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

[Resumé 1](#_Toc449954373)

[Abstract 1](#_Toc449954374)

[Indledning 4](#_Toc449954375)

[Læsevejledning 4](#_Toc449954376)

[Projektformulering 4](#_Toc449954377)

[Afgrænsning 6](#_Toc449954378)

[Systembeskrivelse 7](#_Toc449954379)

[Krav 7](#_Toc449954380)

[Valg af user stories 7](#_Toc449954381)

[Aktører 8](#_Toc449954382)

[User story beskrivelser 8](#_Toc449954383)

[Ikke funktionelle krav 8](#_Toc449954384)

[Projektgennemførelse 8](#_Toc449954385)

[Iterativ udvikling 8](#_Toc449954386)

[Dokumentation af koden 8](#_Toc449954387)

[Projektstyring 9](#_Toc449954388)

[Systemarkitektur 9](#_Toc449954389)

[4+1 View Model 9](#_Toc449954390)

[Designovervejelser og beslutninger 11](#_Toc449954391)

[3-lags modellen 11](#_Toc449954392)

[MVVM 12](#_Toc449954393)

[GUI design 13](#_Toc449954394)

[Repository pattern 17](#_Toc449954395)

[Design af databasen 18](#_Toc449954396)

[Pristjek220 Forbruger 19](#_Toc449954397)

[Presentation Layer 19](#_Toc449954398)

[Business Logic Layer 19](#_Toc449954399)

[Data Access Layer 19](#_Toc449954400)

[Pristjek220 Forretning 20](#_Toc449954401)

[Presentation Layer 20](#_Toc449954402)

[Business Logic Layer 20](#_Toc449954403)

[Data Access Layer 20](#_Toc449954404)

[Resultater og diskussion 21](#_Toc449954405)

[Fremtidigt arbejde 21](#_Toc449954406)

[Konklusion 21](#_Toc449954407)

[Referencer 21](#_Toc449954408)

[Underskrifter 22](#_Toc449954409)

# Indledning

# Læsevejledning

# Projektformulering

Som forbruger er det svært at danne sig et overblik over, hvor det er billigst at handle, de ting man mangler, 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. Hvis man som forbruger gerne vil forsøge at handle billigst muligt ind, er det svært at danne et overblik over, hvor dette gøres med de mange forskellige forretninger. Der vil derfor fremstilles et program, kaldet Pristjek220, som tilstræber at give forbrugeren et let og simpelt overblik over, hvor de forskellige varer kan handles billigst.

Pristjek220 vil have 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, samt tilføjes nye varer. Baseret på disse tre brugere bliver 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 vil administratoren kunne 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 vil samtidig blive 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. Han kan også fjerne en forretning fra Pristjek220, samt en vare, såfremt der ikke længere er nogle forretninger, der fører varen.



Figur 1: Rigt billede over opslag af en vare i Pristjek220.

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, vil han, så snart han har indtastet et bogstav, blive 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 vil så informere forbrugeren om, hvor varerne fra indkøbslisten kan findes billigst, samt hvad de koster. Her vil forbrugeren samtidig have 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 vil ligeledes have mulighed for at kunne justere, hvor varerne skal købes, efter Pristjek220 er kommet med listen over, hvor det er billigst. Dette vil give 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. Eftersom den generede indkøbsliste kan give anledning til, at der skal handles i mange forskellige forretninger, er det samtidig muligt, at angive det højeste antal af forretninger man vil handle i, og hvor langt væk man vil køre for at handle. Yderligere kan det angives, hvis der er nogle forretninger, man ikke ønsker at handle i, samt om man f.eks. foretrækker økologiske varer. 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.

Når forbrugeren har fået sin genererede indkøbsliste, har han mulighed for at se den afstand, der skal tilbagelægges for at købe varerne i de forretninger, der er specificeret på indkøbslisten, samt en kørselsvejledning imellem dem. Han har samtidig mulighed for at få vist åbningstiderne for en forretning, så han derved nemmere kan planlægge sine indkøb.

