vi gør dette, kan ses nedenfor et kodeudklip fra førnævnte metode; deleteBooking. Metoden’s navn bekriver metoden’s funktionalitet; at slette en booking.

```
protected int deleteBooking(Connection con, int b_id) throws BookingException {
        int rowDel = 0;
        String SQLString = "delete from booking where b_id = ?";
        PreparedStatement stat = null;

        try {
            stat = con.prepareStatement(SQLString);
            stat.setInt(1, b_id);
            rowDel = stat.executeUpdate();

        } catch (SQLException e) {
            throw new BookingException("Booking could not be deleted");
        } finally {
            try {
                stat.close();
            } catch (SQLException e) {
            }
        }
        return rowDel;
    }
```

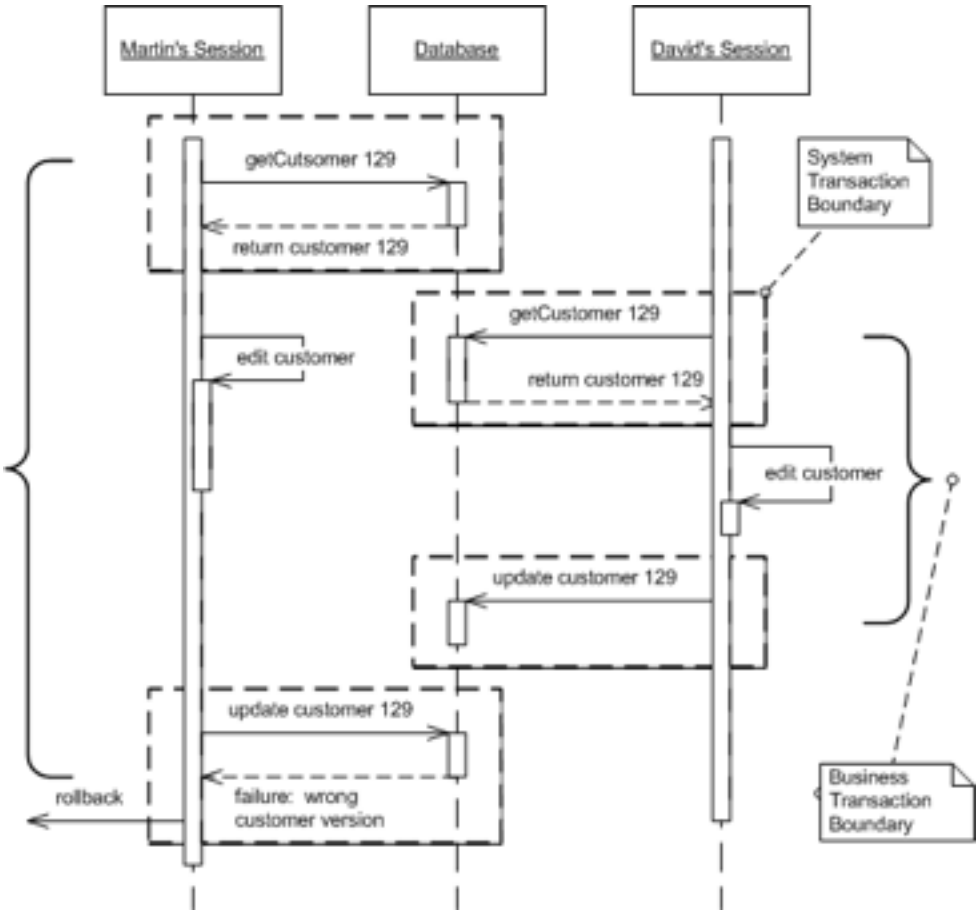
Kodeeksempel 8 - lag: dataSource, klasse: BookingMapper , Linje: 32-37

For at slette den ønskede reservation på databasen, skal vi skrive kommandoen ud i SQL. Bemærk at vi i metoden på linje 4 formidler SQL ind i en String. Vi har importeret JDBC API'et, der giver os funktionalitet til at snakke med databasen.
Vi skriver stadig som vi ville gøre i SQL, men med det importerede API's funktionalitet, kan vi bruge et PreparedStatement objekt til at udføre kommandoen. Efter vi har deklareret objektet, og sat det til at være lig med con.PreparedStatement, kan vi bruge metoden setInt. Denne metode tager det første "?"(spørgsmålstegn) i vores SQL-String, og sætter det til at være det pågældende booking ID, der kommer med som input-parameter i metoden. Bemærk at en booking's primærnøgle på databasen, er booking ID'et. Input parameteren b_id gør derfor, at udelukkende en booking med dét bookingID bliver slettet fra Booking-tabellen. For at eksekvere vores PreparedStatement, bruger vi metoden executeUpdate. Denne metode returnerer et heltal, der svarer til antallet af påvirkede tupler. Hvis den returnerer 1, ved vi at en tupel er blev slettet, returneres 0 blev ingen tupler slettet. Til sidst kører vi en close metode på vores Connection objekt, der frigiver de JDBC og database-ressourcer der blev brugt til dette PreparedStatement. Dette gør sig gældende for alle metoder i vores program, der har med database kommunikation at gøre.

Pointen i at lægge al programmets SQL under samme lag, er at overholde 3-lags modellen ved at placere Mapper-klasserne under dataSource laget, samles klassernes relaterede ansvarsområde det samme sted. Hver gang vi skal gemme et `Booking` objekt på databasen ved vi, at vi skal bruge `BookingMapper`-klassen, hver gang vi skal gemme et `Apartment` objekt på databasen ved vi, at vi skal bruge `ApartmentMapper`-klassen osv.
Derudover opnår vi et af Mapper-mønstret's store fordele; at isolere entitetsobjekter og databasen fra hinanden. Objekter i vores program har ingen kendskab til databasen, og ej noget kendskab til SQL. Dette er en fordel af flere årsager. En af mange kunne være, at man fra kunden's side fik et behov for at skifte til et andet DBMS(Database Management System). Programmet arbejder på nuværende tidspunkt med en Oracle-database. Hvis kunden skulle skifte til f.eks. en MySQL-database, skal programmet's SQL-kode skrives om, da hvert DBMS snakker sin egen version af SQL. Denne ændring er en mere simpel proces, hvis al SQL-kode er samlet ét sted.

Optimistisk Offlinelock

Dette mønster er relevant i forhold til manipulation af data på databasen, og registrering af ny data. Vi er nødt til at sikre os, at den data der ændres, ikke går tabt, og ny data ikke interferere med anden ny data.
En receptionist i programmet prøver at ændre en registreret kundes telefonnummer. I programmet navigerer man ind under *Edit Booking* siden fra menuen på forsiden, og finder kunden. Her ændrer receptionisten kundens telefonnummer til det nye, og trykker på knappen *Save*. Hvis ingen begrænsinger på databasen, constraints, bliver brudt, burde dette gå igennem uden problemer.



Figur 14 - Fowler Martin - p. 416, "*Patterns of Enterprise Application"*

Problemstillingen vi skal tage hensyn til, bliver relevant så snart samtidighed er aktuelt. Der er fire terminaler i receptionen, så fire brugere kan tilgå databasen på samme tid. Tag udgangspunkt i eksemplet ovenfor med kunden's telefonnummer. Vi antager, at hver gang en receptionist gemmer en ændring, foretages et commit på databasen. Samme situation opstår mens flere receptionister betjener programmet, fra hver deres computer. Receptionist 1 ændrer kunden's telefonnummer, og receptionist 2 ændrer kunden's adresse. Receptionist 1 gemmer det ændrede telefonnummer, mens receptionist 2, der ændrer adressen, stadig sidder med det gamle telefonnummer, og ikke har gemt endnu. Receptionist 2 har nu ændret adressen, og trykker på "Save". Alt hvad receptionist 2 arbejdede med bliver gemt, fordi hun er den sidste der gemmer. Fordi receptionist 2 sad med det gamle telefonnummer, bliver dette nu også gemt på databasen. Kunden's nye telefonnummer blev gemt da receptionist 1 gemte, men blev overskrevet med gammelt data, da receptionist 2 gemte. Kunden's telefonnummer er altså det gamle igen.
Denne situation kaldes en "lost update"[4], og er noget vi til enhver tid vil forsøge at undgå. I eksemplet ovenfor vil receptionist 1 tro, at kunden's telefonnummer er ændret og gemt til det nye, selvom det blev overskrevet da receptionist 2 gemte.

