DTU Compute

02350 Windows Programming

Gruppe Opgave

ØL Program

Nicoloai Søborg (s134832)
Rasmus Arpe Fogh Jensen (s134843)
Silas Sebastian Pihl (s134860)
Christian Grevil (s093434)

Teacher assistant:
Jesper Rytter Jensen

Dato:
14. december 2015

Ordliste

[Følgende forkortelser bruges gennem rapporten og i kildekoden.]

- **UC:** UserControl. Fx "AddUserUserControl" bliver til "AddUserUC".
- **VM:** ViewModel. Fx "AdminViewModel" bliver til "AdminVM".
- **OLModel:** En DLL som repræsenterer vores model.
- **ØLProgram:** Rigtige navn for programmet, som vil blive brugt i rapporten.
- **OLProgram:** Den namespace vi benytter for ØLProgram, da "Ø" kan give problemer.

Indhold

1 Indledning

Vi har udarbejdet dette projekt i forbindelse med kurset *02350 Windows Programming using C# and .NET* på DTU i efteråret 2015.

Vores projekt vil forsøge at løse et praktisk problem, der kan gavne nuværende og fremtidige DTU-studerende. Der afholdes årligt en lang række rusture og andre hytteture for de studerende på DTU. Her arrangeres der næsten altid et fælles indkøb af drikkevarer, der danner et lokalt lager under hytteturen, hvor deltagerne har mulighed for at købe fra.

I gamle dage gjorde man regnskab med papirskemaer, ved at brugeren satte en streg ud for sit navn og det pågældende produkt, hver gang han/hun foretog et køb. Til sidst kunne de regnskabs ansvarlige tælle sammen, og opkræve den enkelte for deres køb.

På hytteturene i dag er der som regel stillet en dedikeret computer op, der kører et program, som kan holde regnskab. Dette gør tab af data mindre sandsynligt og det kan gøre det lettere for brugeren at foretage et køb. Det åbner også flere muligheder for arrangørerne af turen, for at holde styr på lagerbeholdningen, se tendenser/statistikker og andet.

Det eksisterende program, der i øjeblikket benyttes på størstedelen af turene er funktionelt, men lever efter vores mening ikke op til DTU's vanlige høje standard hvad angår tekniske løsninger. Vi er overbeviste om at vi kan gøre det bedre. Målet for dette projekt er derfor at udarbejde den første version af, hvad der forhåbentligt fremover vil blive benyttet, til at holde regnskab over drikkevarer på DTU's hytteture.

1.1 Arbejdsfordeling

Vi har samarbejdet om alle dele af koden i projektet. Vi har også bidraget ligeligt til rapporten, hvor vi alle har været inde over alle afsnit.

2 Analyse

2.1 Vores domæne

Vi vil gerne udpege følgende koncepter fra vores domæne, og give en klar definition af betydningen:

- **Bruger (User):** Personer tilknyttet programmet, som har adgang til at købe drinks.
- **Admin:** Person med ekstra privilegier. En admin har mulighed for at slette/tilføje brugere og produkter med videre.
- **Produkt (Product):** Produkterne angiver de tilgængelige drinks, som er i "databasen". Et produkt er ikke nødvendigvis tilgængeligt for brugerne.
- **Kurv (Basket):** En (midlertidig) kurv, som brugeren kan fylde produkter i. Indholdet af kurven kan tilføjes til brugerens købshistorik.
- **Log:** En oversigt over købshistorik, samt en oversigt over ændringer foretaget af administratorer.

2.2 Kravspecifikation

I forbindelse med et stort programmeringsprojekt er det vigtigt at analysere sit problem og udarbejde en kravspecifikation, inden selve implementeringen begynder.

Vores kravspecifikation er delt op i to kategorier; user funktioner og admin funktioner. Løbende i processen har vores kendskab til WPF udviklet sig, og derved kan interaktionspunkter og fejlhåndtering være løst smartere end angivet i kravspecifikationen.

User funktioner

Funktion	Precondition	Postcondition	Fejlhåndtering	Interaktionspunkter
Log ind i systemet	Der skal eksistere en bruger med tilhørende password (4tal) i databasen	Brugeren er nu logget ind og kan vælge produkter	Hvis brugeren ikke eksisterer skal der komme en fejlbesked	En log ind boks
Log ud af systemet	En bruger skal være logget ind	Brugeren er nu logget ud af systemet	Hvis brugeren har tilføjet ting i kurven men ikke har tjekket ud, så skal der komme en box	En log ud knap.
Tilføj et produkt til indkøbskurv	En bruger skal være logget ind	Der tilføjes et produkt til indkøbskurven	Hvis brugeren ikke er logget på, skal der intet ske.	En knap for de forskellige produkter.
Slet et produkt af type X	En bruger skal være logget ind og have et produkt i kurven af type X	Et produkt af type X bliver slettet (og tilføjet til undo-køen)	Ignorer fejl	En slet knap.
Gang antallet af dit produkt med en "multiplier"	En bruger skal være logget ind, et produkt skal være valgt	Det valgte produkt bliver multipliceret med X	Vis fejlbesked til brugeren	En knap til multiplicering, evt. en kontekstmenu
Undo/redo	Et produkt er i kurven (undo) eller lige slettet (redo)	Et produkt bliver slettet (undo) eller tilføjet igen (redo)	Ignorer fejl	Frem / tilbage pile.
Foretag køb	En bruger skal være logget ind	Brugeren logger ud og alt i indkøbskurven bliver købt	Vis fejlbesked til brugeren	En knap.
Slet kurven	En bruger skal være logget ind.	Kurven slettes. (Skal brugeren logges ud?)	Hvis brugeren ikke er logget på, skal der intet ske.	En knap.

Admin funktioner

Funktion	Precondition	Postcondition	Fejlhåndtering	Interaktionspunkter
Tilføj/fjern varer	Admin skal være logget ind	En ny vare bliver fjernet eller tilføjet	Fejlbesked hvis man forsøger at fjerne en vare og der ikke er nogle varer registreret	En knap til både tilføj og fjern / kontekstmenu på varerlisten
Administrer varer	Admin skal være logget ind	En varer bliver ændret.	Tillad ikke at ændringer gemmes, hvis de indeholder fejl. Giv fejl-besked.	Kontekstmenu på varelisten
Tilføj/fjern bruger	Admin skal være logget ind	En ny bruger bliver fjernet eller tilføjet	Fejlbesked hvis man bruger fjern bruger og der ikke er nogle brugere registreret	En knap til både tilføj og fjern / kontekstmenu på brugerlisten
Administrer bruger	Admin skal være logget ind, minimum én bruger skal eksistere.	En bruger bliver ændret	Tillad ikke at ændringer gemmes, hvis de indeholder fejl. Giv fejl-besked.	Kontekstmenu på brugerlisten
Dan en regning	Admin skal være logget ind, der skal eksistere brugere samt der skal være et produkt scannet ind	Et Excel ark bliver generet hvor brugere betaler for hvad der er scannet	Fejlbesked hvis precondition ikke er opfyldt	Knap til dette
Load/Save data	Admin skal være logget ind	En fil gemmes med programmets tilstand (database)	Ved fejl er det meget vigtigt at informere administratoren.	En knap
Se log	Admin skal være logget ind	En log over de seneste/alle hændelser vises	Vis fejlbesked til brugeren	En knap

2.3 Arkitektur

I forbindelse med dette projekt har vi valgt at benytte arkitekturen *Model–View–ViewModel* (MVVM). Denne model gør det muligt at separere *View*, den grafiske brugergrænseflade (GUI), af vores program ved at benytte *data-bindings*, der binder til logikken i vores *ViewModel*. MVVM pattern kan ses illustreret i Figur 1.