Pristjek220 Forbruger vil yderligere have et afsnit til opskrifter. Her kan forbrugerne tilføje opskrifter, samt ændre og finde allerede eksisterende opskrifter. Der kan så vælges at tilføje alle ingredienserne fra en opskrift til ens indkøbsliste, hvorefter der, som tidligere beskrevet, kan genereres en indkøbsliste, så varerne kan handles billigst. Forbrugeren kan samtidig også oprette en madplan for en uge med opskrifter og generere en indkøbsliste ud fra det. I Pristjek220 Forretning er det administratoren, der kan fjerne opskrifter hvis dette ønskes.

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

Projektet er blevet afgrænset til, at det endelige produkt består af en ekstern database, så flere brugere kan tilgå de samme informationer. Databasen vil indeholde informationer omkring forretningerne og deres varesortiment, herunder åbningstider og priser. Databasens informationer kan tilgås gennem de to programmer, med tilhørende grafisk brugergrænseflade, som de forskellige brugere af systemet kan interagere med.

Projektet er afgrænset således, at de vigtigste funktionaliteter for at få et funktionelt program er prioriteret højest. Herved opnås at få et basalt program, som der ville kunne udbygges på, med de funktionaliteter der ikke er implementeret. Dette vil helt konkret sige, at brugernes muligheder i det endelige produkt, der er udviklet, er afgrænset til de funktioner, der giver produktet mest værdi. De funktionaliteter, brugerne er afgrænset til at have, beskrives i det følgende.

Forbrugeren kan i Pristjek220 Forbruger finde den billigste forretning for en vare, samt indtaste en indkøbsliste med forskellige varer og få genereret, hvor varerne kan købes billigst.

Forretningsmanageren kan tilføje varer med tilhørende priser til varesortimentet for hans forretning i Pristjek220 for at holde den opdateret.

Administratoren kan tilføje en ny forretning med tilhørende forretningsmanager.

Programmet vil køre som en computerapplikation, med mulighed for at sende indkøbslisten til brugerens E-mailindbakke. Eftersom størstedelen af mobiltelefoner understøtter E-mails vil det være let at medbringe indkøbslisten, når man handler. Programmet kommunikerer med databasen over internettet.

# Systembeskrivelse

Kort beskrivelse af systemet med et eller andet fint billede.

## Krav[[1]](#footnote-1)

### Valg af user stories

Da der skulle opbygges en kravspecifikation, blev der lavet nogle overvejelser omkring, hvordan de funktionelle krav skulle opbygges. Her var der tidligere arbejdet med use cases, men til introduktionen af semesterprojektet blev der nævnt, at der alternativt kunne benyttes user stories. Det blev derfor undersøgt, hvad forskellen på use cases og user stories var, for at kunne tage en beslutning omkring, hvilken af de to metoder gruppen følte, der passede bedst til dette projekt.

En user story beskriver et behov, som en bruger af systemet har. Det er altså derved en funktionalitet, der har værdi for brugeren, der beskrives her. Derfor skrives en user story også fra brugerens synspunkt og i et sprog, som en bruger ville benytte, og er derfor ikke fyldt med en masse tekniske termer. De er meget simpelt opbygget med en titel først, der kort fortæller, hvilken user story der er tale om. Herefter følger en kort beskrivelse, hvor behovet beskrives, gennem den handling det ønskes muligt at udføre, samt det mål der er med handlingen. Endeligt bliver der opstillet et sæt accepteringskriterier, som skal være opfyldt, for at user storien er godkendt, og behovet er imødekommet. User stories er derved meget løst beskrevne uden et højt detaljeringsniveau. Når en user story skal implementeres, ligger der herved en del indledende arbejde, idet der hertil hører en diskussion med kunden om, hvad der ønskes af den specifikke user story, samt hvilke ting der kan være problematiske, eller som ellers skal overvejes. Det er altså under denne diskussion, at detaljerne omkring implementeringen af user storien kommer på plads. Dette gør også, at kunden kan have ændret sin 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, uden at der er en masse ting, der skal laves om. Kunden har derved bedre mulighed for, at ende ud med det produkt han egentlig gerne ville have.