[4] Fowler Martin - p. 165 l. 5, "*Patterns of Enterprise Application Architecture*"

Måden hvorpå vi vælger at løse denne problemstilling, er ved at lave en Optimistisk Offlinelock. Hensigten er at undgå dataændring i en session, kommer i konflikt med dataændring fra en anden. Vi skal implementere en teknik hvor vi kan registrere om det data man prøver at ændre, er blevet ændret af en anden i mellemtiden. Hvis dette er tilfældet, skal du genindlæse, og dermed få det nyeste ind fra databasen. På denne måde vil vi undgå at miste vigtig data.
Løsningen fremgår bl.a. af vores relationelle skema (se side). Tabellerne Booking og Customer, har begge en attribut der hedder VERSION_NUM. Denne attribut er et heltal. Tallet er et check vi bruger, hver gang vi ønsker at ændre(Update) en tuple. Tallet's værdi bliver automatisk øget med én, hver gang en ændring fra programmet er gået igennem, og er blevet gemt på databasen.

Når du fra programmet går ind og ændrer en kunde's information, henter du først kunden op fra databasen. Her får vi bl.a. fat i det versionsnummer, for den pågældende tuple i databasen. Når man prøver at gemme ændringerne for kunden, skal versionsnummeret være det samme som da vi fra starten af hentede kunden ind fra databasen. Hvis dette ikke er tilfældet betyder det, at en anden receptionist har ændret i netop denne tuple, fordi versionsnummeret har ændret sig, inden man selv nåede at gemme ændringerne. På denne måde undgår vi f.eks. at miste det nyeste telefonnummer der blev sat ind, fordi receptionist 1 gemte ændringerne, og versionsnummeret for reservationen blev talt op i tabellen på databasen. Se nedenstående model, der illustrerer lignende situation. Som sagt undgår vi med denne metode en lost update-situation. Ulempen ved denne fremgangsmåde kan dog være, hvis en en receptionist sidder og laver ændringer, der tager længere tid at skrive ind, før hun kan trykke "Save". Hvis dette er aktuelt, og en anden ændrer samme booking og commiter, er hun nødt til starte forfra.

At bruge dette mønster var en designbeslutning, vi som gruppe blev enige om at tage, da ovenstående tilfælde kun vil være aktuelt for én booking ad gangen. Sådan som vi har designet vores program, kan receptionisten ikke ændre flere reservationer ad gangen. Hun er nødt til at gemme/tjekke pr. reservation hun vil redigere, så hvis versionsnummeret ikke er det samme, og hun skal starte forfra, er det begrænset hvor mange ting hun skal skrive ind igen.
En anden løsning er pessimistisk offlinelock[5]. Dette er et mønster hvor intentionen er den samme; at undgå data tab når flere prøver at ændre den samme tuple. Med dette mønster sætter man den pågældende tuple i et "lock-mode"; exclusive. Den receptionist der først begynder at redigere i reservation 1, låser med det samme den pågældende tuple på databasen. Dette betyder at ingen anden kan redigere reservation 1, før receptionisten trykker *save* eller *cancel*. Dermed er man sikker på, at den reservation man redigerer, vil have det samme versionsnummer som da man startede, fordi ingen anden ville kunne ændre reservationen, så længe den er låst.

---

[5] Fowler Martin - p. 426, "*Patterns of Enterprise Application Architecture*"

Grunden til vi ikke valgte denne løsning var, at vi “kun” har fire terminaler der bruger programmet/databasen. Chancen for, at én reservation vil blive redigeret af flere receptionister på samme tid, antager vi er minimal. Derudover kunne man forestille sig, at alle fire receptionister sjældent vil stå i receptionen og ændre reservationer på samme tid. Deraf også navnet optimistisk-offline lock; vi antager at ændringer vil gå igennem uden problemer.
Endnu en vigtig pointe der gjorde, at vi fravalgte at bruge pessimistisk offline lock, var muligheden for længere lock-tid på databasen. Da vi under beslutningsprocessen diskuterede dette mønster, overvejede vi hvad chancen kunne være for, at en receptionist tog længere tid om at redigere en reservation end forventet. F.eks. hvis en receptionist der står og ændrer en reservation, og får dermed låst den pågældende tuple, pludselig forlader sin terminal. Den pågældende tuple ville forblive låst, indtil der blev trykket på *Save-* eller *Cancel-knappen*. Andre ville ikke kunne tilgå den samme reservation i et ukendt tidsrum. Dette undgår vi ved at anvende en optimisk offlinelock.
Det eneste tidspunkt vi låser i databasen, er i det øjeblik vi opretter en booking. Her sætter vi booking-tabellen i et exclusive-mode, der først bliver ophævet når der køres en commit eller rollback. Dette er grundet, at vi da er sikre på, at den ny indsatte reservation enten går igennem, eller slet ikke registreres på databasen og ingen anden kan tilgå databasen imens at ingen anden reservation laves på samme apartment.

Chunky, Chatty eller “Unit of Work”

Klassen `Chatty` var originalt designet efter mønstret Unit of Work. Dette mønster bygger på ideen om, at køre alle ændringer der sker i programmet, ned på databasen over én omgang. Vi undgår med dette mønster, at lave nogen former for databasekald, før der bevidst bliver kørt en commit. Unit of Work-klassen’s ansvar er, at registrere alle ændringer, sletninger og oprettelse af data i programmet, gemme det i klassen’s lister, og registrere det ned på databasen. Hvis transaktionen indsætter en fejl, bliver der kørt en rollback, så databasen ender på den samme tilstand som før commit blev kaldt. I Unit of Work er det alt eller intet.
Denne process medfører at der går længere tid imellem hvert commit, og vores data i listerne kan være forældet. Dette er et problem, hvis f.eks. en receptionist har lavet en reservation for en kunde på lejlighed 1, og en anden receptionist vil lave en reservation på samme lejlighed, to timer efter, uden der i mellemtiden er blevet commited. Begge reservationer vil blive tilladt, da der ingen reservation er på lejlighed 1, ifølge deres lokale lister. Spørgsmålet er, hvor ofte listerne så skal opdateres, men denne opdatering af listerne, kan hurtigt blive et stort databasekald.
Derfor valgte vi, at benytte “micro-transactions”, dette forklarer klassens navn, “mange små database kal - Chatty”. Dette betyder f.eks., hvis en reservation bliver oprettet på apartment 1, bliver reservationen registreret på databasen med det samme. Når en anden receptionist prøver at oprette en booking med identisk startdato, ved hun med det samme, at en anden booking allerede er registreret på lejlighed 1, fordi vi søger direkte på databasen.

Ideen med at søge op mod entitets-listerne, når der skal hentes data, har vi beholdt. Listerne bliver opdaterede, når du skal vælge en lejlighed, ved oprettelse af ny reservation samt når du skal redigere en reservation. Opdateringen garanterer det nyeste data fra databasen.
`Chatty` indeholder de ting vi skal bruge fra vores database; reservationer i listen Bookings, registrerede kunder i listen Customers og alle lejligheder i listen Apartments.
Ideen i at have vores lister er, at vi kan skære ned på brugen af databasens ressourcer. Når vi f.eks. prøver at finde en bestemt kunde i systemet, kalder vi på klassens metode "searchForCustomers"(se kodeeksempel 6). Denne metode søger direkte i listen "customers". På denne måde sparer vi SQL-kode, der skulle søge direkte i Costumers-tabellen og derved databaseresourcer.

```
protected ArrayList<Customer> searchForCustomers(String keyword) {
        ArrayList<Customer> tmpList = new ArrayList();

        for (Customer c : customers) {
            if (c.getFirst_name().toLowerCase().contains(keyword)
                || c.getLast_name().toLowerCase().contains(keyword)
                || c.getPhone().toLowerCase().contains(keyword)) {
                   tmpList.add(c);
            }
        }
        return tmpList;
    }
```

Kodeeksempel 9 - lag: dataSource, klasse: Chatty, Linje: 261-272

Lo fi - modelering

Lo Fi modellen er et mockup af vores brugergrænseflade, som vi forestiller os at den kommer til at se ud. Lo Fi'en vil vi bruge til test af brugergrænsefladens design og brugervenlighed. Når vi når til, at skulle implementere et stykke af brugergrænsefladen, bliver det også nemmere, når vi har et design klar, at arbejde op imod.

