Projektrapport

Pristjek220

Dato for aflevering: 27-05-2016

|  |  |  |
| --- | --- | --- |
| Aarhus Universitet - Ingeniørhøjskolen | | | Vejleder: Troels Fedder Jensen (TFJ) |

|  |
| --- |
| Gruppe 7 |

# Resumé

Resumé af projektet med problemstilling, hvordan det er løst og konklusion

# Abstract

Resumé på engelsk

Indholdsfortegnelse

[1 Resumé 1](#_Toc451330286)

[2 Abstract 1](#_Toc451330287)

[3 Indledning 4](#_Toc451330288)

[4 Læsevejledning 4](#_Toc451330289)

[4.1 Termliste 4](#_Toc451330290)

[5 Projektformulering 4](#_Toc451330291)

[6 Afgrænsning 5](#_Toc451330292)

[7 Systembeskrivelse 6](#_Toc451330293)

[7.1 Krav 6](#_Toc451330294)

[7.1.1 Udformning af krav 6](#_Toc451330295)

[7.1.2 Aktører 6](#_Toc451330296)

[7.1.3 User story beskrivelser 7](#_Toc451330297)

[7.1.4 Kvalitetskrav 8](#_Toc451330298)

[8 Projektgennemførelse 8](#_Toc451330299)

[8.1 Iterativ udvikling 9](#_Toc451330300)

[8.2 Dokumentation af koden 9](#_Toc451330301)

[8.3 Dokumentation af systemarkitektur 10](#_Toc451330302)

[9 System Design 10](#_Toc451330303)

[9.1 Arkitektur 10](#_Toc451330304)

[9.2 GUI design overvejelser 12](#_Toc451330305)

[9.3 Pristjek220 Database 13](#_Toc451330306)

[9.3.1 Design af databasen 13](#_Toc451330307)

[9.3.2 Databaseadgang 14](#_Toc451330308)

[10 Produkt beskrivelse 15](#_Toc451330309)

[10.1 15](#_Toc451330310)

[10.2 Delte funktionaliteter 16](#_Toc451330311)

[10.3 16](#_Toc451330312)

[10.4 Forbruger 16](#_Toc451330313)

[10.5 17](#_Toc451330314)

[10.6 17](#_Toc451330315)

[10.7 17](#_Toc451330316)

[10.8 Administration 17](#_Toc451330317)

[10.8.1 Administrator 17](#_Toc451330318)

[10.8.2 Forretningsmanager 18](#_Toc451330319)

[11 Test af pristjek220 18](#_Toc451330320)

[11.1 Unittest 19](#_Toc451330321)

[11.2 Integrationstest 19](#_Toc451330322)

[11.3 (evt. Code Metrics) 19](#_Toc451330323)

[11.4 CI 19](#_Toc451330324)

[11.5 Forbrugertest 19](#_Toc451330325)

[12 Resultater og diskussion 19](#_Toc451330326)

[13 Fremtidigt arbejde 20](#_Toc451330327)

[14 Konklusion 20](#_Toc451330328)

[15 Referencer 20](#_Toc451330329)

[16 Underskrifter 21](#_Toc451330330)

# Indledning

Beskrivelse af hvad vi er kommet igennem projektet og hvad formålet er med projektet.

# Læsevejledning

Her skal stå en læsevejledning

## Termliste

BLL = Business Logic Layer

DAL = Data Access Layer

PL = Presentation Layer

FDD = Feature Driven Development

TDD = Test Driven Development

SRP = Single Responsibility Principle

DIP = Dependency Inversion Principle

# Projektformulering

Som forbruger er det svært at danne sig et overblik over, hvor det er billigst at handle ind. Der er i dag mange forskellige forretningskæder, der konkurrerer med hinanden, og dette resulterer i et stort udbud af forskellige varer med forskellige priser. Der er derfor blevet fremstillet et program, kaldet Pristjek220, som tilstræber at give forbrugeren et let og simpelt overblik over, hvor de forskellige varer kan handles billigst.

I Pristjek220 Forbruger er den grundlæggende funktionalitet, som forbrugeren har at kunne finde den billigste forretning for en vare. Han kan samtidig også se hvilke forretninger, der har varen, og hvad den koster de forskellige steder, når han søger efter en vare. Konceptet bag at slå en vare op i Pristjek220 er illustreret på Figur 1. Det er denne grundidé, som resten af funktionaliteterne for Pristjek220 Forbruger bygger på. Når forbrugeren søger efter en vare i Pristjek220, bliver han, så snart han har indtastet to bogstaver, foreslået varer, der indeholder de bogstaver, han har indtastet for at effektivisere brugen af Pristjek220.

For at gøre Pristjek220 mere effektivt i forbindelse med forbrugerens indkøb kan han indtaste en indkøbsliste, og Pristjek220 informerer så forbrugeren om, hvor varerne fra indkøbslisten kan findes billigst, samt hvad de koster. Her har forbrugeren samtidig mulighed for at se en sammenligning af, hvad det vil koste at købe alle varerne i én forretning i modsætning til at købe varerne, der hvor de er billigst. Forbrugeren har ligeledes mulighed for at kunne justere, hvor varerne skal købes, efter Pristjek220 er kommet med listen over, hvor det er billigst. Dette giver forbrugeren mulighed for at vælge at flytte én enkelt vare på listen til en anden forretning, f.eks. i det tilfælde at han skal handle i tre forretninger, men kun skal have én vare i den ene forretning. Yderligere kan det angives, hvis der er nogle forretninger, man ikke ønsker at handle i. Når forbrugeren har fået genereret sin indkøbsliste af Pristjek220, som han vil have den, har han mulighed for at ændre den og derefter få den sendt til sin egen E-mail. Dette gør, at han kan tilgå listen via hans mobiltelefon.