En use case beskriver opførslen af systemet for at imødekomme et behov. Den er mere kompleks og skrives af udvikleren i samarbejde med brugeren af systemet. Use cases stræber efter at være komplette, og de har derved en høj detaljeringsgrad fra starten af. Der forsøges herved også at afdække alle relevante undtagelser fra hovedforløbet i use casen. Use casen er bygget op således at alle rammer omkring use casen først angives, såsom målet, initieringen af den, aktører, referencer, samtidige forekomster, forudsætninger og resultatet af use casen. Efterfølgende præsenteres hovedscenariet, som er opstillet i punktform, hvor alle trinene der skal gennemføres, for at nå fra initieringen til målet, er angivet. Endeligt følger alle undtagelser fra hovedscenariet, og disse er ligeledes opgivet i punktform, med de trin der skal gennemføres. Meget af arbejdet i forbindelse med use cases ligger i starten, når der defineres krav, da de er så detaljerede. Det er altså i starten, at udvikleren sammen med brugeren afdækker kravene og detaljerne omkring, hvordan systemet skal opføre sig for at imødekomme disse krav.

Idet use cases er mere detaljerede fra starten, mens user stories først fastlægger detaljer under diskussionen inden implementeringen, egner user stories sig bedre til en agil og iterativ udvikling. Det passer derved godt til vores semesterprojekt, da viden omkring detaljerne først behøves at klarlægges lige inden implementeringen. Meget detaljerede use cases i starten af projektet kan her indebære en masse spildt arbejde, da der senere ved implementeringen kan erhverves viden, der gør, at disse detaljer skal ændres. User stories er derved på en måde mere åbne indtil inden implementeringen, hvor diskussionen omkring den afholdes. Ved at benytte use cases lægger man sig mere fast på detaljer fra starten af, og der er herved mere at ændre efterfølgende, hvis forholdene omkring implementeringen ændrer sig. Det er i gruppen vedtaget at benytte user stories i projektet. Denne beslutning er taget på baggrund af, at gruppen vurderede, at de passede bedre til en agil udvikling af et projekt på den størrelse, som vores projekt har. Der har samtidig været erfaring fra tidligere projekter med, at det hele vejen igennem projektet har været nødvendigt at ændre i use casene efter nyerhvervet viden. Derfor var der i gruppen samtidig et ønske om at undersøge andre muligheder for beskrivelse af krav for at undgå at bruge for meget tid på dette. Da der er valgt at benytte user stories, hvor en stor del af arbejdet med dem er 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 selv har diskuteret internt i starten af hver iteration omkring ønskerne og detaljerne for de specifikke user stories, der skulle implementeres i iterationen.

### Aktører

### User story beskrivelser

### Ikke funktionelle krav

# Projektgennemførelse

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

## Iterativ udvikling

Eftersom der fra IHA’s side er givet et krav om, at der skal arbejdes iterativt, 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.

Udviklingen af Pristjek220 er foregået med Scrum, fordi gruppen vurderede, at det er lettere at planlægge med på længere sigt og derved bedre til at strukturere projektforløbet i forbindelse med skolesemesteret. 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.

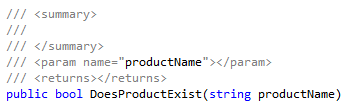
Yderligere har gruppen fra starten hældt mere mod Scrum, både fordi at sådan har arbejdsfaconen været på tidligere semesterprojekter, og alle gruppens medlemmer har gennemført et Scrum kursus på Systematic A/S.

Gruppen har afviget 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.

## Dokumentation af koden

For at lave dokumentering af source koden til projektet blev der valgt, at der skulle benyttes en dokumentationsgenerator, hvilket er et programmeringsværktøj, der genererer software dokumentation ud fra koden. Dette er en smartere løsning, end selv manuelt at skulle skrive dokumentationen for al koden. En dokumentationsgenerator sætter automatisk dokumentation op for klasser, metoder og variabler, der findes i koden, og programmøren skal blot tilføje kommentarer i en speciel syntaks i koden, til de ting der skal dokumenteres. Syntaksen af kommentarerne afhænger af, hvilken dokumentationsgenerator der benyttes. Når dokumentationen bliver genereret, sætter værktøjet selv outputtet pænt op, så det er overskueligt, og der let kan navigeres rundt i det.