Design & Overvejelser

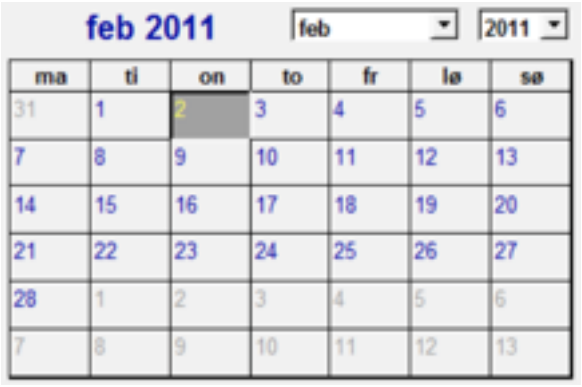
En Lo Fi model var en ekstra god ide at have med, da vores Product Owner prioriterede højt, at produktet skulle have en flot brugervenlig brugergrænseflade, som skulle kunne anvendes af en receptionist uden nogen oplæring. Ligeledes blev der sat meget vægt på at det skulle være hurtigt og nemt at finde informationer for en booking, check in og check ud for en dag og reservationer i en specifik måned. For at gøre det nemt, at navigere rundt, mellem de funktionaliteter programmet skulle have, besluttede vi at have en forside med en sigende knap til hver funktionalitet.
(Se bilag 4 - Lo Fi forside)

For at gøre grænsefladen mindre forvirrende afgrænsede vi funktionaliteter som f.eks. at oprette en ny booking eller redigere en booking til hver deres side. Men grundskelettet ville vi holde så ens som muligt, så en formular der indeholder booking informationer, er ens uanset om receptionisten er i gang med at oprette en ny eller redigere i en gammel booking.
At holde grundskelettet ens over flere sider gør at det ikke vil være nødvendigt at lede efter et bestemt felt, da brugeren hurtigt vil lære layoutet at kende.
(Se bilag 5 - Lo Fi new booking) & (Se bilag 6 - Lo Fi edit booking)

Da receptionisterne der anvender programmet hurtigt skal kunne ekspedere en kunde er det vigtigt at grænsefladen er intuitivt. Receptionisten skal ikke selv udfylde mere end højst nødvendigt, programmet skal klare resten. Det skal være nemt at få store mængder information ud af programmet på en gang f.eks. ville vores Product Owner gerne have muligheden for at se booking information for alle reservationer der starter eller slutter på dags dato. Dette valgte vi at løse ved at oprette en grafisk blok visning.(Se bilag 7 - Lo Fi list booking today). Blok visningen fungere ved at repræsentere aller reservationer i farvede blokke. Alle dagens check ins vises i grønne bokse og alle check outs vises i blå bokse. Klikkes Der på en boks vises en popupboks med alle den valgte booking's informationer. Ved at gøre det på denne måde, er det nemt for en bruger, hurtigt at checke informationer omkring en kunde, som skal forbi i receptionen i løbet af dagen.

Vores product owner nævnte at han gerne ville kunne se alle reservationer indenfor en søgt måned. Vi har valgt at vise dette grafisk da det giver et bedre overblik for receptionisten en hvis vi

f.eks. bare listede reservationerne op i en liste. Vi brugte lang tid på at analysere og diskutere forskellige måder at tegne en måned grafisk i vores grænseflade. En af overvejelserne var at tegne en klassisk kalendervisning(se nedenstående figur).



Figur 15: kalender eksempel - google.com

Dette gik vi dog væk fra igen da det ville blive for rodet/stort. Hvis alle reservationerne på en dag skulle placeres indenfor samme felt, er der et potentiale for 104 overlappende reservationer på samme dag. Vi fandt i stedet på, at tegne alle månedens dage vertikalt og lade booking blokke løbe horisontalt fra bookingens start til slut dato(Se bilag 3 - Lo Fi show month). På denne måde er det let for receptionisten hurtigt at finde en booking, danne sig et overblik over igangværende reservationer eller se hvor travl en fremtidig måned potentielt bliver.
For at holde konventionen om genanvendelighed, vil vi genbruge ideen med at vise reservationer i blokke, hvor der ved klik vises en pop up med bookingens info.
Det var vigtig for vores Product Owner at programmet nemt kunne udvides. Ved at holde forsiden simpel med en enkelt knap til hver funktionalitet har vi sørget for at der er plads til senere tilføjelser af flere sider/funktionaliteter (Se figur 2: Program - front page, afsnit: beskrivelse af programmet).

Vi har for at fjerne arbejde fra receptionisten tænkt meget på hvordan vi kan sikre os unødig fejl. Vi har derfor brugt tid på at kigge alle tekster og placeholders til felter igennem, for at sikre os at de er sigende.

Sammenhæng mellem Lo - Fi og Sporbarhedsmodel

Under hele Lo Fi designprocessen har vi hele tiden haft vores sporbarheds model i baghovedet. Hvordan kan vi bedst vise disse features grafisk.
Vi viser status af alle værelser under BookingList Se Billede. Kunderegistrering sker under NewBooking (Se figur 3 - new booking page). Der er funktionel søgning under både Previous Customer & Edit Booking(Se figur 4 - previous customer page) & (Se figur 5 - edit booking page)

Test af Lo Fi

Vi testede vores Lo Fi model på en elev uden for gruppen. Feedbacken vi fik

Plusser

Nemt at anvende: jeres program er nemt at navigerer gennem, jeg var på intet tidspunkt i tvivl om hvor jeg skulle hen for at opfylde en opgave.

Flot: blokstrukturen er en god ide til at vise meget information på et begrænset område.

Sigende tekst i grænsefladen: jeg vidste hele tiden hvad der skulle skrives på hvilke felter i formularene

Minusser

Intetsigende fejlmeddelelser: i brugte samme fejlmeddelelser til flere felter hvilket var forvirrende.

Der var for mange vinduer: jeg ville bedre kunne lide at hvis noget af funktionaliteten kunne samles.

Efter test refakturerede vi designet så der blev kastet sigende fejlmeddelelser. Derudover samlede vi Edit Booking delen til en side istedet for 2. (se bilag 8 - editBookingNew)

Testning - Brug af JUnit Test

Struktureret kode læsning

Efter færdiggørelse af programmet, startede vi på struktureret koderefakturering. Vi søgte efter tastefejl og andre fejl, specielt fordi Java er case sensitiv. Vi har at gøre med forskellige entiteter i vores klasse; reservationer, kunder og lejligheder, så det er derfor vigtigt at de alle bliver initialiseret med de korrekte værdier. Loops er specielt en stor fejlkilde, da man ofte misser det første og/eller det sidste komponent af en collection. Ligesom loops, så er boolean-udtryk også giftige; derfor gennemtjekkes logiske operatore for fejl. Vi har gjort brug af muligheden for at sætte det færdige kode op på storskærm, hvilket medfører flere input til metoder der kan simplificeres og forbedres, hvilket kan udrette fejl. Det er vigtigt at mange forskellige øjne på koden, når der refaktureres[6].

JUnit test

Det er vigtigt at teste den kode man producerer undervejs i forløbet, så man kan udrette fejl der kan opstå, og være klar på en uforudsigelig situation i programmet. Ved brug af tests, kan man bekræfte hvorvidt logikken i programmet virker korrekt. Brugen af en skudsikker test indebærer korrekt sammensatte tests af alle vigtige komponenter i vores program. Det er vigtigt at teste et passende stykke kode, så der ikke bliver testet for meget logik per individuelle test. Hvis tilfældet opstår, bliver det svært at sætte en finger på præcis hvilken del af koden, der foresager en bestemt fejl. Derfor benytter vi os af metode testning. Heraf har vi udvalgt én essentiel metode fra Chatty klassen; createNewBookingTransaction(p).
Testresultaterne er afhængig af testerens indstilling til, hvordan vedkommende forventer metodens resultatet kommer til at se ud. Forventer personen at testen i første omgang går rent igennem, er der tendens til at blive skabt tests på komponenter, der ikke skaber fejl i programmet. Den sidstnævnte er den vi stræber at følge.
Man skal fokusere på, at testning ikke er en konstruktiv proces, til at programmørerne kan få en bedre mavefornemmelse. Det skal anses som, at være en destruktiv proces, hvor det forventes at der er fejl i ens test case. Vi testede på følgende:

- Programmet forventes at indeholde fejl.
- Testning er lavet til at finde fejlene.
- En test skal have et forventet resultat.
- En test er succesfuld hvis den finder en ny fejl, ellers er den ikke.
- Der skal testes med gyldig og ugyldig data.