Pristjek220 har tre forskellige brugere; en forbruger, en forretningsmanager og en administrator. Forbrugeren er ham, der bruger Pristjek220 til at organisere sine daglige indkøb. Forretningsmanageren holder Pristjek220 opdateret med korrekte informationer om de varer og priser, der findes i netop hans forretningskæde. Administratoren servicerer Pristjek220, så der kan oprettes og fjernes forretninger. Baseret på disse tre brugere er Pristjek220 opdelt i to programmer; et program, Pristjek220 Forbruger, til forbrugeren og et fælles program, Pristjek220 Forretning, til både forretningsmanageren og administratoren. I Pristjek220 Forretning kan administratoren benytte sig af sine funktionaliteter ved at logge ind med et administratorlogin, som han har.

I Pristjek220 Forretning kan forretningsmanageren logge ind med sit forretningsmanagerlogin. Derefter har han mulighed for at tilføje og fjerne varer, samt ændre prisen på en vare, fra den forretning, han styrer. Han bliver samtidig bedt om at bekræfte sine valg, så sandsynligheden for, at han gør noget ved en fejl, er minimeret. Ved at forretningsmanageren holder informationerne om varerne fra hans butik opdaterede, holder han forbrugerne oplyste og giver dem de bedste vilkår for at få fuldt udbytte af Pristjek220. Administratoren kan i Pristjek220 Forretning logge ind med sit administratorlogin, hvorefter han kan tilføje en forretning med tilhørende forretningsmanager eller fjerne en forretning fra Pristjek220.



Figur : Rigt billede over opslag af en vare i Pristjek220.

Pristjek220 henvender sig til det moderne menneske, som er økonomisk anlagt og gerne vil spare penge, når der handles dagligvarer i hverdagen. Det henvender sig altså både til studerende uden mange penge på lommen, samt familier der ønsker at få mest muligt ud af deres penge.

# Afgrænsning

Pristjek220 består af to programmer hvor det ene er tiltænkt til forretningerne og det andet til forretningernes forbruger. Gennem disse programmer kan forretningerne tilføje og fjerne informationer, om deres sortiment af forretningen i pristjek220 database via internettet. Pristjek220 database kan kun tilgås lokalt eller via VPN, derfor er det nødvendigt at oprette en VPN forbindelse. Forbrugerne kan sammenligne forretningernes sortiment med andre forretninger, og derved finde ud af hvor det er billigst at handle. Da forretninger selv står for at opdatere deres sortiment vil forbrugerne ikke have mulighed for at sammenligne med de forretninger som vælger ikke at være med i Pristjek220.

Pristjek220 er udviklet som computerapplikationer, og der er af den grund udviklet en mulighed til forbrugerne så de kan sende deres indkøbsseddel på E-mail, da de fleste mobiltelefoner understøtter E-mails.

# Systembeskrivelse

Kort beskrivelse af systemet med et eller andet fint billede.

## Krav

Ud fra *Projektformuleringen* er der opstillet en række User Stories, der beskriver brugerens interaktion med systemet. User Stories danner grundlag for kravspecifikationen, og bruges til at fastsætte systemets funktionalitet.

### Udformning af krav

For at opbygge en kravspecifikation for Pristjek220 skulle der i starten af projektet formuleres funktionelle krav i form af forskellige scenarier. Gruppen havde tidligere kun arbejdet med use cases, hvor der var erfaring med, at de ikke fungerede særlig godt, når der skulle arbejdes iterativt. Use cases lægger op til at blive uddybet detaljeret, når de formuleres, hvilket betyder, at der lægges meget arbejde i at udarbejde dem i starten af projektet. Det betyder også, at når der i løbet af den iterative proces ændres på detaljerne i designet og implementeringen, skal man tilbage og ændre på detaljerne i sine use cases. Gruppen ønskede derfor at finde en metode til at udforme de funktionelle krav, som lagde mere op til en iterativ udvikling.

Der er derfor valgt at benytte user stories til at formulere de scenarier, der udgør de funktionelle krav. En user story er en kort beskrivelse af scenariet, samt hvad der skal til, før implementeringen af det kan godkendes. Detaljeringen af scenariet bliver i stedet udskudt, indtil det skal designes og implementeres. Til den tid hører der en diskussion med kunden til, hvor han fortæller, hvordan han ønsker, den specifikke funktionalitet for scenariet skal være, og dette diskuteres i forhold til, hvad der er muligt. Det er ved denne diskussion, at detaljeringen fastsættes.

Fordelen ved at benytte user stories er netop, at detaljeringen udskydes indtil implementeringen. Dette gør, at erfaringen fra de allerede implementerede funktionaliteter kan benyttes når detaljerne fastlægges. Samtidig kan kunden have ændret mening omkring, hvordan en user story skal implementeres på baggrund af ideer eller tanker, han har gjort sig efter at have set de allerede implementerede user stories. Hvis der var benyttet use cases, kunne det resultere i en masse ændringer i de formulerede detaljer for scenarierne. Ved user stories er det derimod ikke et problem, da detaljerne først fastlægges i dette skridt. Det er samtidig en fordel, i et semesterprojekt som dette, at kunne udskyde detaljeringen så lang tid, som det er muligt, da der hele tiden erfares ny viden igennem undervisningen.