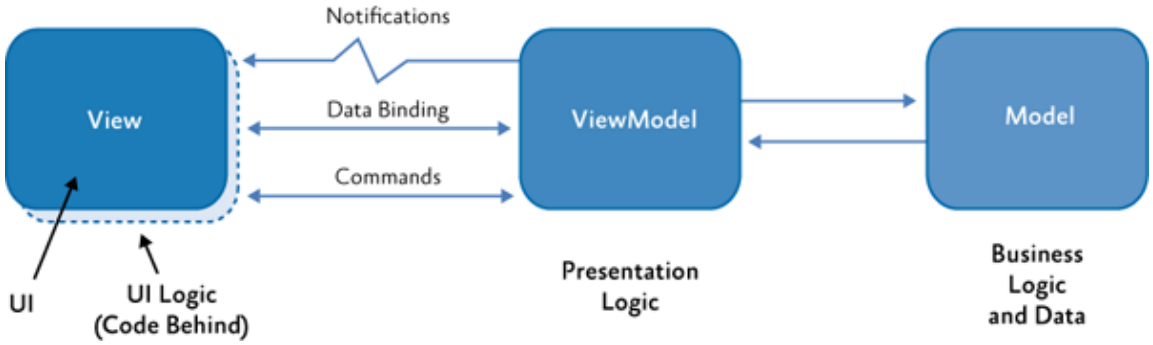
Model-delen af arkitekturen skal kun repræsentere den data som vi gerne vil manipulere i forhold til interaktion. Model bliver ofte betegnet som domænet, og man kan derfor sige at den sørger for at repræsentere domænet med de detaljer der er relevante i forhold til ens applikation.

View er vores GUI og derved den del af programmet som præsenterer vores data og alt hvad der har med udseendet at gøre. Ydereligere så sørger View delen for at interagere med brugeren. Den tager inputs såsom *key-presses* og *mouse-movements* og binder disse inputs til *properties* i ViewModel. Derved kan man sige at vores View er "aktivt" i forhold til arkitektur, hvor View udelukkende er manipuleret af en controller/presenter [1]. Ifølge Microsoft, så er den ideelle View kodet udelukkende ved XAML, med meget lidt code-behind der ikke indeholder nogen form for logik [2].

ViewModel fungerer som mellemledet mellem View og Model. Den sørger for at hente data fra modellen og repræsentere det på en måde således at det er let for View at bruge det. ViewModel er forbundet til View ved hjælp af bindings. Når en binding bliver "aktiveret" ved brugerinteraktion, så sørger ViewModel for at den nødvendige logik bliver udført samt at notificere View, hvis der er relevante ændringer, der skal vises.

MVVM arkitektur er i sær brugbart i forbindelse med *Windows Presentation Foundation* (WPF), da man kan gøre brug af alle de data binding funktioner som WPF understøtter. Fordelen ved denne arkitektur er at den code-behind som ligger bag vores GUI ikke kræver mere end initialisering. Derved kan en designer fokusere på GUI-design ved udelukkende at kende til XAML og data-bindinde til ViewModel hvor alt logikken ligger. Unit-tests er også lettere for programmøren at lave, hvis man holder View helt adskilt fra logikken.

Vi håber på, at programmet bliver videreudviklet, hvis det bliver taget i brug til sommerens rusture. Godt struktureret kode i forhold til MVVM, gør det lettere for andre mennesker at arbejde på vores applikation, hvis der kommer et behov for det.



Figur 1: MVVM illustreret [3]. ViewModel henter og manipulerer den data fra modellen, der er nødvendig. View og ViewModel er bundet sammet i form af Data-Bindings og Commands. ViewModel informerer View, når der er lavet ændringer til data, som vises i View.

MVVM Galasoft Framework

Til at implementere MVVM arkitektur har vi benyttet os af *MVVM Light Toolkit* som er et framework fra Galasoft. Dette framework gør selve processen i at bruge MVVM design pattern lettere for os udviklere. Hvis man vælger at integrere MVVM light i sit projekt, så bliver der sørget for de nødvendige biblioteker bliver inkluderet. MVVM light har mange funktioner, men i dette afsnit vil vi kort forklare hvad vi har benyttet.

MVVM light sørger for at vi bruger ViewModelBase klassen, hvilket hjælper os i forbindelse med databindings. Den sørger for vi har adgang til `RaisePropertyChanged` metoden som hjælper os med notifiable properties. Derudover så giver den os adgang til RelayCommand som vi bruger i forbindelse vores databindings mellem View og ViewModel (se mere omkring data-bindings i sektion 2.5).

2.4 Singleton Pattern

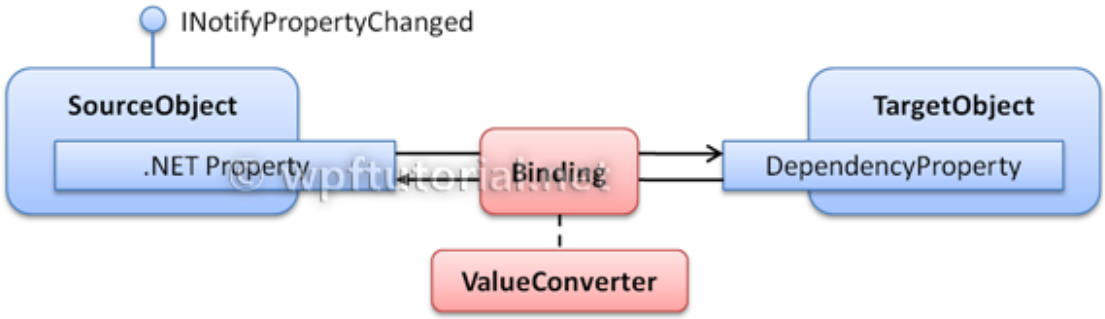
Singleton pattern er en design pattern der er brugbar når man kun skal bruge et enkelt objekt af en klasse. I forbindelse med vores projekt vil vi bruge singleton pattern til de objekter, der kun må være én instans af. En singleton klasse består af en instans af sig selv samt en private constructor.

Det gode ved at benytte et singleton pattern er at lige meget hvor meget programmøren forsøger at oprette flere objekter af en singleton, så kan der kun eksistere én. Det er altså en hjælp til programmøren.

2.5 Data-binding

Data-binding er en simpel og central del af WPF, der gør det let at håndtere præsentation- og interaktion af data. Data-binding skaber typisk forbindelsen mellem vores GUI og den logik der ligger bag koden. I MVVM arkitektkur svarer dette til forbindelsen mellem View og ViewModel.

En data-binding kan enten være *one-way* (source → target eller target → source) eller *two-way* (source ↔ target). Source er enten en .NET property eller en dependency property, hvorimod target skal være en dependency property [4]. Se figur 2.



Figur 2: På denne figur ses en two-way data-binding mellem en source og et target. Kilde: [4].

I forbindelse med forklaringen af data-binding, så er det naturligt, også at knytte nogle få ord til *dependency properties*. Dependency properties er meget advanceret og bidrager bl.a. til smart arving og bedre brug af memory. På det basale niveau, så er de properties, som vi sætter på de forskellige controls i vores UI typisk dependency properties, hvilket er et krav hvis der skal bruges data-binding.

Data-binding er typisk implementeret i XAML ved hjælp af den *markup extension* betegnet `{Binding}`. Følgende er et eksempel på en simpel data-binding; `<TextBox Text="{Binding Path=Property}">`. Her vil selve textboxen være target og `Path=Property` specificere hvad vores source skal være. En binding kan have mange felter, som kan sættes, men det er som oftest ikke nødvendigt, da de sættes til default værdier. Typisk binder man dependency properties fra controls i View til properties i vores ViewModel. Det er dog også muligt at lave bindings, der ikke har relation til ViewModel.

DataContext

En meget brugt og generel måde at benytte data-binding på, er ved at sætte en data-context, således at en control (og alle controls der nedaver fra den) har den valgte data-context. Denne property skal sættes til den data, som en control skal visualisere eller bruge. Dette er den letteste måde at få mange controls til at dele en enkelt datakilde.

Ressource Dictionary

Alle objekter har muligheden for at sætte et ressource-dictionary. Dette kan bruges til at definere en stil (eng: *style*) for bestemte controls, således at vi undgår redundant og lang kode i XAML. En binding til en lokalt defineret stil sættes ved at bruge keywordet `StaticResource`.

Change Notifications

For at data-bindings fungere optimalt, så er det oftest vigtigt at begge sider sørger for en *change notification*. Dependency properties har typisk en property-changed adfærd tilknyttet, når en relevant property bliver ændret.

Ved .NET properties er det dog ikke helt så let. Hvis man ikke selv sørger for en change notification, så bliver View ikke visuelt opdateret selvom man har etableret en data-binding. For at løse dette, så bruger man *PropertyChanged Event* fra interfacet *INotifyPropertyChanged*. Da vi har valgt at benytte Galasoft MVVM light frameworket, så har vi også mulighed for at bruge metoden `RaisedPropertyChanged`, der sørger for den relevante change notification.