---

[6] "Tremblay & Cheston - “Data Structures and Software Development” s. 671-704"

Selvom vi kan forvente at en test vil give fejl, hvis der testes med ugyldig data, gør vi det alligevel. Netop for at se kontrasten af resultaterne, når man også tester med gyldig data. Dokumentationen på, at ugyldig data ikke kan bruges frem for gyldig data giver vores test case et mere troværdigt resultat.
I NetBeans, hvor vi har kodet programmet, kan man gøre brug af JUnit Tests. Man stiller en testcase op, kører testene og man modtager herefter feedback på eventuelle fejl, samt hvor de opstår i testklassen. Den måde vi unittester på i en JUnit Test Case, kan rettes imod en testteknik kaldet Black Box Testning. Testeren ser vores program som en sort boks, hvor vi her ville fokusere på at oprette en ny booking, med en række af forskellige tilfælde. Vi fokuserer på, at vi kender det input vi indsætter ned på databasen og forventer objektets tilstand bagefter.
Testene skal køres hver gang, der er foretaget betydelige ændringer i programmet, så programmet kan blive fejlfrit hele vejen igennem.
Processen starter, ved at oprette testklassen for de klasser der skal testes op imod, lokaliseret i pakken “Test Packages”. Vores klasse “ChattyTest” indeholder testmetoder, som hver består af en JUnit test.
I samme pakke, er der oprettet en ”Fixture” klasse. Klassen indeholder en række af metoder, som bliver kaldt, til at oprette de nødvendige test tabeller vi har i databasen, samt andre metoder med kendte SQL-queries. Dette medfører, at den database vi tester metoderne på, er i en tilstand vi kender, hvilket er en af nøgleprincipperne for vores test. Den er konstrueret til at indeholde metoder, der altid indsætter det samme data tabellerne.
Metoden, placeret i testklassen, med annotationen; “@Before” har en afgørende betydning for, at vores JUnit tests kan fungere korrekt. Den indeholder metodekald fra Fixture, som sørger for at blive kaldt inden hver JUnit test-metode bliver kørt. Dette er essentielt, fordi JUnit Tests bliver kørt i tilfældig rækkefølge, og vi vil derfor sikre os, at alle tests bliver kørt uafhængigt af hinanden, op mod en database, der er i det samme stadie.
Tests op imod en database skal gøres med forsigtighed, så vi har undladet at teste på vores primære database, men sat en ekstra database op, som udelukkende bruges til test. Den kører på det identiske script der bruges på vores primære database, derfor de samme tabeller. Med denne løsning, kan vi stole på, at testen bliver ligeså troværdig, som var der testet på den primære.
Hensigten var udelukkende at fokusere på, at teste createNewBookingTransaction, men for at vi kan tjekke om vores forventede resultat af en bookings tilstand holder stik, skal vi bruge updateLists(p), som opdaterer ArrayListen med reservationer i Chatty, samt getBooking(p), som returnerer en enkelt booking, baseret på et ID, derfor disse to metoder også inddrages i testen.
Ved at vi inddrager tests for de beskrevede metoder, øger vi troværdigheden af testresultaterne.
Når vi Black Box-tester, vil vi teste op imod scenarier som arbejder med værdier, der tester hvordan programmet agerer situationer det skulle være forberedt på.

Figur 16 - Model af processen der forløber sig i Black Box Testning. Konstrueret af gruppen.

Test af createNewBookingTransaction()

I stedet for at oprette en booking der varer 7 dage, kan man derimod prøve at oprette en booking på nul dage eller derunder, samt en booking der varer én million dage. Dette burde give fejl, hvilket er det vi stræber efter at finde. Foruden de reservationer, som fixture indsætter ind i databasen, så vil vi lave et nyt Booking-objekt i test-metoden, som kalder createNewBookingTransaction(p). Vores metode returnerer en boolean status, hvorvidt det lykkedes eller ej, hvilket er fint at teste op imod. Metoden vi har dokumenteret testcases for, updateLists(p) og getBooking(p), bruges til at hente den samme reservation, som vi lige har konstrueret lokalt, for da at teste om den findes. Vores BookingMapper-klasse øger det forrige højeste bookingID med 1, når vi opretter en ny reservation. Da vi kender bookingID'et på det højeste, søger vi efter en booking med det samme bookingID plus 1.

Test Cases: vi laver en reservation med attributter der ligger tæt på grænsescenarierne for hvad vores database kan indeholde. Vi laver en reservation med attributter der har negative værdier; en gyldig og en ugyldig dato. Ugyldig som var der mere end 31 dage på en måned. Vi laver en gyldig reservation, og tjekker om den er identisk med den vi oprettede, efter den blev hentet tilbage fra databasen. Derudover opretter vi nogle, der overlapper hinanden.

Det relationelle skema

Det relationelle skema er en måde at vise strukturen i databasen meget simpelt.
Vi lavede oprindeligt et relationelt skema, så vi havde en idé om hvordan vores database skulle se ud. Siden hen har vi fuldstændig genopbygget vores database, baseret på det nuværende relationelle skema, efter det viste sig at være en nødvendighed.

Hver tabel er vist med dens navn efterfulgt af dens attributter i parentes. Attributterne er vist på følgende måde;

Attribute	Attributten er uafhængig
Attribute	Attributten er en foreign key
Attribute	Attributten er en primary key

Database struktur

Vi besluttede at vores database skulle have 5 entiteter:

- Apartment
- Booking
- Customer
- Guest
- History

Apartment blev inkluderet sådan at alle lejligheder også lå i databasen, da det vil gøre hele programmet meget simplere at lave. En lejlighed indeholdt originalt kun info om dens nummer, hvor vi så også havde en entitet det hed apartment_type, hvor der lå information om priser og gæste kapacitet, men denne blev senere droppet, da vi besluttede os for kun at en entitet relateret til en lejlighed.

Booking er den mest essentielle af vores entiteter, da det er reservationer som alle andre entiteter hænger sammen med, og da det at kunne gemme reservationer er hovedefunktionen af hele vores program.

Customer gemmer på information for en enkelt kunde, som så er tilknyttet en reservation. Customer holdes separat fra booking, da tidligere kunder skal kunne hentes frem igen, og at ligge dem ind i en booking, vil skabe unødvendige dubleter.

Guest vil gemme en id for enhver person, som er tilknyttet til en booking. Guest blev lavet som forberedelse til implementering af sportsfaciliteter, men da den feature blev droppet til fordel for at forbedre vores eksisterende features, bliver guest ikke brugt til noget.

History er en separat entitet der indeholder gamle reservationer. Vi valgte at holde booking entiteten udelukkende for nuværende eller fremtidige reservationer, og så flytte gamle reservationer til history med et automatisk cronjob.
Se Bilag 15 for databasen E/R model

Normalisering af databasen

Normalisering af databaser går ud på, at dele tabellerne i en database op i mindre tabeller, da det vil medføre bedre overskuelighed over hvilke entiteter der arbejdes med. Ved normalisering af et relationelt skema, stræber man efter at udarbejde en passende mængde af tabeller, så entiteternes attributter udelukkende kun beskriver hvad entiteten er kendt for. Hvis man opdager, at en tabel i sit relationelle skema, har en til flere attributter, der ikke kan beskrive entiteten nøjagtig, er der ikke blevet normaliseret optimalt[7].
Vores relationelle skema er listet herunder:

Relationelt skema
apartment(a_num, type, price, capacity)
booking(b_id, cust_id, a_num, date_from, number_of_nights, travel_agency, number_of_guests, price, version)
customer(cust_id, first_name, last_name, phone, email, country, city, zipcode, street, version)
guest(b_id, guest_id)
history(b_id, cust_id, a_num, date_from, number_of_nights, travel_agency, number_of_guests, price)

Vi har gjort os en del overvejelser om hvor meget vi skulle normalisere vores tabeller, men vi mente at de ville blive for overflødigt at have mere end fem tabeller. Selvom vi har valgt denne opsætning, kan vores tabeller normaliseres yderligere, ved brug af de tre mest normale normaliseringsformer; 1NF, 2NF og 3NF. Man kan bruge normaliserings formerne på at måle hvor meget en database er blevet normaliseret.

1NF - Vi kunne eksempelvis have tabeller der så således ud;

apartment(a_num, type, price, capacity)

[7] Kilde - sams teach yourself SQL

hoteltransaction(b_id, cust_id, guest_id, a_num, date_from, number_of_nights, travel_agency, number_of_guests, price, first_name, last_name, phone, email, country, city, zipcode, street, version)

Disse tabeller har allerede gennemgået første normale form. Førhen lå alle attributterne i en stor samlet hotel_database tabel, men er nu blevet delt op i mere overskuelige tabeller. Dataen blev brudt ned til logiske enheder, som har relateret information. For at sikre os, at alt ser fint ud, så har hver tabel en primærnøgle og der er ingen dubletter af attributter.

2NF - Målet for anden normal form, er at finde attributter der kun er lidt relaterede til primærnøglen i tabellen, og forsøge at indsætte dem ind i endnu en tabel. I hoteltransaction er der informationer som kan relateres til en guest, så den kan indsættes ind i sin egen tabel. Vi ændrer "hoteltransaction" til "booking". Resultatet af anden normale form ser således ud:

apartment(a_num, type, price, capacity)
booking(b_id, cust_id, guest_id, a_num, date_from, number_of_nights, travel_agency, number_of_guests, price, version)
customer(cust_id, first_name, last_name, phone, email, country, city, zipcode, street, version)

Vi har nu en booking-tabel der indeholder en række vigtige informationer, kun fokuseret på en booking, samt en customer-tabel, der indeholder de nødvendige informationer for hvad en kunde består af. Det var muligt at splitte hoteltransaction op i to tabeller, da attributterne til en gæst ikke er lige så relaterede til primærnøglen b_id, som date_from eller number_of_guests er.