Til sidste semesterprojekt blev der gjort de første erfaringer med at benytte en dokumentationsgenerator, hvor der blev benyttet Doxygen. Doxygen er fleksibelt, da det understøtter mange forskellige platforme og sprog, hvilket var nyttigt sidste semester, da der blev skrevet i C++ på en Linux platform. Til dette semesterprojekt er der derimod kun skrevet på Windows i C#, hvilket giver mulighed for at benytte Microsofts dokumentationsgenerator Sandcastle. Fordelen ved at benytte denne er, at Visual Studio allerede kender syntaksen, så når der laves tre skråstreger efter hinanden i koden, kommer der automatisk emner frem, som ville give mening at dokumentere som minimum for den klasse, metode eller variabel, man er ved at skrive til. Så hvis man var ved at skrive dokumentationskommentarerne for en metode med en returtype og én parameter, ville den foreslå, at man skrev et kort resumé af, hvad funktionen gør, skrev noget omkring returtypen, og skrev noget omkring den parameter, som funktionen skal have. Et eksempel på dette kan ses på Figur 2, hvor der, efter der er lavet tre skråstreger, er blevet genereret, at der kan skrives et resumé, noget omkring parameteren og omkring returtypen.



Figur 2: Eksempel på syntaksen på dokumentationskommentarer til Sandcastle

Når dokumentationen genereres, vil outputtets opsætning se ud, som det gør på Microsoft Developer Network. Der er derfor af disse grunde valgt at benytte Sandcastle til at dokumentere koden i dette projekt. Da Sandcastle kan være kompleks at benytte[[2]](#footnote-2) og opsætte fra bunden af, er der benyttet et værktøj til at hjælpe med opsætningen af dokumentationen. Det værktøj, der er benyttet, hedder Sandcastle Help File Builder[[3]](#footnote-3), og det laver automatisk opsætningen, og giver mulighed for at ændre på opsætningen igennem den grafiske brugergrænseflade, der hører til værktøjet.

# Projektstyring

# Systemarkitektur[[4]](#footnote-4)

## 4+1 View Model

I de tidligere semesterprojekter har der altid indgået noget hardware, som der også skulle beskrives i systemarkitekturen for projektet. Til dette semesterprojekt er der derimod kun software, så der er en masse, der førhen er blevet lavet i systemarkitekturen, som ikke skulle laves for dette projekt. Der findes derimod andre modeller, der er gode til at beskrive arkitekturen af meget softwareprægede systemer. Fra skolens side var der lagt op til, at der kunne benyttes modellen, der hedder ”4+1 View Model”.

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 deplyment view. 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.

Det logiske view viser de funktionaliteter, som systemet besidder. Der er her valgt at udarbejde klassediagrammer, da de giver det gode overblik over systemet, samt sekvensdiagrammer, for at vise hvordan kommunikationen foregår i systemet for de forskellige funktionaliteter.

I development view bliver systemet beskrevet ved at blive opdelt i mindre dele, som subsystemer og lag. Her er der valgt at lave package diagrammer, som deler systemet op i mindre pakker, og herved danner overblik over de forskellige dele, der indgår i systemet. Der er også valgt at benytte package diagrammerne til at opdele klassediagrammerne, så afhængigheder imellem klasser i forskellige pakker, og de klasser som hver pakke indeholder, kan vises på en overskuelig måde. Et diagram, der også typisk kan benyttes i dette view, er et component diagram. Det bliver typisk brugt i systemer, hvor komponenter, som andre tidligere har lavet, genbruges og sættes sammen. Dette diagram er fravalgt i dette projekt, da gruppen ikke følte det tilførte nogen værdi.

Process view fokuserer på dynamikken i systemet, og viser de forskellige processer, og hvordan de kommunikerer. Her vil der typisk blive benyttet flow charts til at illustrere dette. Gruppen var dog enige om, at flow charts ikke viste noget, som sekvensdiagrammerne i det logiske view ikke allerede viste. Der er derfor valgt ikke at udarbejde flow charts, da sekvensdiagrammerne lettere kan relateres til koden. Selve process view er derfor heller ikke implementeret i arkitekturen, da der ikke var noget, gruppen følte var relevant for projektet at vise her.