Ved .NET properties er det også muligt at benytte `ObjectModel.ObservableCollection<T>` fra `System.Collections`. En oberservable collection sørger selv for en change notification når man tilføjer eller fjerner items fra listen. Dette gør det nemt at følge et "Observer Pattern", da meget af håndteringen af "opdaterings notificationer" sker automatisk.

Commands

Som programmør har man ofte lyst til at tilknytte events fra View til *buisness logic*. WPF gør dette meget let ved at introducere konceptet commands. Fra View kan man tilknytte controls eller andre interaktionspunkter en command, og binde denne til den logik som skal eksekvere i ViewModel. Dette gør at designeren kan kalde relevante commands uden at vide noget om den begivenhed, der reelt set vil blive udført. Commands resulterer i minimal code-behind og bidrager derved til den optimale opdeling af programmet i forhold til MVVM arkitektur.

ICommand og RelayCommand

Vi bruger interfacet `ICommand` fra .NET til at implementere kommandoer. `ICommands` er meget brugt sammen med MVVM, da den muliggør nem separation mellem View og ViewModel. `ICommand`'s skal implementere `CanExecute`, `Execute` og `CanExecuteChanged`. Vi benytter dog ofte kun `Execute`-metoden, så for at slippe for en masse *boilerplate*-kode benytter vi derfor `RelayCommand`, som implementerer `ICommand`-interfacet men kun tvinger os til at implementere `Execute`-metoden. Ved at benytte `RelayCommand` vil `CanExecute` (fra `ICommand`-interfacet) altid være sand og vi skal ikke tænke på `CanExecuteChanged`. Vi kan altså nu lave en `ICommand` hvor vi kun definere `Execute`.

3 Design

3.1 Model

Vores model skal beskrive den data og logik, der hører til vores domæne. Vi fokuserer på at vores model først og fremmest skal understøtte de features vi har i vores kravsspecifikation. Først når alt det grundlæggende virker som tilsigtet, kan vi begynde at overveje hvilke features, der kunne være gode at tilføje.

Eftersom at vi har valgt at vores projekt skal følge en MVVM arkitektur, bør Model være uafhængig af hvordan vores ViewModel og View ser ud. Resten af vores program har en række forventninger til vores model, som bliver beskrevet ved interfaces.

Det vil sige at modellen skal kunne skiftes ud med en anden model, som også indfrier forventningerne, således at ViewModel og View stadig kan fungere. Vi har fx valgt at vores model kan gemme og hente data i XML format, hvilket er uafhængigt af ViewModel og View. Vores model vil samtidig også kunne benyttes af et helt andet View og ViewModel, hvis det fx. blev besluttet at ændre GUI'en fuldstændigt.

Model i DLL

Vi har valgt at dele vores program op i to projekter. Det ene projekt, *OLProgram*, som bliver kompileret til en eksekverbar fil, samt *OLModel* der bliver kompileret til en DLL, som *OLProgram* bruger til at repræsentere modellen.

Idéen er at en "user" kan være mange ting, men i vores *OLProgram* har vi bare defineret et interface som beskriver at en "user" skal have et id, navn, m.m., mens vi i modellen beskriver hvordan en "user" skal gemmes i en bestemt fil.

Man kunne sagtens have inkorporeret modellen i *OLProgram*, men ved at flytte den til et separat projekt tillader vi muligheden for at andre kan kompilere deres egen DLL og repræsentere "users", "produkter", m.m., anderledes. Et eksempel kunne være at lave en model som i stedet for at serialiseres objekter til XML filer på disken, så kunne *getters* og *setters* være sat til at udføre SQL forespørgsler og gemme dataen i en database.

Typer

De centrale omdrejningspunkter i vores model er produkterne og brugerne. Det er derfor essentielt at vores model beskriver hvad hhv. en bruger og et produkt har af attributter og funktioner.

En bruger skal som minimum have et unikt ID og et navn. Der skal kunne registreres køb hos den enkelte bruger. Derudover skal man, via en bruger, kunne hente den pågældende brugers personlige købshistorik.

Et produkt skal også have et unikt ID og et navn. Derudover skal det indeholde data om lagerbeholdning, købshistorik og pris. Et produkt kan indeholde en sti, der angiver hvilket billede/logo, der skal benyttes når produktet skal repræsenteres grafisk.

Et andet centralt element i vores model er indkøbskurven. Når en bruger skal foretage køb, gøres det ved at tilføje produkter til indkøbskurven, hvorefter brugeren kan vælge at købe indholdet. En indkøbskurv skal derfor indeholde en samling af de tilføjede produkter. Man skal kunne forøge eller reducere antallet af et givet produkt i kurven.

Indkøbskurven kan godt indeholde flere af det samme produkt, men det skal være repræsenteret ved et sæt, bestående af produkt og antal. Hvis vi fx vil tilføje to af det samme produkt X til kurven, skal det ikke gøres ved at X optræder to gange i listen, men ved at vi én gang i listen, har X med et tilhørende antal sat til to.

3.2 View

Som beskrevet i 2.3, så skal skal View præsentere vores data på en flot og hensigtsmæssig måde. I vores ØLProgram er der tre centrale vinduer. En bruger skal kunne logge ind i systemet, tilføje produkter til en kurv og en administrator skal kunne administrerer systemet.

Ifølge MVVM arkitetktur, så skal View være koblet til ViewModel ved hjælp af databindings (se evt. figur 1). Vores mål er derfor et design, hvor code-behind af `Usercontrol`s egentlig bare sørger for selve initialisering.

Den autogenerede `App.xaml` er den centrale View klasse og den der bestemmer hvilken vindue der skal vises når applikationen starter. For at bestemme dette benyttes `App.xaml` `startupUri-property`, der bliver sat til et MainWindow. Som default skal View i `MainWindow` være sat til vores login `UserControl`. Vi har valgt et design således at når en bruger eller administrator logger ind i systemet, så skiftes den `UserControl` i `MainWindow` der bliver vist.

Mockup

For at få en ide om, hvordan programmet skal udforme sig visuelt er det naturligt at lave GUI mockups af programmet. Vi har visualiseret det i XAML. Vi har tre mockups, som udgør programmets fundamentale vinduer; log ind, administrator og bruger vinduet. Det vigtigste ved vores tre mockups er, at de viser hvordan vores krav fra kravspecifikationen tænkes at blive implementeret visuelt i vores program.

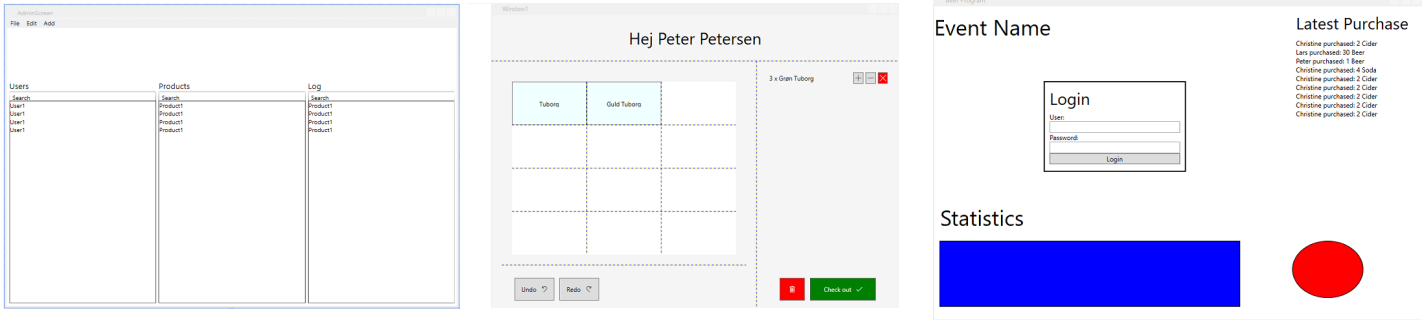
Første mockup er log ind vinduet, som er hovedvinduet brugerne bliver præsenteret for ved åbning af programmet. Udover at have en standard login boks placeret omkring midten er der en boks med sidste køb af produkter til højre i vinduet (en log). Nederst vises diverse statistikker over køb af produkter.