3NF - Målet for den tredje normale form er at flytte data som ikke er direkte afhængig af den oprindelige primærnøgle, men også er relaterede til hinanden.
Vi kan udplukke phone, email, country, city, zipcode og street, og indsætte dem ind i en separat tabel med navnet address_table, da de alle er relaterede til hinanden. Primærnøglen er email. Resultatet ses nedenfor.

apartment(a_num, type, price, capacity)
booking(b_id, cust_id, guest_id, a_num, date_from, number_of_nights, travel_agency, number_of_guests, price, version)
customer(cust_id, first_name, last_name, version)
address_table(email, phone country, city, zipcode, street)

Reflektion over normalisering

Vi har valgt at tilføje yderligere to tabeller til vores relationelle skema, fordi det gjorde arbejdet lettere for os. Vi kan se, at normalisering giver en overskuelighed over database, men vi valgte

ikke at normalisere vores database til tredje form. Databasen kan også handle hurtigere, jo færre tabeller den indeholder, da det kræver mindre Database-kraft at trække information ud af få tabeller. Vi har foretaget en denormalisering af vores tabeller, under udvikling af det relationelle skema. Førhen havde vi apartment(a_num, type) og apartment_type(type, price, capacity), som vi nu har kombineret til apartment(a_num, type, price, capacity), netop fordi vi ville øge hurtigheden af forespørgsler i databasen og da vi kun har 104 lejligheder at arbejde med, hvilket ikke giver en stor belastning på databasen.

Vi tilføjede guest-tabellen, da vi fandt ud af, det var et lovkrav, at gemme alle de kunder i en tabel, hvis mindst et uheld skulle forekomme. Vi tilføjede yderligere en tabel, history, som skulle bruges til at gemme alle de gamle gæster der har boet på hotellet. Data kan bruges til et væld af muligheder, så hver gang en booking slutter, bliver den slettet i booking-tabellen og flyttet hen i history-tabellen.

Konklusion produkt

Der var en række krav til vores program, som blev stillet af vores Product Owner, såvel som opgave oplægget.
Det ene af kravene, var at man skulle kunne oprette og gemme reservationer, baseret på en eksisterende registreringsform. Systemet skulle fungere på minimum fire computere, på en gang. Systemet skulle relativt nemt kunne gøres web-baseret, da der er planer om at udvide systemet på et senere tidspunkt. Programmet blev fuldt ud lavet til at tage højde for alle disse krav, og alt nødvendig funktionalitet til dette, er blevet implementeret deri.
Det andet krav, var at it-systemet også skulle kunne håndtere sportsfaciliteter, og booking af instruktører. Det skulle gøres så enkelt at gæsterne selv kunne lave reservationer i opsatte terminaler på resortet, og at systemet kunne håndtere unikke id'er til alle resortets gæster. Vi foretog små forberedelser for at kunne implementerer disse funktionaliteter, men i sidste ende, på grund af begrænset arbejdstid, valgte vi at putte alt vores fokus på at finpudse det første krav. Vores program lever derfor ikke op til dette krav.
Det tredje krav til vores program, var at vores brugergrænseflade skulle være simpel, informativ, og brugervenlig. Det kan vel siges, at der ikke er noget program der er designet uden at tage højde for dette på en eller anden form, men vi har dog valgt at gøre en ekstra indsats på dette område, med listvisninger, forklaringer og rigeligt med feedback til brugeren. Da dette krav er meget vagt, er det svært at sige om vores program lever op til det. Vi vil dog erkende at vi, med vores store arbejde på at gøre programmet brugervenligt, såvel som meget positiv feedback fra vores Product Owner om dette, lever op til dette krav.

Fejl og mangler

Guest & history (mangel)

Vi har i programmet implementeret mapperklasser til history og guest. Derudover var der også lavet tabeller på database til disse samt en entitetsklasse. Men vi nåede ikke at anvende dem, da vores Product Owner prioriterede at vi i det sidste sprint, pudsede det sidste støv af programmet og fejlsikret alt.
Ideen med history var på databasen at flytte gamle reservationer over i et slags arkiv, hvilket vi også gør. Næste skridt ville have været, at lave en form for listevisning i grænsefladen, så en receptionist eller anden bruger, kunne tilgå disse informationer.
Guest tabellen i databasen skulle indeholde information om alle de guest_id'er der skulle være tilknyttet en booking. Ved check in i lobbyen skulle disse generes og gives til de overnattene gæster. Deres guest_id ville så have være deres id til reservation af sportsfaciliteter og booking af instruktører.

Århundrede skift (fejl)

Som koden er nu vil det ikke være muligt at lave en booking i år 2100 og over, da vores program kun anvender de to sidste cifre, altså 00 i dette tilfælde, vil programmet tolke dette som en dato tilbage i tiden(2000). Dette vil gøre det umuligt at oprette reservationer der starter efter 31-12-2099, før datoen 01-01-2100 og samme fejl vil så gælde indtil næste århundrede skift. Denne fejl er ikke så kritisk da der er over 75+ år til dette vil begynde at kunne ses og derfor begynde at være en kritisk fejl. Vi går udfra at et standard IT-system skal udskiftes/opdateres indenfor et væsentlig kortere tidsramme. Den bedste og mest simple løsning ville være at vi inkluderede de to sidste cifre i årstallet så formatet bliver DD-MM-YYYY eks. 30-04-2014

Cronjob kører 4 gange samme nat (Fejl)

Når programmet starter, startes der en CronThreader der håndterer cronjobbet om natten(se CronThreader under vigtigste features). Da der på hotellet er 4 terminaler til receptionister skaber det et potentiale for at have fire cronjobs hver nat, hvilket jo er 3 unødige. Dette er ikke en kritisk fejl da det på ingen måde vil påvirke eller korruptere data men blot udføre en masse unødvendige checks. En løsning kunne være at køre proceduren på databasen som med en "scheduler". Dette var også det første vi forsøgte at gøre, men det var ikke muligt da vores brugerettigheder på databasen ikke inkluderer at eksekvere planlagte jobs.
Hvis alle fire terminaler i receptionen er lukket ned (hvis programmet ikke kører), vil cronjobbet aldrig blive foretaget om natten. Som nævnt i forrige fejlbeskrivelse ville vi kunne løse dette på

database-niveau, hvis vi havde rettigheder til det. Vi kunne også istedet for et cronjob havde lavet et anacron der i bund og grund har samme funktion men udvidet så hvis programmet var lukket natten over ville det kører proceduren ved opstart af programmet.

Hvad ville vi gøre anderledes

Hvis der havde været tid til det, ville vi lave en separat klasse til at håndtere alt vores datomodelering; altså beregninger, format checks og lignende. Lige nu laver programmet nogle af de samme checks flere steder. Dette er sket fordi vi har oplevet mange problemer med metoder der har returneret forkert data fordi der er sket et format flip ved kommunikation mellem mapper og databasen så DD-MM-YY er blevet YY-MM-DD og metoderne har forventet første format. Vi ville også gerne have implementeret vores dato beregninger som vi gør i `DrawMonth` hvor vi laver datoer om til tal(Se GUI Month view, under vigtige features).
Som det er nu, sker der en masse overflødige loops i tegneklasserne, f.eks. for `DrawApartment` som looper igennem listen af reservationer den får ind, for hver række den skal tegne istedet for at loope gennem mindre bidder efterhånden som rækkerne bliver tegnet.
En ting vi havde på vores agenda men som tiden ikke var til, var at udvide logikken i BookingList så når der blev klikket på en boks kunne man navigere til editBooking. Ligeledes hvis man klikker på et tomrum altså et sted uden en Booking skulle man kunne komme til newBooking.

Proces

Hvad er Scrum

Scrum er en fleksibel software udviklings proces, som bygger på at software udvikling er en uforudsigelig proces, hvor man fra starten ikke altid kender alle kravene for sin opgave, og at kravene ændrer sig hele tiden. Scrum er en agil proces, til forskel for udviklingsprocessor som følger vandfaldsmodellen og cirkelmodellen. Vandfaldsmodellen har syv punkter som udviklingsforløbet skal igennem. Det er syv klassiske punkter hvis man følge den originale model.

1. Kravspecifikation
2. Design
3. Konstruktion
4. Integration
5. Afprøvning / Fejlfinding
6. Installation
7. Vedligeholdelse

I Scrum anerkender man punkternes vigtighed, men opdeler dem i stedet i fire mere brede kategorier. Analyse, Design, Implementation og test. Rækkefølgen for disse punkter ligger ikke fast, så man kan starte hvor man vil. De syv andre punkter bliver stadig brugt, men man går i stedet igennem alle syv punkter hver gang man laver en rettelse eller tilføjelse i ens program, eller hver gang ens køber vil have foretaget ændringer. Scrum har en rækker roller, artefakter og møder.