Da der er valgt at benytte user stories, hvor en stor del af arbejdet med dem ligger i diskussionen med kunden, men projektet ikke har en egentlig kunde, er dette håndteret på en alternativ måde. Det er gruppen selv, der har stået for produktets vision og ageret kunde, og diskussionen er derfor håndteret ved, at gruppen samt vejleder har diskuteret i starten af hver iteration omkring ønskerne og detaljerne for de specifikke user stories, der skulle implementeres i iterationen.

### Aktører

**Forbruger:** Forbrugeren er en primær aktør. Forbrugeren ønsker at gøre hans indkøb så billigt og let som muligt, ved at få Pristjek220 til at finde ud af hvor han skal købe hans forskellige varer.

**Forretningsmanager:** Forretningsmanageren er en primær aktør. Forretningsmanageren kan tilføje/fjerne varer fra deres forretning, samt ændre priserne.

**Administrator:** Administratoren er en primær aktør. Administratoren kan tilføje og fjerne forretninger fra pristjek220.

### User story beskrivelser

Der er her udvalgt nogle relevante User Stories fra Kravspecifikationen[[1]](#footnote-1), ud fra hvilke User Stories der bliver brugt i løbet af rapporten, de er rangeret efter hvilken prioritering de har haft gennem udviklingsfasen.

//DER ER PÅ NUVÆRENDE TIDSPUNKT, LAVET NOGLE EKSEMPLER PÅ US, MEN DA JEG IKKE VED HVILKE VI BIL BRUGE, KAN DER GODT FOREKOMME ÆNDRINGER SENERE.

**US2: Finde den billigste forretning for en vare i Pristjek220:**

**Som** en forbruger af Pristjek220

**Vil** jeg kunne finde den billigste forretning for en vare fra Pristjek220,

**Så** der kan laves en indkøbsliste med den mindste pris.

User storyen er relevant, fordi det er en fundamental egenskab, for at pristjek220 kan hjælpe forbrugeren med at fortage billige indkøb.

**US4:** **Find ud af hvor varerne fra indkøbslisten kan købes billigst:**

**Som** en forbruger af Pristjek220

**Vil** jeg kunne se hvor det er muligt at købe varerne på indkøbslisten billigst ud fra de indtastede indstillinger for indkøbslisten,

**Så** jeg kan spare penge på mine indkøb.

User storyen er relevant, fordi uden indkøbsliste funktionaliteten ville det blive for besværligt at bruge Pristjek220, hvis der skulle handles flere varer.

**US8: Autofuldførelse:**

**Som** en forbruger af Pristjek220

**Vil** jeg kunne se hvilke vare og forretninger, der allerede findes i Pristjek220, når jeg sidder og søger,

**Så** der skal komme forslag op som jeg kan trykke på.

User storyen er relevant, fordi brugeren gerne vil kunne slippe for at skrive hele den ønskede ting og samtidig være sikker på at produktet er i Pristjek220, uden at skulle slå det op først.

**US9:** **Send indkøbsliste på mail:**

**Som** en forbruger af Pristjek220

**Vil** jeg kunne modtage min indkøbsliste/forslag til indkøbssteder på mail,

**Så** min indkøbsliste bliver mobil.

User storyen er relevant, fordi brugeren gerne vil kunne slippe for at have hans computer med ud og handle, og derfor ønsker en mere transportabel måde at have sin indkøbsliste med på. Dette kan opnås ved at han sender en E-mail til sig selv, som så kan se på en telefon.

### Kvalitetskrav

Der er taget et uddrag af de mest relevante kvalitetskrav[[2]](#footnote-2).

1a: Pristjek220, skal leve op til de krav, som Microsoft, still til UX design[[3]](#footnote-3).

For at opfylde de krav, som der stilles fra Microsoft UX design, er tjekliste[[4]](#footnote-4) som de har udgivet, blevet sammenlignet med det udviklede produkt.

Der er gennem Pristjek220 prøvet at leve op til UX kravene, ved at have gennemgået og anvendt checklisten fra Microsoft.

1b: Pristjek220 skal leve op til 3-click rule[[5]](#footnote-5).

Der er i Pristjek220 i navigationen levet op til ” til 3-click” ved at der ingen steder er mere end 3 klik fra hvor man starter til man er ved den ønskede funktionalitet.

1c: Pristjek220 skal leve op til rule of five[[6]](#footnote-6).

Der er i Pristjek220 i de forskellige menu’er levet op til ”rule of five” ved at der ingen steder er mere end 5 forskellige valgmuligheder.

4b: Kodeord der ligger i databasen skal være krypteret

Der er i Pristjek220 valgt at krypterer kodeord med SHA256, ud fra ideen om at det ikke skulle være muligt at hack databasen og derved få adgang til ikke krypterede kodeord.

# Projektgennemførelse

I dette afsnit vil der blive forklaret hvordan gruppen har gennemført projektet.

## Iterativ udvikling

Der er besluttet at der skal arbejdes iterativt, derfor har gruppen måtte tage et valg til hvilken arbejdsmetode, der skulle benyttes. Gruppen overvejede valget mellem Scrum og Kanban.