Næste mockup er brugervinduet (købevinduet). I denne mockup er det muligt at drage en parallel til vores kravspecifikation. Vores funktioner, der vælger produkter til en indkøbskurv, manipulere med disse ved hjælp af slet, tilføj med videre er visualiseret. En vigtig del af programmet er undo/redo, som er en del af dette vindue.

Sidste mockup er administrator vinduet. I modsætning til vores mockup af brugervinduet, hvor fokuset lå på at gøre det simpelt, er det lidt omvendt i dette vindue. De nødvendige funktioner og oplysninger, som indebærer information om brugere, produkter og en log over handlinger i programmet vises med hver deres spalte. I vores mockup er disse detaljer ikke vist, da det først blev klargjort senere og heller ikke var nødvendigt for at lave vores mockup. Vores tre nævnte mockups kan ses i Figur 3.

Element træ

Når man laver et mockup, så er element træet ikke vigtigt, da man under brainstormingen kommer til at tilføje og fjerne utallige grafiske komponenter. Visual Studio har en toolbox hvor nemt og hurtigt kan lave et GUI, men hvis man ikke holder øje med koden, så risikere man en rodet XAML kode.



(a) Admin. (b) User. (c) Login.

Figur 3: Her ses vores mockups af de forskellige dele af programmet. Se appendix A for større figurer.

Senere i udviklingen er det vigtig at element træet er overskueligt og ikke indeholder for mange niveauer af grids i grids. Overskulighed er både vigtigt for programmøren, men rækkefølgen af XAML elementerne har også forskellig betydning i forhold til hvordan de bliver præsenteret.

For at forbedre dette, kan man flytte større underelementer ud i separate `UserControl`s. Dette gør element træet meget mere simpelt, da man nu kan beskrive de fleste elementer som ét enkelt `Grid` delt i x dele, som hver indeholder en enkelt `UserControl`. Det betyder også at man kan genbruge `UserControl`s forskellige steder i koden.

I WPF er det to hovedtyper af træer; et visuelt træ og et logisk træ. Microsoft skriver: "*The distinctions between logical tree and visual tree are not always necessarily important, but they can occasionally cause issues with certain WPF subsystems and affect choices you make in markup or code*" [1]. Vi er ikke stødt på problemer og har derfor som sådan ikke tænkt på forskellen mellem disse for vores program.

DataTemplates

Vi benytter også `App.xaml` til at tilføje `DataTemplate`'s til vores `ResourceDictionary`. Dette gør det let at danne en detaljeret GUI på en ordentligt struktureret måde. Vi kan i separate `UserControl`'s beskrive hvordan elementer i vores GUI skal præsenteres. Det svarer til den klassiske brug af `toString()`. Forskellen er bare, at her fortæller vi i en `DateTemplate` hvordan en ikke-visuel klasse skal visualiseres, i stedet for hvordan den udtrykkes i en streng. Og når vi så i vores GUI vil vise et eller flere instanser af den pågældende klasse, kan vi i vores View bare tilføje dem, og så sker det helt automatisk.

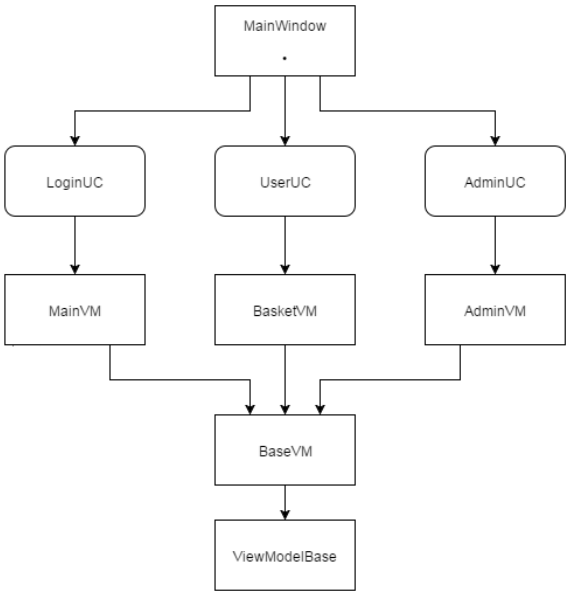
3.3 ViewModel

Det gode ved MVVM er at vi kan separere vores *Business Logic* totalt fra View, gennem ViewModel (se Figur 1). I vores tilfælde er vores Model ikke så stor, mens vores View er væsentlig mere komplekst. Dette giver os et *one-to-many* forhold mellem View og Model, og vores ViewModel skal naturlig være en brik mellem Model og View. Fra vores analyse (sektion 3.2) indså vi hurtigt at vores program har tre distinkte "områder"; main, basket og admin. Det virkede derfor også ret naturligt at oprette en ViewModel til hver; `MainVM`, `BasketVM` og `AdminVM`.

[1] Se `https://msdn.microsoft.com/library/ms753391(v=vs.100).aspx`.

Gennem *GalaSoft MvvmLight* får vi stillet en `ViewModelLocator`-ViewModel til rådighed. Idéen med denne er at et View i *run-time* får en sit `DataContext` (ViewModel) "indsprøjtet" af `ViewModelLocator`en (gennem 'refleksion') – et pattern man kalder *Inversion of Control* (IoC). Dette er dog ikke helt smerte frit, bl.a. da `ViewModelLocator` ikke fungere godt i design-mode. Vi har derfor valgt en mere lavpraktisk løsning, hvor vi opretter en ny ViewModel, `BaseVM`, som alle andre ViewModels extender.

Ved at lade `BaseVM` extende `ViewModelBase` fra GalaSoft, får alle under-ViewModels også adgang til denne. Vi ender altså ud med en struktur som beskrevet i Figur 4.



Figur 4: Overordnet design. Hvordan `MainWindow` spiller sammen med Views og ViewModels.

Undo/Redo

Undo/Redo funktionaliteten kræver to stakke, én med "undo"-handlinger og én med "redo"-handlinger. For at koordinere dette og sørge for at redo stakken bliver slettet når vi udfører en ny handling, så benytter vi et Singleton Pattern. Ved brug af dette sørger for at vi kun har netop én instans af en `UndoRedoController`.

Serialisering af Programmet

En vigtig del af programmet er at vi løbende kan tage backups af programmets data, således at hvis computeren lukker ned, så mister vi minimal data om programmets tilstand. For at gøre dette har vi gennem erfaring konstateret at hvis man løbende tager backup (fx morgen, middag og aften) så er datatab et mindre problem.

For at gemme tilstanden af programmet har vi besluttet at benytte serialisering til XML, da vi i C# har gennem .NET adgang til `System.Xml.Serialization.XmlSerializer`. `XmlSerializer` virker meget simpelt; for at serialisere gemmes alle felter et objekt har, og for deserialisering laver `XmlSerializer` et nyt "tomt" objekt (dvs initialisere med en tom konstruktor) og derefter sættes alle gemte felter manuelt på objektet. Vi har valgt dette, da `XmlSerializer` som udgangspunkt kan serialisere de fleste primitive datatyper og derfor simplificere vores kode.

Til at serialisere benytter vi os af *Asynchronous Programming* ved hjælp `C#` nøgleord som `async` og `await`. Således slipper vi for at programmet fryser, da XML-serialisering kan tage tid.[2] Til dette skal vi også bruge biblioteket `System.Threading.Threads`. Dette gør det muligt at lave *task objekter* der kører fra en anden tråd end den som selve programmet kører på. Yderligere, så kræver serialisering at man kan læse eller skrive til en XML fil. Dette kræver en file stream, hvor vi har valgt at benytte `System.IO.FileStream`.

Alternativt til serialisering kan man benytte en database, hvilket er beskrevet ydereligere i sektion 6.

3.4 Program kompatibelt med USB-Scanner

Intentionen ved dette ØL-program er at det skal være kompatibelt med en USB scanner, således at man kan købe produkter ved at scanne de tilhørende stregkoder. En USB scanner fungerer således, at når du scanner en stregkode, så bliver input fortolket som *key-events*. Scanner du en stregkode der hedder "1001", svarer det til at man taster "1+0+0+1+`Enter`" på tastaturet. Det er meningen at en bruger skal kunne bruge programmet udelukkende ved hjælp af scanner-inputs.