Scrum's 3 roller

- Product owner
- Scrum master
- Development Team

Scrum's 5 møder

- Backlog refinement meeting
- Sprint planning meeting
- Daily scrum
- Sprint review meeting
- Retrospective meeting

Scrum's artefaktor

- Product backlog
- Sprint backlog
- User stories
- Burndown chart

Vores brug af Scrum

I dette forløb har vi lagt meget vægt på Scrum delen af projektet. Spørgsmål vi altid har tænkt over har været, hvordan man arbejder i store software virksomheder? Hvordan bliver arbejdet fordelt? Hvordan bliver arbejdet organiseret? Hvordan holder de ansatte sig opdateret om projektets status? og hvordan planlægger man i det hele taget et stort software projekt?
Nogle af disse spørgsmål er blevet besvaret gennem vores Scrum forløb; men der hersker stadig tvivl i krogene. Når det er sagt, så har vi brugt Scrum, og Scrum har hjulpet os med strukturering, planlægning, arbejdsfordeling og med Git i baghånden har vi haft deling og revision.
Der hersker ingen tvivl om at Scrum har haft et indhug på den tid vi har haft til at arbejde på selve programmet. Der er gået meget tid med snak, møder og planlægning. I tidligere projekter har vi haft en hurtig snak om hvad der skulle laves, hvor folk derefter laver nogle forskellige klasser og fylder dem ud. Så sendte man koden til hinanden og satte det ind og testede om det spillede sammen, og rettede eventuelle fejl. Hvilket godt kunne være ret omstændigt. Med Scrum kan det virke som om at vi har spildt meget tid på planlægning. Ud fra traditionelle omstændigheder er det nok tilfældet, Scrum var nyt så det krævede tilvænning; Alt i alt er det gået hurtigere, vi har sparet tonsvis af tid ved at bruge Git så vi kunne undgå alt “Copy paste”, og arbejdet har været struktureret fra starten. Tiden vi har brugt på ingenting og rettelser i fejlkommunikation er i projektets afslutning dalet til næsten ingenting. I projekter af den her størrelse ville det have været svært at få noget brugbart ud i sidste ende, hvis ikke vi havde haft scrum til at organisere arbejdet, uden ordentlig planlægning er mængden af spaghetti kode og fejl meget større. Vi er ikke i tvivl om at Scrum i sidste ende har sparet os for utrolig meget tid og besvær, og brugen af Git gør hele oplevelsen og arbejdet meget sjovere og effektivt.
Scrum følger en fast procedure for hvordan man skal planlægge sit arbejde. Det har været en positiv oplevelse, men vi har også opdaget at Scrum har sine begrænsninger, som man gjort det lidt udfordrende. Måske er Scrum en agil udviklingsproces, men Scrum i sig selv er temmelig firkantet.

User stories

Vi starter med vores brug af user stories til at planlægge og uddele arbejdet. På billedet under kan man se hvordan en user story ser ud. “O” er vores user story ID, efterfulgt af en beskrivelse, en vægtning af vores product owner, et estimat foretaget af os. Så kommer “How to Demo”, som er en beskrivelse af hvordan dette stykke program skal vises.

O	Receptionisten kan oprette reservationer for kunder, og gemme dem til senere brug.	100	6	Start programmet. Tryk på “new booking” knappen. Udfyld dato, nætter og type i første kolonne. tryk på “find” og vælg en lejlighed. Udfyld reservations skemaet tryk på “Submit”. En besked kommer frem med “Booking created”

I vores gruppe var vi 5 , så alle vigtige beslutninger kunne tages pr. afstemning, hvor flertallet fik trumfet deres holdning igennem. Dette gjorde vi brug af under estimatet. Hvis en estimering af en user story trak ud og alle var uenige om varigheden, blev den kaldt op til afstemning med to ultimatums fremsat af vores Scrummaster.
Da vi estimerede vores user stories lavede vi 4 gule kort til hver, med hver deres bogstav som angav en størrelse. Størrelsen brugte vi kun som guide til hvor lang tid vi mente en task ville tage, for vi havde kun 4 værdier at arbejde med. Efter en estimering med kort snakkede vi os frem til et mere præcist antal dage. kortenes størrelse var: S = 2, M = 4, L = 8 og XXXL var mange. Dvs. at den var en såkaldt "Epic" og skulle derfor revurderes.

Figur 17: Estimerings kort

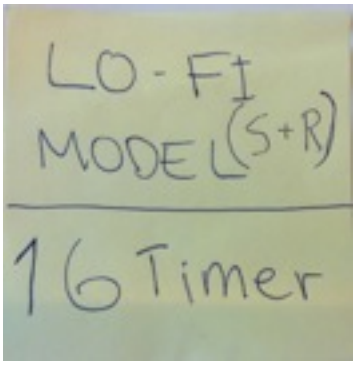
Estimerings processen foregik således:

1. Udvælgelse af user story til estimering.
2. Vurder den individuelt og vælg et kort som passer til den estimerede størrelse.
3. Fremvisning af kort og diskusion om valg.
4. Kort skænderi.
5. Afslutning af estimering af user story.

Processen tog i alt pr. user story cirka 5 - 10 minutter.

Vores estimater for en user story blev udregnet på mandedage. En mandedag i vores projekt svarer til 25 timer. Det var så mange timer der var fastsat pr. dag for hele holdet.
I Scrum skal dette estimat også tage højde for det der hedder *fokus faktor.* Fokus faktoren er den udnytningsgrad vi har af en time, for som regel kan man ikke holde fokus i alle 60 minutter, så noget af tiden vil forsvinde ud i ingenting. Det har sat tydelig spor i starten af vores projekt at vi ikke har medregnet dette. I det hele taget kan man godt se det på hvor langt vi er nået i vores User stories.

Tasks



Figur 18: Task eksempel

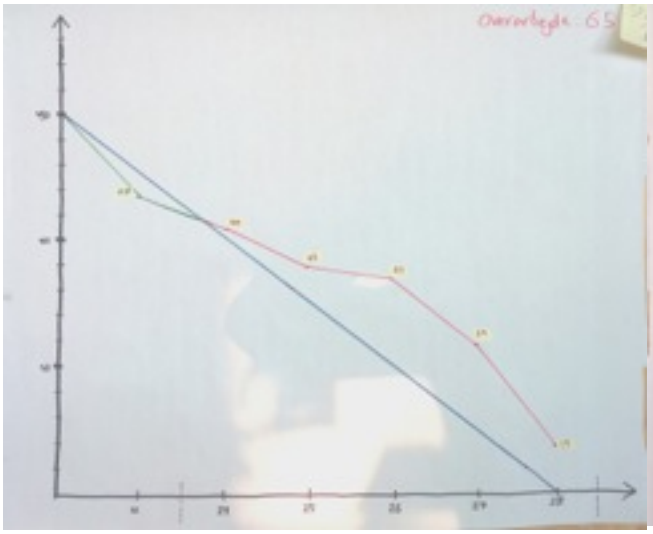
Dykker vi et lag længere ned kommer vi til vores "Tasks". En task er en user story nedbrudt til tekniske opgaver, som skal færdigøres før en user story er færdig. Tasks er forskellige fra hold til hold, de varierer alt efter hvordan man vælger at lave sit program.
Vores tasks bestod af alle de ting som skulle laves i projektet, det gjaldt ikke kun det tekniske men også ting som at lave diagrammer, foretage research på bestemte områder vi ikke vidste noget om, lave

lo-fi modeller af den nye brugergrænse flade som fulgte med den nye user story, kode analyse og refakturering. En konvention vi blev enige om var at skrive initialer på vores tasks så man kunne se om hvem der lavede dem. På denne task på forrige side skulle Simon og Robert klarer Lo-fi modelleringen. For at undgå for meget diskussion blev vi i gruppen enige om at personenerne som havde fået en opgave fik veto-ret på dem. Så hvis nogen var utilfredse, eller de synes at der var noget som manglede, var det helt op til den task ansvarlige om det skulle komme med.
Til estimering af en task brugte vi de samme kort som til estimeringen af vores User stories; i stedet for mandedage, blev det nu til timer. Det var kun i første sprint at denne konvention holdte. Normalt er tasks estimeret på story points som er timer medregnet projektets fokus faktor, det gjorde vi ikke. havde vi medregnet fokus faktor, skulle vi trække fokusfaktoren fra timerne. I det andet sprint prøvede vi at estimere med fokus faktor i tankerne. Hvilket resulterede i at vi fik en meget pænere graf på vores burndown chart, men dog med en del overestimering på nogle af opgaverne.