Begge metoder anvender et taskboard til at organisere det arbejde, der skal laves, så alle medlemmer på teamet kan se, hvad der bliver lavet og af hvem. Den store forskel er dog, at i Kanban er der en begrænsning på hvor mange tasks, der må være aktive ad gangen, og når der er plads, bliver der bare fyldt på fra en product backlog. Hvorimod i Scrum bliver arbejdet opdelt i sprints, og hvert sprint har sit eget taskboard. Disse sprints bliver udfyldt med stories, som nedbrydes til tasks, fra en product backlog, inden de opstartes, og hvis alle tasks er udført, kan man tilføje flere fra product backloggen, og hvis man ikke når alt, føres de videre til næste sprint med højeste prioritet. Generelt set egner Scrum sig bedre til udvikling af systemer, og Kanban fungere bedst til vedligeholdelse af systemer. Scrum sprintene er designet efter et færdigt produkt ved hvert sprint retrospective, så hvis kunden ikke har flere penge at smide i projektet kan han stadigvæk gå derfra med et produkt, der har nogle funktioner implementeret. Med hensyn til vedligeholdelse af andre systemer vil der højst sandsynligt dukke bugs op i produktet, og med et Kanban system kan bugs hurtigt blive givet en prioritet og blive fixet.

Udviklingen af Pristjek220 er foregået med Scrum, fordi det er et nyt system der bliver udviklet. Det var givet fra starten hvornår deadline for projektet faldt, og med en beslutning på at køre sprints af 14 dages længde. Så det var kendt viden, hvor mange sprints der ville forekomme, og derved kunne gruppen danne et overblik over hvor meget tid der ville være til rådighed, hvis et nyt aspekt, fra kravspecifikationen, skulle implementeres.

Yderligere har gruppen fra starten hældt mere mod Scrum, både fordi at sådan har arbejdsfaconen været på tidligere semesterprojekter, og alle gruppens medlemmer har gennemført et Scrum kursus på Systematic A/S.

Gruppen har afveget fra Scrum standarden og ikke tildelt nogen Scrum master rollen, fordi det blev bestemt, at det ikke var en rolle, som en person skulle stå med, og derfor blev Scrum masterens opgaver fordelt på tværs af hele teamet. Yderligere er der blevet gjort erfaringer med opsætningen af product backloggen, især med fokus på hvordan taskene skulle opskrives. Fra tidligere semestre var gruppen oplært i at et taskboard skulle bestå af mange små opgaver, hvori man ofte kunne rykke en opgave fra in progress til review / done. Men denne gang var det essentielt at opgaverne skulle give værdi for kunden. Fra starten af var der en del uenighed på den front, med argumenter som: ”Kunden kan da være ligeglad med hvordan vi strukturere vores arbejde, når bare det bliver lavet” Men efter et par sprints begyndte det at give mening, fordi det føltes mere naturligt at arbejde på den metode. Man var færdig når man kunne rykke sin task, og ikke når man havde rykket 5-6 stykker. Idéen med de små opgaver blev dog ikke helt udredet, de blev beskrevet i de individuelle tasks’ beskrivelses felt, og derigennem kunne gruppen arbejde på deres foretrukne metode, imens deres tasks skabte værdi for kunden.

## Dokumentation af koden

En god dokumentation af source koden er værdifuld for projektet, for let at kunne danne sig overblik over koden, så det er let at sætte sig ind i, og dermed også let at overskue hvilke funktionaliteter der er tilgængelige. Manuel dokumentation af klasser og metoder i koden er tidskrævende, og egner sig ikke godt, når der laves ændringer i koden, da der derved er en masse dokumentation, der skal rettes for at holde dokumentationen opdateret. Det ville derfor være optimalt, hvis klasser, metoder og variabler i koden automatisk kunne registreres, så man kun behøvede at skrive lidt beskrivende tekst til, og derved ikke selv skulle til at skrive hvilke klasser, metoder og variabler, der eksisterer i koden.

Der findes en løsning på dette problem, som er en dokumentationsgenerator, hvilket er et programmeringsværktøj, der automatisk sætter dokumentation op for klasser, metoder og variabler, der findes i koden. Programmøren skal derfor blot tilføje kommentarer i en speciel syntaks i koden, til de ting der skal dokumenteres. Det er Microsofts dokumentationsgenerator, der hedder Sandcastle, der er benyttet i dette projekt.

Fordelen ved at benytte Sandcastle, frem for en anden dokumentationsgenerator som Doxygen, er, at Sandcastle er godt integreret med Microsoft Visual Studio og C#, som allerede kender syntaksen, og derfor er meget nemt at benytte. Sandcastle kan dog kun benyttes på .Net Frameworket, hvor Doxygen er mere fleksibelt, men der er i dette projekt ikke behov for denne fleksibilitet.

Den største og vigtigste fordel ved at benytte Sandcastle er, at opsætningen af dokumentationen sker automatisk, og der derved er mindre vedligehold for at holde dokumentationen opdateret. Programmørens eneste ansvar er derfor at skrive sigende kommentarer i koden, til den metode eller lignende der er blevet skrevet.

## Dokumentation af systemarkitektur

Til at beskrive systemarkitekturen for projektet er der i de tidligere semesterprojekter anvendt flere forskellige SysML diagrammer. Fælles for disse er, at de egner sig bedst til systemer, hvor der indgår hardware og software. Dette projekt er derimod kun et softwareprojekt, og der var derfor et behov for at finde en anden metode til at beskrive softwarearkitekturen, som fokuserede mere på softwareintense projekter.

Af denne grund bliver ”4+1” view modellen benyttet til at beskrive systemarkitekturen. Modellen består af fire views, der bruges til at beskrive systemet fra forskellige synspunkter. De fire views er logical view, development view, process view og deployment view. Deployment view og process view er ingeniørens tilgangsvinkel til systemet. Kundens tilgang til systemet er fra logical view. Projektledere bruger development view til at se projektet fra. Derudover består modellen også af et sidste view, som omhandler scenarierne i systemet. Disse scenarier beskriver de interaktioner, der finder sted i systemet, og er beskrevet i form af use cases eller user stories. Scenarierne er frigjorte fra de andre views, og bruges til at finde frem til elementerne i arkitekturen, og de fire andre views er derfor illustreret ved hjælp af disse scenarier.