Deployment view viser, hvordan softwarekomponenter er distribueret på moduler i det fysiske plan, samt hvordan der kommunikeres mellem disse komponenter. Diagrammet, der benyttes til at illustrere dette, er et deployment diagram. Det er valgt at benytte dette diagram, da det viser, hvordan der kommunikeres mellem applikationerne, der kører på en computer og serveren.

# System Design

## Arkitektur

Da arkitekturen af projektet skulle besluttes, blev der valgt at bruge en lagdelt model, som ville give et godt og hurtigt overblik over systemet. Grunden til at valget faldt på den, var at ved den lagdelte model er der en gruppering af klasser og pakker. Hvorved at de klasser der befinder sig i ét lag har 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 og bruge en 3 lagdelt model[[5]](#footnote-5). Den 3 lagdelte model, består af; DAL (Data Access Layer), BLL (Buisness Logic Layer) og PL (Presentation Layer).

Et eksempel på hvordan et normalt scenarie kunne se ud, kan ses på Figur 3, som viser hvad der sker når en bruger indtaster i et felt der kan autofuldføre. Først ændre brugeren på teksten på GUI, derefter kalder GUI’en ned på GUI viewmodel at den skal udfylde listen, den kalder så ned på BLL, som derefter kalder ned på DAL, at den skal lave 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 4, som viser et package diagram for systemet. 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 applikations specifikke tjenester fra de generelle tjenester, som gør at SRP (Single Responsibility Principle) bliver overholdt. Ved at overholde SRP fås et system, som er mindre sårbart over for ændringer på et senere tidspunkt. Derudover kommer der en separation af højniveaushandlinger fra lavniveaus. 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 simplificere 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.

## MVVM

I forbindelse med produktets brugerinterface er der blevet anvendt et MVVM[[6]](#footnote-6) pattern. Dette er blevet valgt på grund af de fordele det medbringer.

Bindingen mellem GUI’en og buisness logikken løsnes idet at viewet primært er defineret i XAML filen, med et begrænset code-behind, som ikke indeholder noget buisness logik. Ved at binde en ICommand til view modellen, kan der kaldes funktioner fra viewet nede i view modellen. Derudover kan der oprettes events i GUI’en, som er bindet til et objekt, og lytter efter en trigger i view modellen.



Figur 5 - MVVM model

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 oprettet, 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 Euro og i kroner, 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 endnu en kolonne i den generede indkøbsliste.

## GUI design

Der er blevet brugt WPF for at designe GUI’en,Da der i 4. semester GUI fag er blevet etableret erfaring med WPF.

Pristjek220 er blevet lavet med to forskellige GUI’er; en til forbrugerne og en til administration. Administrations delen består af et login vindue hvor efter man bliver dirigeret til Forretningsmanager eller Admin afhængig af hvilket login der benyttes. På Figur 1 kan ses en simpel måde for hvordan der navigeres rundt, imellem login og Forretningsmanager og Admin for at give forståelse.



Figur - Stadiagram simpel navigation



// statediagram for Admin mangler beskrivelse



// statediagram for Forretningsmanager mangler beskrivelse

Forbruger delen har ikke noget login, og man kommer ind direkte til mainwindow’et, hvor man så kan bruge funktionaliteterne med det samme. // mangler forklaring for at den ikke har log in



// statediagram for Forbruger mangler beskrivelse

Selve designet af forretningsmanager, admin og forbruger består af et mainwindow, hvor der er user controls inde i mainwindow’et. Dette design er valgt, så funktionaliteterne skulle deles op i forskellige vinduer og gøre det overskueligt. På den måde bliver ansvaret delt op, og ikke bare det hele ligger i et samlet vindue. Der er en usercontrol for hver funktion f.eks Tilføj produkt, søg efter produkt eller indkøbsliste.

For at alle knapper er ens for programmet, er der lavet en xaml fil der beskriver hvordan knapperne skal se ud. Hvor der så bliver henvist i alle filerne til den xaml fil, så designet er ens for hele programmet. Der er så en undtagelse i alle mainwindows, hvor vi ikke bruger den samme buttonstyle som i resten af programmet. Det bliver gjort så man kan se forskel på funktions knapper og menu knapper.