For at implementere dette har vi valgt at tolke de *key-events*, der er relevante. Scanner skal kun fange tal fra $0-9$ samt `Enter` fra tastaturet, hvilket gør implementeringen lettere. Vi har valgt at udnytte WPF data-bindings således at input bliver sendt til ViewModel, der kontrollere logikken. Dette kan gøres ved at benytte `System.Windows.Controls.TextBox` eller ved at sende *key-events* direkte til ViewModel, hvor relevante commands sørger for at de rigtige funktioner bliver udført.

3.5 Password box

Lige meget hvor "sikkert" vi laver vores program, så betyder præmisserne for programmets brug, at en bruger altid kan snyde systemet, ved noget så simpelt som bare at tage en øl og ikke benytte programmet til at registrere sit køb. Vi forsøger ikke at løse dette problem og erfaring siger, at dette ikke er et stort problem for de typer mennesker vi tager med på rustur. Dog ser vi det som essentielt at brugere stadig ikke har adgang til at ændre i hinandens data, eller sine egne. Vi har derfor gemt administrator delen væk, så misbrug (og forsøg derpå) bliver minimal.

For at tilgå administrator delen skal man trykke "Ctrl + K". Kommandoen til at skifte til administrator delen er bundet i vores "MainWindow" (se Figur 4), så den nedarves og kan bruges i hele resten af programmet, på samme måde som undo/redo kommandoerne.

Når man forsøger at skifte til administrator delen bliver man mødt af et `ToolWindow`, hvorefter man kan taste en global administrator kode (standard koden er "**OLProgram**") og får derefter adgang til administrator funktionalitet. Det smarte ved at lave et `ToolWindow` er at der som standard kun kan laves én instans af et sådan, så popup boksen (`ToolWindow`) vil forblive i fokus indtil man enten lukker det, eller skriver den korrekte kode (som vil skifte kontekst på MainWindow'et samt lukke ToolWindow'et).

[2] Typisk vil det nærmere være at skrive og læse fra hard-disken der tager tid.

4 Implementering

4.1 Model

Model i DLL

Den *assembly* vi laver i vores projekt er en lokal DLL. Den vil kun blive brugt i vores program, som generelt kun vil køre i én instans ad gangen, så det giver ikke mening at registere DLL'en i systemet med et *strong name* og 'spilde' plads i "Global Assembly Cache". Som standard leder vores program efter en DLL i den mappe programmet er kørt fra, inden den tjekker system mapper. Vi tager ikke højde for DLL'ens versionsnummer, men til fremtiden kan man lave avanceret versionsstyring og dynamisk vælge den korrekte DLL, alt efter hvilken version af modellen man vil benytte.

Vores model kompileres til "managed code", så sikkerhed og skraldopsamling (*garbage collection*) automatisk bliver håndteret af .NET-platformen. Man kunne sagtens tillade "unmanaged code", fx skrevet i C, men så mister meget af den indbyggede sikkerhed i .NET-platformen. Da vores program skal køre i lang tid og gerne være så stabilt som mulig, ville det være ærgerlig at tillade dette, da det hurtigt ender med fejl (memory leaks, overflows, off-by-one, osv..)

Klasser

Bruger: Vi har defineret en klasse, til at repræsentere en bruger i vores program:

```
public class User {
    // Class variables
    public int UserID { get; set; }
    public string Name { get; set; }
    public Dictionary<string,int> ProductsBought { get; }
    ...
```

klassen `User` har følgende metoder og konstruktør:

```
    // Constructor
    public User(int userID, string name){...}

    // Register purchase of product
    public void BuyProducts(string ProductID, int amountBought){...}

    // XML serialization
    public MemoryStream Serialize(){...}
    public static User Deserialize(Stream serialization){...}
```

Produkt: Vi har defineret en klasse, til at repræsentere produkterne der sælges via programmet:

```
public class Product : NotifyBase {
    // Class variables
    public string ProductId { get; set; }
    public string ProductName { get; set; }

    public int Stock { get; set; }
    public int Bought { get; set; }
    public int Price { get; set; }

    // Path to local image file
```

```
public string ImageFileName { get; }
...
```

Bemærk at klassen extender `NotifyBase`. Dette er en del af det ObserverPattern vi benytter, så vores View holdes opdateret, hvis den bagvedliggende data ændres. Klassen `Product` har følgende metoder og konstruktører:

```
// Constructors
public Product(Product product){...}
public Product(string productName, string imageFileName){...}

// XML Serialization
public MemoryStream Serialize(){...}
public static Product Deserialize(Stream serialization){...}
...
```

Produkter i kurven: Vi har defineret klassen `BasketItem`, der extender `Product`. Kurven er en samling af `BasketItem` objekter, hvor antal og produkt kan findes. Klassen er defineret således:

```
public class BasketItem : Product {
    // Private variables
    private int _count;
    private string _name;

    // Public getters and setters
    public int Count { get { return _count; } set { _count = value; NotifyPropertyChanged(); } }
    public string Name { get { return _name; } set { _name = value; NotifyPropertyChanged(); } }
    ...
```

Vi benytter private variable med public get- og set-metoder. Dette er for at registrere med et `NotifyPropertyChanged()` kald, når værdien ændrer sig i en variabel, så View kan blive opdateret.

Indkøbskurven: Vi har defineret en klasse der skal repræsentere en kurv:

```
public class Basket {
    // BasketItems currently in this basket
    public ObservableCollection<BasketItem> BasketItems { get; set; }
    ...
```

Vi gemmer samlingen af `BasketItem` objekter i en `ObservableCollection`, således at den kan holdes opdateret i viewet. Vi definerer følgende metoder og konstruktører i klassen `Basket`:

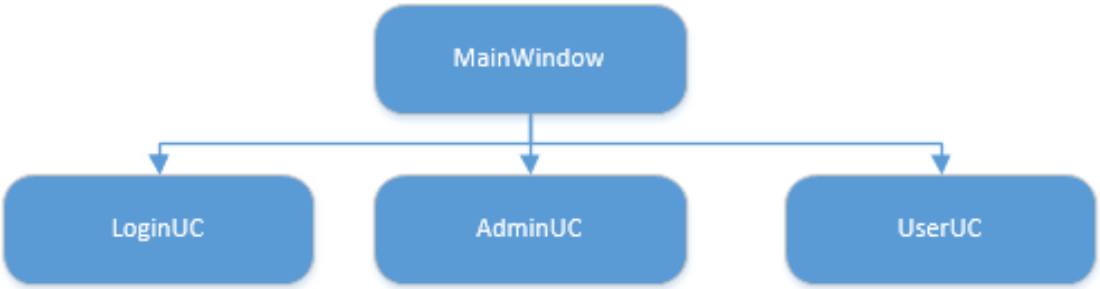
```
public Basket() { BasketItems = new ObservableCollection<BasketItem>(); }
public void Increase(Product product, int additional) {...}
public int getCount(Product product) {...}
```

Metoden `Increase` vedligeholder følgende invarianser, som også er overholdt initielt:

- To `BasketItem` objekter i kurven kan ikke have samme `ProductId`
- Alle `BasketItem` objekter i kurven har et antal (`Count`) større end nul

4.2 View

Som beskrevet i afsnit 3.2 har vi implementeret et `MainWindow.xaml` samt tre `UserControls`: `LoginUC.xaml`, `AdminUC.xaml` og `UserUC.xaml`. Alle disse er koblet sammen ved hjælp af controls fra WPF. Nedarvning og visuelisering af dette kan ses i afsnit 4.2



Figur 5: Elementtræ for vores MainWindow

I vores endelige vinduer og program fik vi brugt flere responsive GUI elementer. I flere af vinduerne har vi valgt at have en menulinje øverst, som giver mulighed for at tilføje avancerede funktioner, uden at gå på kompromis med det simple udseende. I admin vinduet har vi brugt `DataGrid`, som gør det let og effektivt at ændre værdier i *Run-time*, bl.a. ved brug af *tab* eller *enter* til at navigere rundt mellem felterne. I brugervinduet har vi gjort brug af `UniformGrid`, som intelligent placerer elementerne pænt. Det er smart, da vi ikke på forhånd kender antallet af produkter tilgængelige. Dertil har vi også flere steder brugt `ScrollViewer`. Flere detaljer om UI opbygningen kan ses i Section 4.2. Alle vores vinduer er vist i Appendix C.