Fordelen ved at benytte denne model er, at man får en systemarkitektur, der er godt beskrevet fra flere forskellige relevante synspunkter. En stakeholder i systemet har derved nem adgang til de informationer, der er behov for, for at forstå systemet på hans niveau. Når man har tilføjet en funktion, er modellen samtidig god til at sørge for, at man kommer rundt omkring alle views, og får overvejet, om man har ændret noget i forhold til de forskellige views. Derved er man sikker på, at man får dokumenteret det, hvis man f.eks. har lavet ændringer i samtidigheden. Ulempen ved modellen er derimod også, at der nemt kan bruges alt for meget tid i systemets udviklingsproces på at dokumentere igennem en masse forskellige diagrammer i de forskellige views. Det er vigtigere at have et fungerende system frem for en komplet dokumentation, da dette sløver udviklingsprocessen mere, end det gavner. Derfor er der prioriteret efter hvilke views og diagrammer, der giver mest mening at benytte i projektet, og de resterende diagrammer er derfor udeladt. Dette betyder, at der i Pristjek220 ikke er benyttet process view, samtidig med at det kun er nogle bestemte diagrammer, der er benyttet i de andre views, da det var dem, der blev vurderet til at være mest værdifulde. For en gennemgang af hvilke diagrammer der er benyttet, henvises der til dokumentationen for Pristjek220[[7]](#footnote-7), hvor selve diagrammerne for projektet også kan findes.

# System Design

## Arkitektur

Da arkitekturen af projektet skulle besluttes, blev der valgt at bruge en lagdelt model. Grunden til at valget faldt på den, var at ved den lagdelte model er der en gruppering af klasser og pakker. Herved har de klasser, der befinder sig i ét lag, et sammenhængende ansvar for et vigtigt aspekt i systemet i den logiske separation, der er lavet i projektet. Ved at gruppere systemet på denne måde, kan der hurtigt findes ud af, hvor der skal ændres ting, hvis der er noget, der skal ændres. Med valget af at bruge den lagdelte model skulle der så tages en beslutning om, hvor mange lag der skulle inddeles i. Ud fra størrelsen af systemet gav det mest mening at bruge en 3 lagdelt model[[8]](#footnote-8). Den 3 lagdelte model, består af; DAL, BLL og PL.

Et eksempel på hvordan et normalt scenarie kunne se ud, kan ses på Figur 2, som viser hvad der sker, når en bruger indtaster i et felt, der kan autofuldføre. Først ændrer brugeren på teksten på GUI’en, hvorefter GUI’en kalder ned på GUI viewmodel, for at den skal udfylde listen. Dette sker i PL, hvorefter der så kaldes en funktion i klassen AutoComplete i BLL. Denne funktion kalder så en funktion i en klasse i DAL, som laver et database udtræk ud fra den indtastede tekst.



Figur : Sekvens diagram for Autofuldførelse, med lag opdeling

Hver af disse lag kan så bestå af flere klasser og pakker, som det kan ses på Figur 3, som viser et package diagram for Pristjek220. Hver pakke er så inddelt under de forskellige lag (PL, BLL og DAL). Hver pakke indeholder så klasser, som har de egenskaber, som pakken beskriver. I programmet ligger de forskellige klasser under pakkens namespace.



Figur : Package diagram for Pristjek220

Ved at der er valgt at bruge en lagdelt model og under hvert lag have forskellige klasser, kommer der en separation af applikationsspecifikke tjenester fra de mere generelle tjenester, som gør SRP bliver overholdt. Derudover kommer der en separation af højniveaushandlinger fra lavniveaushandlinger, som gør, at DIP bliver overholdt. Ved at overholde SRP fås et system, som er mindre sårbart over for ændringer på et senere tidspunkt. Koblingen og afhængighederne formindskes deraf mellem de forskellige klasser, samt der kommer høj samhørighed. En af de vigtigste ting ved at bruge 3 lags modellen er dog, at der er muligheden for at genbruge kode, sådan at man for eksempelvis kan genbruge de nederste lag af kode til en applikation med en anden brugergrænseflade. Det gøres endnu mere simpelt ved, at de forskellige lag er implementeret med interfaces, som simplificerer en ændring eller udskiftning af dele af systemet. Ved at lave den logiske segmentering mellem de forskellige lag er klarheden af koden øget, for andre der skulle ønske at arbejde videre med koden.

## GUI design overvejelser

Til design af GUI’en dukkede et problem op i form af det er svært at teste den, og derfor kan koden ikke verificeres. Men der findes en løsning, i form af et MVVM[[9]](#footnote-9) pattern.

Ved brug af MVVM bliver bindingen mellem GUI’en og buisness logikken, løsnet idet at viewet primært er defineret i XAML filen, med en begrænset code-behind. Ved at binde en ICommand til view modellen, kan der kaldes funktioner fra viewet nede i view modellen og ved databindings kan der deles data. Derudover kan der oprettes events i GUI’en, som er bindet til et objekt, og lytter efter en trigger i view modellen.



Figur - MVVM model[[10]](#footnote-10)

For eksempel, i Pristjek220 kan en forretningsmanager tilføje en ny vare til sin forretning. Når varens navn, samt pris, er indtastet og der bliver klikket på ’Tilføj vare’, tager view modellen navnet og prisen og forsøger at sende dette videre til modellen. Hvis dette er succesfuldt (varen kunne jo allerede eksistere i databasen), bliver varen gemt i databasen og en bekræftelses tekst bliver sat, som et label i GUI’en er bindet til, og derved kan forretningsmanageren se at hans vare tilføjelse var en succes.