## Repository pattern

Et repository pattern er et abstraktionslag til databasen for at simplificere koden. Da patternet giver et ekstra abstraktionslag, kan det benyttes på alle databaser, selv om de bruger Entity Framework, ADO.NET eller lignende, da Buisness logic layer (BLL) kalder metoder gennem interfaces. Derudover giver repository pattern mulighed for at unit teste BLL i stedet for at integrationsteste det, netop på grund af det abstraktionslag som kan mockes ud.

Der blev af disse grunde valgt at implementere et repository pattern i Pristjek220, så det netop er muligt at unit teste programmets BLL. Repository patternets abstraktionslag giver dermed også Pristjek220 mulighed for at kunne benytte sig af databaser, som ikke bruger Entity Framework. Så hvis der i fremtiden skulle blive udarbejdet noget nyere og bedre, eller der bare bliver besluttet, at det ikke skal køre på Enity Frameworket længere, så kan programmet nemt skiftes over.



Figur : Implementering af Repository pattern i Pristjek220.

På Figur 6 kan der ses hvordan Repository patterenet er blevet implementeret i projektet. Hvor de forskellige repositories indeholder CRUD[[7]](#footnote-7) funktionerne for de tabeller de hører til, hvilket vil sige ProductRepository indeholder 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 buisness logic layeret.

## Design af databasen

Planen med den første udgave af databasen var, at den skulle indeholde en enkel tabel, hvor man kunne se varens navn i rækkerne og de forskellige forretninger hen ad kolonerne, og på den måde finde prisen for varen i den enkelte forretning. Dette viste sig at have nogle problemer, da Product klassen skulle lavesmed 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 varer i deres sortiment. Der er en mange til mange relation imellem forretningerne og produkterne, da en forretning kan have mange forskellige produkter, og ét produkt kan blive solgt i mange forskellige forretninger. Der er derfor valgt at lave en klasse til relationen, da der er behov for en variabel til at indeholde prisen for produktet i den specifikke forretning. Derfor blev der oprettet 3 entiteter i databasen; Store, Product og HasA.

Klassen HasA indeholder en pris samt relationer til produktet og forretningen. Disse tre klasser gør, at databasen kan håndtere, at en forretning begynder at sælge et nyt produkt, samt at en ny forretning åbner. Måden, det er implementeret på, er ved, at Product indeholder en liste af HasA. Dette betyder, at for produkterne og forretningerne er der kun én instans, for hvert produkt eller forretning der er, og så er der en HasA imellem hver af de forretninger, som sælger ét produkt, og de kobles dermed sammen på den måde. På samme måde indeholder Store også en liste af HasA, der angiver, hvilke produkter der sælges i den forretning.

For at forretningerne ikke kan ændre prisen, eller hvilke produkter der sælges ved andre forretninger, 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, de har kontrol over. Så når der logges ind for en forretningsbestyrer, så har personen kun mulighed for at ændre, hvordan hans 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 blev Administratorens login lavet med en forretning, der hedder Admin, og på den måde tages der i programmet højde for, hvilken retning administrationsprogrammet skal tage efter login.

# Pristjek220 Forbruger

## Presentation Layer

## Business Logic Layer

## Data Access Layer

# Pristjek220 Forretning

## Presentation Layer

## Business Logic Layer

## Data Access Layer

# 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. Reference til hvor kravspecifikationen findes [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Sandcastle\_(software) [↑](#footnote-ref-2)
3. http://ewsoftware.github.io/SHFB/html/bd1ddb51-1c4f-434f-bb1a-ce2135d3a909.htm [↑](#footnote-ref-3)
4. Reference til hvor systemarkitekturen findes [↑](#footnote-ref-4)
5. Applying\_UML\_and\_Patterns\_\_\_Object-Oriented\_Analysis\_and\_Design\_\_\_3rd\_Edition\_\_Addison\_2004.10\_ Fig 13.2 / SWD Software Architecture I [↑](#footnote-ref-5)
6. https://msdn.microsoft.com/en-us/library/hh848246.aspx?f=255&MSPPError=-2147217396 [↑](#footnote-ref-6)
7. Create, Read, Update og Delete [↑](#footnote-ref-7)