Element træ

Vi har valgt at vise element træ'er for vores centrale `UserControl`'s `LoginUC.xaml`, `AdminUC.xaml` og `UserUC.xaml`.

Øverst i vores UI er MainWindow, hvortil de tre nævnte UserControls lægger sig på alt efter hvilket vindue, som skal vises. Se Figur 5, hvor detaljer for UserControls er udeladt.

Vi har taget de væsentlige grafiske elementer med og visualiseret elementræerne for de tre UserControls hver for sig. Se Figur 6.

Se de to sidste elementtræer i Appendix B.

DataTemplates

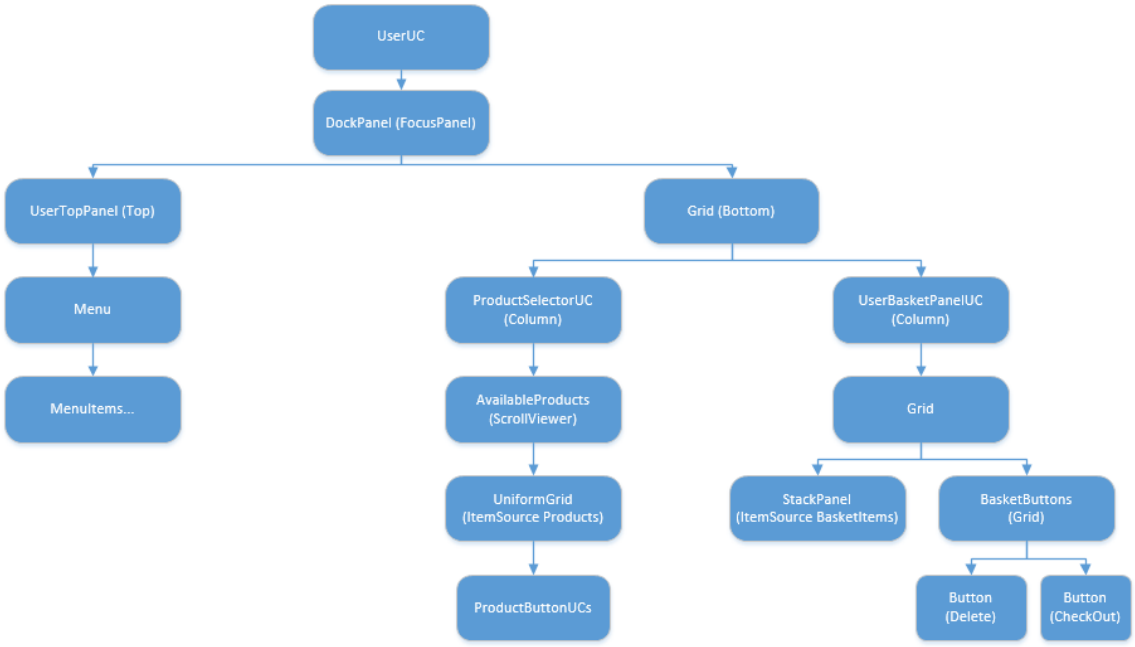
Vi implementerer `DataTemplate`'s helt simpelt, ved at have separate `UserControl`'s for de forskellige visuelle elementer. Vi opsætter `DataTemplate`'s ved f.eks. at tilføje følgende linjer til `ResourceDictionary` i `App.xaml`:

```
<DataTemplate DataType="{x:Type Model:BasketItem}">
    <View:BasketItemUC/>
</DataTemplate>
```

I `BasketItemUC` kan vi så definere hvordan et `BasketItem` fra modellen skal vises. Se Figur 7 og Figur 8.

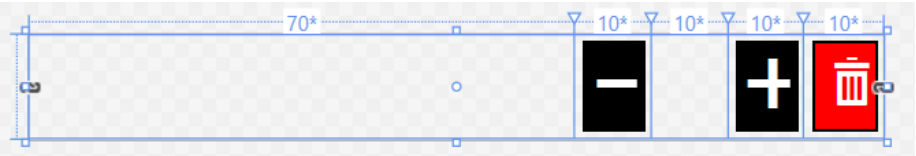
Vi tilføjer disse elementer til vores GUI ved at binde et `ItemsControl` op på `Basket.BasketItems` fra vores ViewModel. De `BasketItem`'s der vises i Figur 7 og 8 er tilføjet på følgende måde i vores View:

...

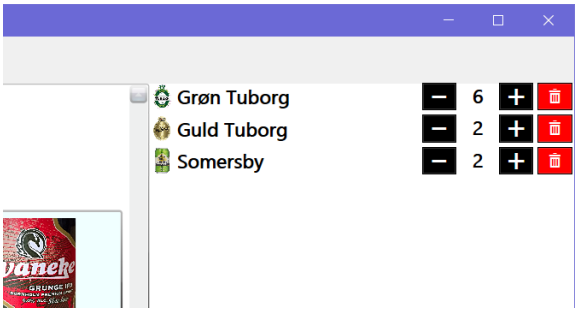


Figur 6: Elementtræ for UserUC

```
<ItemsControl ItemsSource="{Binding Basket.BasketItems}"/>
...
```



Figur 7: En `UserControl` kaldet `BasketItemUC` der definerer hvordan et produkt i kurven vises.



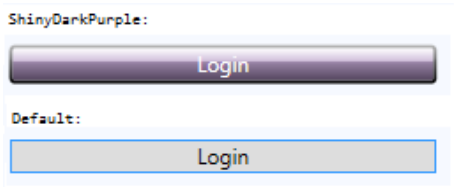
Figur 8: Hvordan produkterne i kurven vises.

Design ved hjælp af Microsoft tema

For at gøre vores GUI pænere, har vi valgt at benytte et Microsoft tema ved navn *ShinyDarkPurple*. Dette gør blandt andet at alle buttons får et lækkert design, som kan ses i figur 9.

Dette er implementeret i `App.xaml` ved at tilføje `ShinyDarkPurple.xaml` som et resource dictionary.

```
...
<ResourceDictionary Source="/Themes/ShinyDarkPurple.xaml"/>
...
```



Figur 9: Login button med Microsoft tema i forhold til default.

4.3 ViewModel

Som beskrevet i sektion 3.3, så benytter vi en `BaseVM` til at "samle" data på tværs af ViewModels. Dette er muligt, da vi i `BaseVM` kan lave statiske variabler til data der skal deles, selvom instansen af `BaseVM` er forskellig. Ved brug af et Singleton Pattern til fx `UndoRedoController`en er det meget nemt at sørger for at der kun findes én instans af denne.

Vores `BaseVM` holdes så vidt muligt så lille som mulig, så *performance overhead* er minimalt. Ideen er så, at hver skærm har en logik og data, som kun er gældende i den skærm. Derfor har vi lavet et `ViewModel` til hver af de tre store *UserControls*.

Undo/Redo

For at implementere Undo/Redo funktionaliteten for mange forskellige kommandoer benytter vi et interface, `IUndoRedoCommand`, som specificere hvad alle '*fortrydelige*'-kommandoer skal implementere. Vi er inspireret af demoen som ganske simpelt kræver at kommandoer implementere `void Execute()` og `void UnExecute()`.

For at udføre en handling benytter vi `undoRedoController.AddAndExecute(IUndoRedoCommand command)`. Som beskrevet i analysen, så vil der altid kun være én `undoRedoController`.

For at håndhæve "Singelton Pattern" har `UndoRedoController` en privat konstruktor, så det kun er den selv der kan lave en `new UndoRedoController()`. Andre objekter har kun adgang via `public static UndoRedoController Instance { get; }`.

Serialisering af Programmet

Vi er interesseret i at gemme Product objekter, User objekter og de logs programmet løbende danner. I forbindelse med dette har vi lavet en klasse i vores model ved navn `Data.cs`. Denne klasse indeholder lister, der repræsentere det vi vil skrive til XML. Dette gør det let fra vores ViewModel, at skrive vores ObservableCollections til et data objekt og derved serialiserer data-objektet. `User.cs` bruger et `System.Collections.Generic.Dictionary<TKey,TValue>` til at holde styr på hvilke produkter en User har købt samt hvor mange. Dictionary kan ikke serialiseres til XML og derfor er vi nød til at lave et "Work-around". Vores løsning går ud på at skrive alle keys og values for alle Users til to lister. Yderligere så skal vi have en liste der holder styr på antallet af forskellige produkter der er købt for hver user. Ved hjælp af disse tre lister kan vi repræsenterer data for alle brugerenes dictionaries.