Fordi buisness logikken helt fjernes fra code-behind, bliver den uafhængig af selve GUI’en og derved bliver det muligt at udføre Unit testing på koden. Det er vigtigt at koden er testbar, fordi ellers kan der ikke garanteres at programmet fungere efter hensigten.

En anden fordel ved MVVM er, at hvis der i Pristjek220 ønskes at den generede indkøbsliste skal vise prisen i både kroner og og i euro, så er view modellen oplagt til dette. En view model’s primære opgave er at tage dataen fra modellen og formatere det således at det tilpasser viewet’s ønske. Så i stedet for at tilføje prisen i euro helt nede i databasen, bliver prisen bare konverteret til euro i view modellen og derfra bindet til en ny en kolonne i den generede indkøbsliste.

## Pristjek220 Database

### Design af databasen

I Pristjek220’s database er der fire forskellige entiteter, og de hedder Store, Product, HasA og Login. Mellem entiteterne Store og Product er der en mange-til-mange relation, da én forretning kan sælge mange produkter, og ét produkt kan blive solgt i mange forretninger. Denne relation bliver normalt selv oprettet, hvis relationen ikke har nogle andre properties. Da en forretning ikke nødvendigvis sælger et produkt til den samme pris som i andre forretninger, var det i Pristjek220 nødvendigt at have en property til produktets pris på relationen mellem forretningen og produktet. Denne property skal indeholde, hvad prisen for produktet er i lige præcis den forretning, det tilhører.

Løsningen på dette problem blev erfaret gennem undervisningen i DAB[[11]](#footnote-11). Her blev der fundet frem til, at en mange-til-mange relation, hvor der er brug for properties på relationen, skal have oprettet en entitet til relationen. Af denne grund blev entiteten HasA oprettet i Pristjek220. HasA har derfor en property til prisen, samt en relation til en forretning og et produkt, for at binde de to entiteter sammen. Derved kan Pristjek220 håndtere, at en ny forretning åbner og oprette HasA entiteter til de produkter, som er i den nye forretnings sortiment. Ligeledes, hvis en forretning får et nyt produkt i deres sortiment, vil der blive oprettet en HasA entitet mellem forretningen og produktet. Denne håndtering gør samtidig, at der kun skal være én udgave af hvert produkt og af hver forretning, da der kan bindes mange HasA entiteter mellem dem. Modellen for databasen kan ses på Figur 5.



Figur UML diagram for databasen.

Tabel : Første udkast af opsætningen af databasen

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Aldi | Fakta | Føtex | Kiwi | Rema1000 |
| Banan | 2.95 | 1.95 | 2.50 | 2.55 | 2.70 |
| Tomat | 2.90 | 2.40. | 1.95 | 2.60 | 2.30 |

Inden denne erfaring var planen med den første udgave af databasen, at den skulle indeholde en enkel tabel, hvor man kunne se produktets navn i rækkerne og de forskellige forretninger hen ad kolonerne, og på den måde finde prisen for produktet i den enkelte forretning som kan ses på Tabel 1. Dette viste sig at have nogle problemer, da Product klassen skulle laves med en variabel for hver forretning, som prisen kunne gemmes i. Det resulterede i, at databasen ikke ville være åben for udvidelser, hvilket er ineffektivt, når forretninger kan åbne og lukke, og forretningerne får nye produkter i deres sortiment.

For at forretningerne ikke kan ændre prisen på et produkt i en anden forretning, eller ændre på andre forretningers sortiment, er der lavet et login til hver forretning, og alle logins gemmes i databasen. Denne entitet har et brugernavn, kodeord, og en reference til den forretning, den har kontrol over. Når der logges ind for en forretningsmanager, har han derved kun mulighed for at ændre, hvordan hans egen forretning skal fremstå. Derudover er der lavet så Administratoren, som kan tilføje og slette forretninger, også har et login. Dette login er specielt i forhold til de andre, da det giver adgang til administrationsdelen af forretningerne. Derfor er Administratorens login blevet lavet med en forretning, der hedder Admin, og på den måde tages der i programmet højde for, hvilken retning administrationsprogrammet skal tage efter login.

### Databaseadgang

Gennem Pristjek220’s udvikling blev der erfaret, at det var besværligt at unit teste BLL da der var for hård binding mellem BLL og DAL. Det vil sige at det ikke var muligt at isolere BLL fra DAL, og da unit tests kræver at klassen, som skal testes, skal isoleres fra resten af koden, var det ikke muligt at teste BLL på denne måde. Et andet problem var, at meget af det kode, som blev skrevet, blev duplikeret, da der var brug for det forskellige steder.

Der blev derfor implementeret et Repository pattern for at separere BLL fra DAL. Derved blev det muligt at isolere BLL, så det kunne unit testes da repositoriet kan substitueres. Repository pattern kommer dog stadig med den ulempe, at det så har en hård binding ned til databasen, og ikke kan isoleres fra den. Derfor er det ikke blevet unit testet, men er i stedet blevet integrationstestet med databasen. Udover at lave en separation mellem BLL og DAL giver repositoriet også den fordel, at den laver et abstraktionslag til databasen, hvor man på den måde kan samle al adgang ned til databasen, og derved undgå at samme kode skal skrives flere steder.



Figur : Implementering af Repository pattern i Pristjek220.