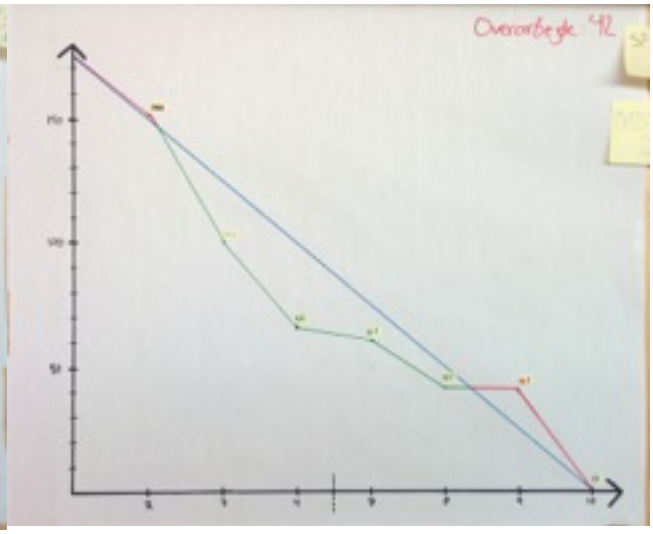
Burndown chart

I Scrum har man en artefakt som er et burndown chart. Burndown chartet visualiserer det tilbageværende arbejde på en pæn graf, hvor hældningen af linjen udgør projektets velocity, altså hvor mange tilbageværende timer der er tilbage af de sammenlagte tasks for det igangværende sprint. I vores Scrum var vores burndown chart ikke særlig vigtig.
Vores burndown chart var en god reminder for hvor travlt vi havde. Når grafen så skidt ud, arbejde vi ekstra hårdt for at få den tilbage på rette kurs, der gik helt sport i det. Vi må anerkende at vores burndown chart har fungeret mere som motiverende graf, end en praktisk graf. Vores mest praktiske anvendelse af grafen var kunne se at vores estimeringer ikke holdte stik. Vi har mange overarbejdstimer hvilket man ikke må have i Scrum, og det var her vi så den første begrænsning i Scrum, hvis estimeringerne i Scrum skal holde stik er man nødt til at have et fast grundlag og det får man kune ved at bruge nøjagtig det antal mandetimer som er sat af det det igangværende sprint. Lige her brød vi også ud af Scrum reglementet og fulgte vores egne veje. Vi mente ikke at man skulle begrænse arbejdet for an holde en pæn linje på grafen. Havde det været i erhverslivet ville vi i have fulgt timer de første par gange, for at få estimatet til at passe og derefter bare have

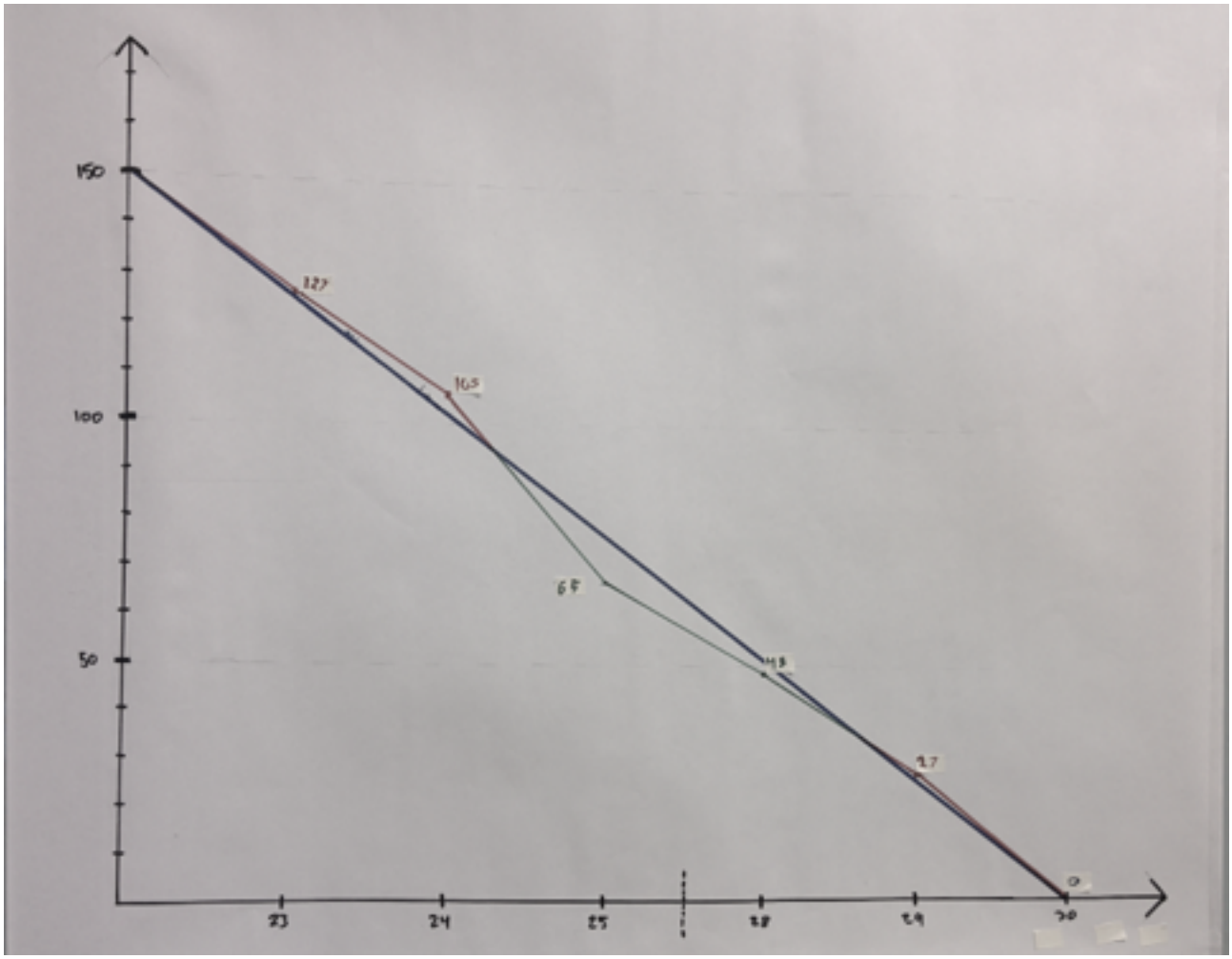
arbejdet løs. Man kan tydeligt så sprint 1 at vi under estimerede vores tasks og ligeledes kan det ses på sprint 2 at vi over estimerede dem.



Figur 19: Sprint 1 - burndown chart



Figur 20: Sprint 2 - burndown chart



Figur 21: Sprint 3 - burndown chart

Daily Scrum

Dette møde var en af de ting vi gjorde meget ud af. Under daily Scrum lavede vi en agenda for hvad der skulle ske i løbet af dagen. Agendaen bestod af alle de pludselig opståede punkter, som krævede en diskussion på gruppen. Det kunne være vejledning, fælles gennemgang af kode, revurdering af et stykke brugergrænseflade, osv.
Vi startede vores daily Scrum med en kort gennemgang af, hvordan folk havde sovet, og om der var sket noget spændende siden sidst vi var samlet. Vi erfarede hurtigt at denne korte gennemgang, gjorde at vi var mere seriøse, koncentrerede, og at det var nemmere at holde fokus Daily Scrum begyndte vi at holde to gange om dagen, et par dage efter vi startede projektet. Efterhånden som arbejdet skred fremad, gik det op for os at det var godt at holde et møde efter frokost, midt på dagen, hvor vi folk snakkede om hvor langt de var kommet med deres tasks. Vores middags Scrum, var også det tidspunkt hvor vi greb chancen for se om der var opstået nogle nye problemer som krævede akut opmærksomhed. Vi havde faktisk et daily Scrum mere på slutningen af dagen. Det sidste Scrum møde holdte vi når vi skulle til at have fri. Det var en gennemgang af dagen hvor vi hørte hvad hinanden havde og sige om deres dag; om hvad der var gået godt og om hvad der var gået skidt. Eftersom vi var en ny gruppe, og at det var et nyt projekt, hjalp det os til at lære hinanden at kende og forbedrede vores gruppe dynamik. I sidste sprint virkede det som om at nu var der styr nok på tingene til at det ikke længere var nødvendigt, men vi har blev ved med at holde mødet, fordi det er vigtig at man føler man bliver hørt, specielt hvis man har noget at sige. Det sidste møde blev også brugt til at opdaterer vores burndown chart.
Alle vores daily Scrum' blev afholdt på klassisk Scrum-vis ved at vi stod op og at mødet maks måtte varer 15 minutter. Derefter skulle arbejdet påbegyndes og dagens opgaver skulle være delt ud. Til vores morgenScrum var der også uddeling af roller. Disse roller er noget vi også selv har indført, både på baggrund af at det er et skole projekt, så vi havde ingen fast Scrum master og på baggrund af noget teori om gruppedannelse. De roller vi var opdelt i var følgende:

ScrumMaster

Klassisk Scrum rolle, men vi brugte dog også rollen som organisator. Vi startede med at rotere Scrummaster's rollen på en daglig basis for at alle kunne få en fornemmelse for rollens ansvarsområder. I andet sprint begyndte vi at have rollen på flerdags basis så man kunne nå at få en bedre fornemmelse af rollens ansvarsområder. Scrum master er personen som står for at styre selve Scrum forløbet. Han holder forstyrrelser væk og sørger for at udviklerne kan arbejde i fred. Han fungerer også som organisator og ordstyrer til Scrummøderne.