Her ses vores data-objekt som er det der bliver serialiseret til XML.

```
[Serializable]
   public class Data
   {
       public List<User> Users { get; set; }
       public List<Product> Products { get; set; }
       public List<String> ProductKeys { get; set; }
       public List<int> AmountBought { get; set; }
       public List<int> ProductsForEachUser { get; set; }
       public List<String> AdminLog { get; set; }
       public List<String> UserLog { get; set; }
   }
```

Når vi enten skal serialisere eller deserialisere så kræves der en stig, enten til en eksisterende XML fil der skal deserialiseres eller en stig til hvor en ny fil skal oprettes. Denne dialog med brugeren bliver håndteret i hjælper klassen `DialogHelper.cs` ved hjælp af Microsofts objekter `OpenFileDialog()` og `SaveFileDialog()`. Denne dialog med brugeren er håndteret i `AdminVM.cs`.

Til selve serialiseringen har vi lavet hjælper klassen `SerializerXML.cs`. Denne klasse er implementeret ved hjælp af en Singleton Pattern (se sektion 2.4), da vi ikke er interesseret i at have flere instancer af denne klasse.

For at serialisere skal hjælperklassen bruge et data objekt samt en stig til hvor den generede XML fil skal gemmes. Data-objektet og stigen bliver generet i `AdminVM.cs` og sendt til `SerializerXML.cs`. Koden til serialiseringen kan ses nedenfor.

```
public async void AsyncSerializeToFile(Data data, string path)
   {
       await Task.Run(() => SerializeToFile(data, path));
   }

   private void SerializeToFile(Data data, string path)
   {
       using (FileStream stream = File.Create(path))
       {
           XmlSerializer serializer = new XmlSerializer(typeof(Data));
           serializer.Serialize(stream, data);
       }
   }
```

Når vi deserialisere skal vi give `SerializerXML.cs` en stig til den XML vi vil load ind i programmet. Deserialiseringen returnere et data-objekt til `AdminVM.cs`, der selv pakker dataen ind i de korrekte observable collections. Koden for deserialiseringen minder meget serialisering og er derfor ikke inkluderet.

4.4 Scanner

I programmet er der to steder hvor scanneren kan tages i brug; `LoginUC` og `BasketUC`.

LoginUC

I `LoginUC` skal en bruger kunne scanne sit tilhørende brugerID, således at han går direkte videre til `BasketUC`, hvor produkter kan købes. Dette er implementeret ved hjælp af en texbox, der

altid vil have fokus. Så snart *key-eventet* `Enter` bliver registreret, så sendes textboxens input til ViewModel, der finder ud af hvorvidt det er et gyldigt brugerID. Hvis gyldigt, så skiftes `LoginUC` til `BasketUC`. Hvis ikke gyldigt, så genstarter textboxen, og man kan prøve at logge ind igen. Dette gør programmets login `UserControl` kompatibelt med scanner inputs i forhold til scannerbeskrivelsen i sektion 3.4

BasketUC

I `BasketUC` skal en bruger kunne scanne produktkoden på det produkt, han gerne vil købe samt logge ud ved at scanne sit eget brugerID igen. For at håndtere dette, så sender vi alle *key-events* i denne UserControl til Viewmodel. ViewModel sørger for at behandle det indsendte input, når `Enter` bliver sendt. Derved tolker ViewModel indputtet for at bedømme hvad der skal ske.

4.5 Password Box

Et problem vi oplevede ved at benytte en `PasswordBox`, er at – modsat en `TextBox` hvor man nemt kan binde til `TextBox.Text` – så kan man ikke lave en data-binding til `PasswordBox.Password`! Dette skyldes at Microsoft ikke har lavet `Password` en *dependency property* som en sikkerhedsmekanisme. Til vores brug er dette dog meget overdreven sikkerhed og det tvinger os væk fra MVVM-arkitekturen. For at undgå dette benytter vi os af en *PasswordHelper* som vedhæfter sig PasswordBox'en og blotlægger kodeordet. Det giver en fin sikkerhed til vores *angrebsmodel* og sikre mest muligt MVVM.

For at gemme kodeordet permanent, uden at hardkode det ind i applikationen, benytter vi den indbyggede "Settings"-funktionalitet. Dette sikre at hvis man ændre kodeordet, så stadig være ændret når programmet bliver genstartet. Vi har valgt at "Settings" skal gemmes under "`Scope: Application`", så den samme bruger kan have flere forskellige instanser af programmet med forskellige kode (fx et til test, et fra en gammel rustur og et her-og-nu).

Vi skal dog stadig passe på, fordi .NET betragter ikke "Settings" som værende sikkerhedskritiske, så de er gemt ukrypteret. For at man ikke skal kunne finde frem til administrator-kodeordet, gemmer vi kodeordet på forsvarligvis, gennem en kryptografisk hash funktion. Mere bestemt benytter vi PBKDF2 (*Password-Based Key Derivation Function 2*) som i .NET hedder `Rfc2898DeriveBytes`.

5 Konklusion

Vi har udviklet hvad man kan kalde alpha versionen af ØLProgrammet, og vi har dokumenteret udviklings processen i denne rapport. Hvis vi holder vores nuværende produkt op imod kravspecifikationen fra afsnit 2.2, ses det at alle funktioner fra vores kravspecifikation er implementeret i løsningen. Vi har ændret nogle små ting hist og her, primært ift. interaktionspunkter og fejlhåndtering, fordi vi løbende har arbejdet på at gøre GUI'en intuitiv og effektiv.

Vi har hele vejen igennem haft et stort fokus på at overholde arkitekturen. Dette skyldes især at vores mål er, at programmet tages i brug på DTU's rusture, og det skal derfor være muligt for andre end os, at videreudvikle på projektet. Dette er besværligt til tider, men det betaler sig ofte tilbage på den lange bane, da projektet bliver mere robust og lettere at vedligeholde.

Vi er klar over at modellen i dette projekt er meget simpel. Men det har efter vores mening været en stor fordel for vores læringproces. Vi har ikke skulle sætte specielt meget tid af til at løse komplicerede problemer i modellen, som lige så godt kunne være løst i Java eller andet. Vi har i stedet benyttet langt størstedelen af tiden på, at arbejde med de værktøjer og biblioteker, som

.NET, WPF og MVVM Light har stillet til rådighed. De stiller rigtig mange til rådighed, og vi benytter kun en meget lille del, men mange af principperne vi har benyttet er generelle og går igen. Vi har undervejs i projektet fået øjnene op for mange af de fordele, der er ved at arbejde med WPF og .NET. Det er f.eks. super godt, at designere kan arbejde på XAML, uden at have en videre forståelse af business logic delen.

Alt i alt er vi rigtig godt tilfredse med resultatet. Vi vil videreudvikle på projektet, så det er klar til at blive taget i brug til sommeren rusture.

6 Videre udvikling af projektet

Et projekt som et ØL-program kan videreudvikles på mange forskellige måder. Der vil nærmest altid være "ekstra features", som nogle brugere vil finde interessante og derved ville kunne tilføjes til projektet.

I dette afsnit har vi valgt at fokusere på nogle af de få ændringer, som vi mener, ville være interessante i forhold til programmets forhåbentligt fremtidige brug på DTU's rusture.

6.1 Integration med en Server

For at forhindre at den tilknyttet computer "crasher" og data dermed bliver tabt, så kunne man integrere løbende backup til en server online.

Den data som serveren modtager, ville kunne bruges til at lave statistik over ALLE rusture, der vælger at benytte vores øl-program. Dette vil man kunne bruge til at finde interessant fakta, såsom hvilken øl er DTU studerendes yndlingsøl? Fakta som umiddelbart kan virke ubrugeligt, men som kan gavne fremtidige øl-bestillinger til rusture de mange kommende år.

Serveren kunne også stå for at hente data, såsom billeder til de forskellige produkter ind i programmet. Programmet kan evt. have en lille database af tilgængelige produkter, således at den foreslår disse ved *auto-completion*, når man ændrer `Product.Name`. Hvis man vælger et navn fra den lokale database, så er der et billede tilgængeligt på serveren, som kan blive loaded ind i programmet. Hvis udvalget af produkter er meget stort, kan dette være en stor optimering.