På Figur 6 kan der ses, hvordan Repository pattern’et er blevet implementeret i Pristjek220. De forskellige repositories indeholder CRUD[[12]](#footnote-12) funktionerne, for den tabel de hører til. ProductRepository indeholder derved funktionerne til Product tabellen i databasen. I Repository klassen, som de specifikke repositories nedarver fra, ligger de generelle funktioner som Add og Remove for at undgå duplikeret kode. Unit of Work er lavet som et access point til repositoriesne fra BLL. Det samler alle repositoriesene i en klasse så administrationen og forbruger ikke skal have alle repositoriesene med når de oprettes. Derudover giver Unit Of Work også den fremtidsmulighed, at der kan implementeres funktioner hvor der kan tilføjes eller fjernes mange ting på én gang, uden at gemme efter hver tilføjelse.

# Produkt beskrivelse

I dette afsnit vil der blive gennemgået funktionaliterne i produktet, hvor først der vil blive det funktionaliteter de deler. Derefter vil blive fortalt om forbrugeren og så bliver der blive fortalt om administrationsdelen, hvor først der vil blive kigget på administratoren og derefter forretningen.

## 

## Delte funktionaliteter

Der er nogle delte funktionaliteter mellem administrations- og forbrugerprogrammet. Den første funktionalitet er autofuldførelse, som ligger på tekstboksene i Pristjek220. Den går ned i databasen og tjekker hvad der ligger i databasen og sammenligner det, med det som brugeren allerede har skrevet. Som man kan se på Figur 1 er der blevet skrevet ”ban”, hvor autofuldførelse så anbefaler banan, som er et produkt der ligger i Pristjek220.



Figur : Autofuldførelse

## 

## Forbruger

Forbrugeren er den primære del af produktet, og derfor er der blevet lavet flest funktionaliteter til forbrugeren. Hovedessensen er at kunne lave en indkøbsliste og genere en indkøbsliste, hvor der er blevet udregnet hvor det er billigst at handle de forskellige vare. På Figur 1 og Figur 2 kan man se et udsnit af brugerinterfacet for indkøbslisten ser ud i Pristjek220.

Figur 8: Pristjek220 indkøbsliste



Man kan se på ger i Pristjek220 vises. Den sidste funktionalitet er generer indkøbsliste, hvor pristjek udregner hvor det er billigst at handle, med den valgte indkøbsliste og indstillinger at brugeren kan skrive et produkt ind, og tilføje det til indkøbslisten. Herefter bliver produktet lagt i indkøbslisten og brugeren kan vælge antal. Herudover kan brugeren slette et produkt, eller slette hele sin indkøbsliste. På Figur 3 ligger indstillingsmuligheder, hvor brugeren så selv vælge hvilke butikker, der skal handles i. Alle forretningerne der ligger i Pristjek220 vises. Den sidste funktionalitet er generer indkøbsliste, hvor pristjek udregner hvor det er billigst at handle, med den valgte indkøbsliste og indstillinger.



Figur : Pristjek220 indkøbsliste indstillingsmuligheder

Den genereret indkøbsliste viser brugeren, hvor hver vare skal købes. På Figur 5 kan man se at indkøbslisten fra Figur 2 er indsat, med de produkter der findes i en forretningerne i Pristjek220. Produkterne der ikke findes i en forretning i Pristjek220, bliver lagt i en anden liste. På Figur 5 kan man se hvor det er billigst at købe hver enkelt vare, og på Figur 4 viser hvad det koster at købe det hele i en butik. Her kan man se at det koster 1,05 kr. mere at handle det hele i Rema100, men så spare man turen til fakta. Her kan brugeren vælge på Figur 5, vælge at klikke på forretningen og vælge en anden forretning. Derved udregner Pristjek220 en ny genereret indkøbsliste. Den sidste funktionalitet der er i den genereret indkøbsliste er på Figur 3, der er at man kan sende Indkøbslisten til sin mail.



Figur 10: Pristjek220 genereret indkøbsliste sum, besparelse og E- mail



Figur : Pristjek220 Genereret indkøbsliste



Figur : Pristje220 søg efter produkt

Den sidste funktionalitet som forbruger er, at der kan søges efter et produkt. Pristjek220 viser så hvilke butikker der sælger det, og hvad prisen er for hver forretning. Som der kan ses på Figur 6 er der blevet søgt efter ”banan”, hvor Pristjek viser hvor banan sælges.

## 

## 

## 

## Administration



Figur : Pristjek220 Login

Administration består af to dele, administrator og forretningsmanager. Når Administrations programmet bliver startet, kommer der en log ind skærm. Derefter kan man logge ind som Administrator eller Forretningsmanager. Login ses på Figur 7, hvor der både logges ind som administrator og forretningsmanager.

### Administrator

Som administrator kan man tilføje og fjerne forretninger. For at tilføje en forretning skal man angive forretningsnavn og password, derefter kan man trykke tilføj som kan ses på Figur 8. Hvor så Fakta bliver tilføjet til Pristjek220, og man kan logge ind som fakta, og tilhørende password.



Figur : Pristjek220 tilføj forretning

Man kan også slette en forretning igen som administrator, som man kan se på Figur 9. Hvor man indtaster en forretning der skal slettes, på Figur 9 er det fakta der skal fjernes. Man kan også se at autofuldførelse hjælper med at finde de forretninger, der findes i Pristjek220.



Figur : Pristjek220 fjern forretning

### Forretningsmanager

Som forretningsmanager kan man tilføje, ændre og slette vare. Hver enkelt forretningsmanager kan kun ændre på de vare, som er tilknyttet til det login, der er blevet logget ind med. For at tilføje en vare til forretningen skal man indtaste en vare og en pris, som man kan se på Figur 10. Når der er blevet trykket på tilføj produkt, kommer der så kommer en pop, hvor man skal bekræfte om oplysningerne er korrekte. Hvis produktet allerede findes i Pristjek220, bliver man opmærksom på at det allerede findes.