Logmaster

Logmaster-rollen har været, at notere og skrive vigtige observationer, gjort i løbet af dagen ned. Når vi så har skrevet logbog til slut på dagen, var vigtige erfaringer og observationer ikke gået tabt.

I det sidste sprint arbejdede vi oftere adskilt og da logmasteren ikke kunne være alle steder samtidig, skiftede rollen hurtigt til at være et fælles ansvar. Et bestemt særligt ansvar logmasteren havde var at holde styr på straffe kassen, hvilket har været en af de vigtigste tiltag i dette projekt.

Boardmaster

Det har været boardmasterens job, at sørge for vores opslagstavle hele tiden har været up to date. Under Daily Scrum skulle boardmasteren skrive dagens agenda op og i løbet af dagen holde denne opdateret, så ingen var i tvivl om hvor langt vi var. Det har også været boardmasteren's ansvar at holde notetavlen opdateret.

Roller var helt sikkert noget som har hjulpet på vores effektivitet. Da vi startede på projektet følte vi at der var stor chance for at vi ville gå over og blive useriøse. Rollerne og vores Scrummøder har uden tvivl hjulpet på vores fokus igennem hele projektet. Under forløbet og efter vi havde haft et par konflikter hvor vi var blevet irriteret på hinanden efter at der var sket en fejlkommunikation og arbejdet ikke var blevet udført som ønsket, fik vi anskaffet en rispose som vi brugte under diskussioner hvor en ophedet debat var under opsejling. Det lyder meget pædagogisk… Og det var det også! Men det korte og det lange var at det var en fair måde at skabe lighed i diskussionerne. Ved at fjerne ordstyren og indsætte risposen ind, var der ingen følelse af partitagen fra ordstyren. Det kunne også nogle gange kamme den anden vej og blive til total spild af tid og et pædagogisk mareridt hvor ingen kunne tage noget seriøst.

De andre møder

Sprint planning, retrospective review og backlog refinement møderne har vi ikke så meget at sige om. Alle møder er nogen vi har holdt så godt vi kunne, efter de kendte scrumprincipper. Retrospective mødet, har været det møde der er kommet mest ud af, hvad angår konstruktiv kritik om samarbejdet i gruppen. Det skal ikke lyde som om at alt har været guld og grønne skove. Der har mange gange været en anspændt stemning og et skarpere tonefald når vi har snakket med hinanden, men det er problemer som vi fik afviklet ved at det sure individ enten gik en tur eller at vi tog risposen i brug. Ved dette møde reflekterer Scrum holdet over hvad der er gået godt og hvad der er gået skidt i Scrumforløbet. Ved dette møde udarbejder man typisk en liste over disse ting. Ved dette møde bliver der lavet en præsentation af de færdige User stories. Dette foregår typisk med en demonstration af det færdige software. Ved dette møde kan alle se hvad der er blevet lavet, product owner, aktionærer osv.

Git

Fordele

Git har været vores mest brugte værktøj foruden vores IDE, Netbeans, og dens debugger værktøj. I vores gruppe ville vi fra starten bruge branches til at administrerer vores arbejde. Vi ville gerne have lavet mange flere branches end vi egentlig gjorde, men Git var stadig nyt, så for at spare tid og gøre det nemmere for os alle sammen begrænsede vi os i sidste ende til at lave 4 branches:

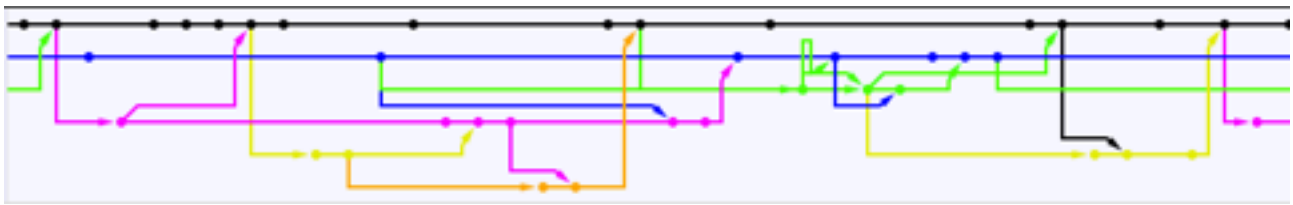
- master
- DataSource
- JUnit
- debugger

master branchen var der hvor vores sidst færdige, virkende version af programmet var. Alt der blev skubbet og flettet med master branchen skulle være testet. Hvis man skubbede op noget med fejl, til origin/master, som var master branchen på Github, så var der straf som er udspecificeret i gruppekontrakten.

DataSource branchen var den branch, hvor alt logik til datasource laget i vores projekt blev lavet. Vi lavede den opdeling fordi vores datasource lag var en stor del af projektet som hele tiden blev refaktoreret og ændret, så for at undgå at vi skulle flette med vores masterbranch hele tiden kom den i et lag for sig selv vi flettede sammen med master når der var foretaget store ændringer.

JUnit var vores testbranch. Det var først meget senere i projektet at vi begyndte at teste med JUnit, så skelettet blev sat op i en branch for sig selv, så vi kunne udvikle videre uden at beskæftige os med tests undervejs. Det kan virke dumt at fjerne testpakken, men de fleste fejl og tests var nogen som kom i vores grafiske brugergrænseflade og der var debuggeren i netbeans meget bedre.

Debugger branchen var den branch hvor vi splittede ting ad. Den her branch blev slettet og oprettet mange gange i løbet af projektet. Hvis vi havde lavet en fejl vi først opdage efter at den var blevet flettet og skubbet op til origin/master på github, så kom debugger i brug. Når fejlen var var fundet og en løsning var lavet, så blev master opdateret og debugger branchen slettet.



Figur 22: Github - Branching

Ulemper

Den første uge vi brugte Git, havde vi nogle store problemer som tog fire timer at løse. Fire timer pr. man vel at mærke. Git er et godt værktøj men vi har også haft vores problemer med det. De største problemer kom af at vi havde kommet til at skubbe en mappe med med private konfigurations indstillinger som var forkellige fra computer til computer, op til vores repository på Github, så pludselig havde vi merge konflikter vi ikke kunne løse. Fejlen lå selvfølgelig på vores side, men vi fik løst problemet ved at bruge Git revision værktøj, så vi kunne rulle tilbage til en tidligere version.

Vores workflow blev også påvirket af netbeans xml generering til swing. Vi kunne kun sidde en mand og arbejde på projektets brugergrænsefalde af gangen fordi vi ikke kunne løse de XML konflikter som swing ville skabe, fordi neatbeans autogenerede kode, når vi arbejde i swing.

Konklusion

Vi har brugt dette projekt, som en platform til at lære om Agile udviklingsmetoder. Vi har fået en bedre forståelse, for forskellige kerneprocesser i udvikling af software som krav specificering, designfase, implementation og testing. Vi har lært at anvende versionsstyringsværktøjer(Git). Vi er blevet bedre til at estimere tider på opgaver (se afsnit om burndowncharts). Med scrum erfarede vi hvordan vi kunne udpensle kravene til en problemstilling, som kunne afgrænses til separate userstories, som så kunne opdeles i estimerbare tasks og implementeres til færdige bidder af det endelige mål, og præsentere det for en Product Owner. Vi har lært hvordan et fast nedfældet regelsæt, der beskriver normer og social omgang, kan reducere antallet af konflikter, da der er noget konkret at holde sig op imod. Vi blevet er bedre til at prioritere og vurdere hvor stor betydning noget har, for det nuværende tilstand et projekt befinder sig i, og det endelige mål der arbejdes mod. Vi har erfaret at det er utroligt vigtigt, at have stor kendskab til hinandens styrker og svagheder, for at kunne optimere arbejdsprocessen mest muligt.