Figur : Pristjek220 tilføj vare

Så kan man også ændre i en vare, som ligger inde i Pristjek220, hvis nu man nu har tilbud på en vare. Der er autofuldførelse en hjælp til at vise om produktet allerede findes, så kan man ændre på prisen på produktet. Man kan se autofuldførelse funktionalitet i ændre pris på vare, på Figur 11. Hvor der bliver skrevet ”ban”, og autofuldførelse hjælper med at fortælle at der findes banan i forretningen, så prisen kan ændres.



Figur 17: Pristjek220 ændre pris på vare

Hvis man skal slette et produkt skal man angive et produkt, som kan ses på Figur 12. Hvor der er blevet skrevet ”ban” og autofuldførelse hjælper med at fortælle at banan ligger i Pristjek220. Hvis man prøver at skrive et produkt der ikke findes i forretningen, bliver brugeren informeret om det.

Figur : Pristjek220 fjern produkt



# Test af pristjek220

For at lave et godt produkt, og sikre at produktet lever op til product owner’s forventninger, bliver produktet udsat for nogle forskellige former for test. Dette er illustreret på Figur 19, hvor der kan ses de forskellige teststadier, som er unittest, integrationstest, systemtest, accept test. Gennem udviklingen af Pristjek220 er der brugt FDD, hvilket giver den fordel at man kan udvikle en feature til programmet, hvorefter der kan bestemmes om den skal sættes i produktion. Derefter kan der opstilles tests, således at den lever op til kravene. På den måde spares der en masse tid på de funktioner, som product owner beslutter sig for, han ikke vil have. TDD er modsat, der skrives test før man laver features, hvilket giver den fordel, at koden lever op til testenes kravene med det samme.



Figur : V-model for udvikling af produkt.[[13]](#footnote-13)

## Unittest

Unit testning er det første teststadie, som programmet kommer igennem, da det tester de individuelle klasser isoleret fra resten af systemet. Det vil sige at når en klasse får tilføjet en funktion, kan den testes uafhængigt af, hvor langt udviklingen af resten af systemet er. Gennem unit testning sikres der, at hver klasse individuelt opfører sig på den forventede måde ved de givne inputs, og derved bliver programmet kvalitetssikret, så det lever op til forbrugernes forventninger. Til unittest af Pristjek220 er der blevet benyttet Nunit frameworket og Nsubstitute, til at kunne opstille de forskellige testscenarier, og lave unittestene automatiske. Nsubstitute er også et framework, som bruges til at substituere de forskellige klasser, som klassen, der testes, har relationer til. Derved isoleres klassen fra resten af systemet. Det at automatisere testene giver den fordel at programmøren ikke skal bruge tid på at teste manuelt flere gange. Derudover giver automatiske test muligheden for, at koden kan optimeres efterhånden. Denne optimering kan så blive kørt igennem de automatiserede tests, uden at skulle bruge for lang tid på at teste manuelt, at funktionaliteten stadig fungere, som den skal. Automatiske unittests er dog ikke den gyldne hammer, der løser alle problemer, da de automatiserede tests kun tester, det de er lavet til at teste, hvilket vil sige, at nogle ting, kan blive overset. Derudover er der nogle ting som unittests ikke kan teste, så som brugervenlighed og hvordan den enkelte klasse opfører sig, når den sættes sammen med resten af systemet.

## Integrationstest

## (evt. Code Metrics)

## CI

## Forbrugertest

# Resultater og diskussion

# Fremtidigt arbejde

# Konklusion

# Referencer

# Underskrifter

|  |  |
| --- | --- |
| Anders Meidahl Münsberg |  |
|  | 201404246 |
| Christian Slot Winkel |  |
|  | 201370493 |
| Mette Grønbech |  |
|  | 201305561 |
| Nicklas Nielsen |  |
|  | 201370286 |
| Rasmus Brædder Hemmingsen |  |
|  | 201404469 |

1. Kravspec [↑](#footnote-ref-1)
2. Kracspec [↑](#footnote-ref-2)
3. <https://dev.windows.com/en-us/desktop/design> [↑](#footnote-ref-3)
4. https://msdn.microsoft.com/library/windows/desktop/dn742479.aspx [↑](#footnote-ref-4)
5. Jeffrey zeldman taking your talent to the web <http://takingyourtalenttotheweb.com/Taking%20Your%20Talent%20to%20the%20Web.pdf> [↑](#footnote-ref-5)
6. Jeffrey zeldman taking your talent to the web <http://takingyourtalenttotheweb.com/Taking%20Your%20Talent%20to%20the%20Web.pdf> [↑](#footnote-ref-6)
7. Ref til 4+1 I dokumentation [↑](#footnote-ref-7)
8. <https://msdn.microsoft.com/en-us/library/ff648105.aspx> [↑](#footnote-ref-8)
9. https://msdn.microsoft.com/en-us/library/hh848246.aspx?f=255&MSPPError=-2147217396 [↑](#footnote-ref-9)
10. https://msdn.microsoft.com/en-us/library/hh848246.aspx?f=255&MSPPError=-2147217396 [↑](#footnote-ref-10)
11. I4DAB - Databaser [↑](#footnote-ref-11)
12. Create, Read, Update og Delete [↑](#footnote-ref-12)
13. SWT PP – 07.1 Integration Test Pattern slide 2. [↑](#footnote-ref-13)