6.2 Importer data

Programmet understøtter importerring af data, hvis det er gemt som XML. Ofte er man interesseret i at importerer brugere eller produkter fra fx. en Excel fil. Dette spare den ansvarlige administrator meget "slave" arbejde og vil derfor være en højprioritet på videreudviklingen af programmet.

Litteratur

[1] Likness, Jeremy (2010): "*Model-View-ViewModel (MVVM) Explained*"
Link: `http://www.codeproject.com/Articles/100175/Model-View-ViewModel-MVVM-Explained` (hentet den 14. december 2015).

[2] Microsoft: "*The MVVM Pattern*"
Link: `https://msdn.microsoft.com/en-us/library/hh848246.aspx` (hentet den 14. december 2015).

[3] Microsoft: "*Patterns & practices*"
Figur taget fra URL: `https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx` (hentet den 14. december 2015).

[4] Moser, Christian (2011): "*DataBinding in WPF*"
Link: `http://www.wpftutorial.net/DataBindingOverview.html` (hentet den 14. december 2015).

Appendix

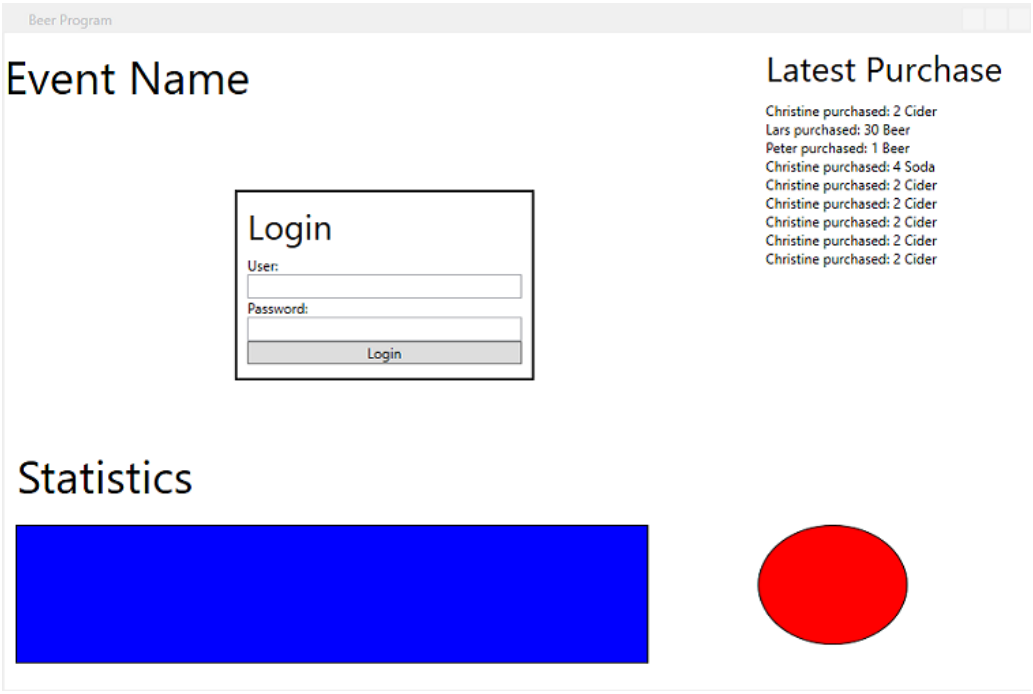
Følgende appendix er vedhæftet denne rapport:

A **Mockups** – En liste af billeder over vores mockups.

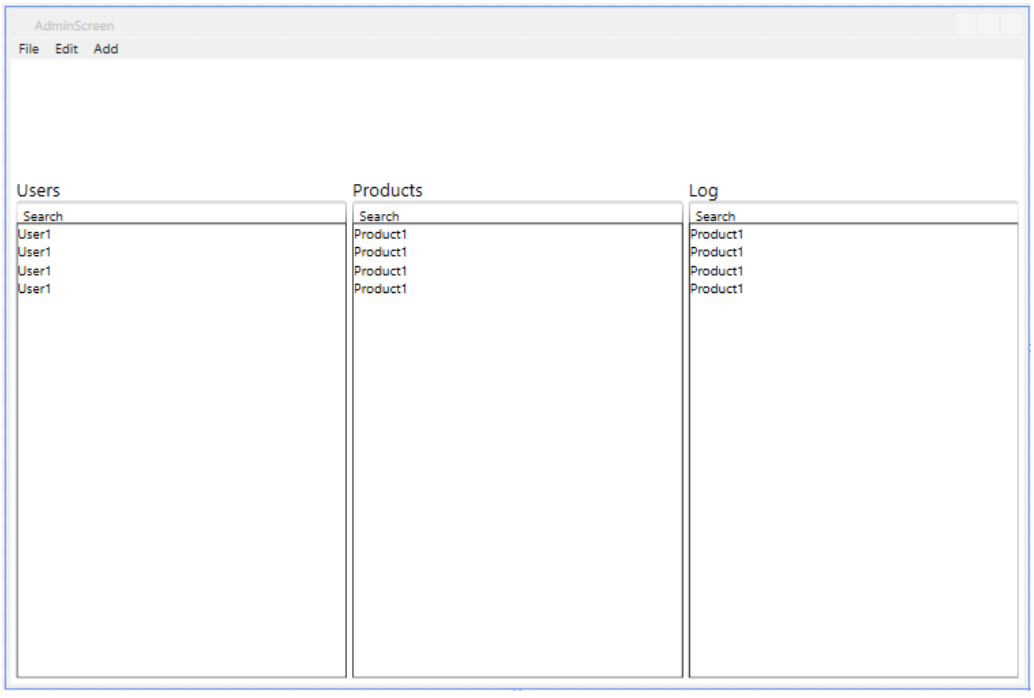
B **Elementtræer** – De resterende elementtræer, som ikke blev vist i brødteksten.

C **ØLProgram** – En liste af billeder over vores vinduer i programmet, som de endte med at se ud.

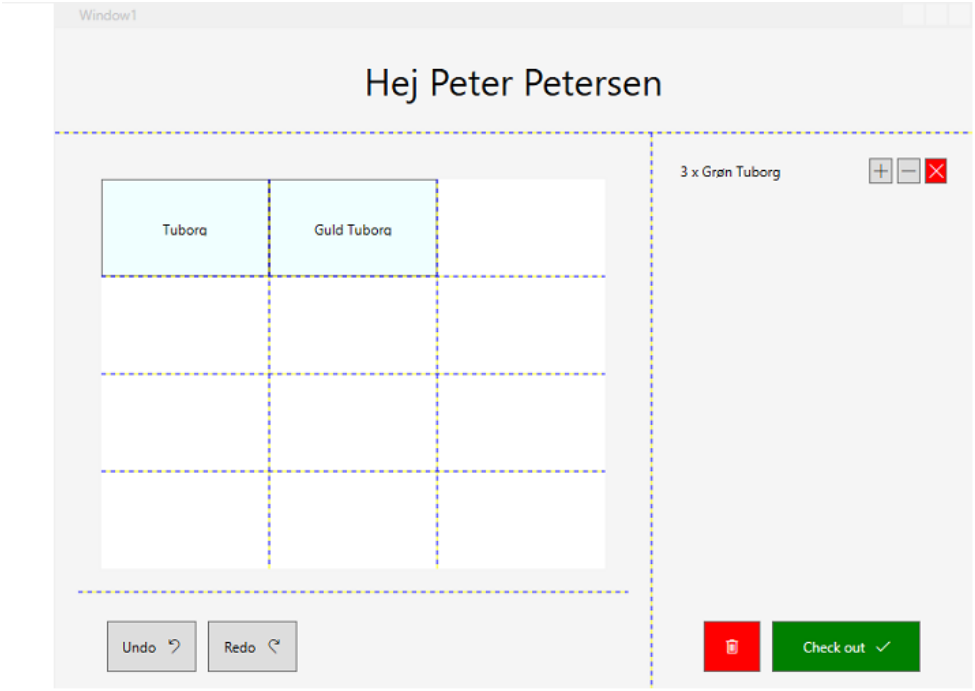
A Mockups



Figur 10: Log ind vinduet (LoginUC).

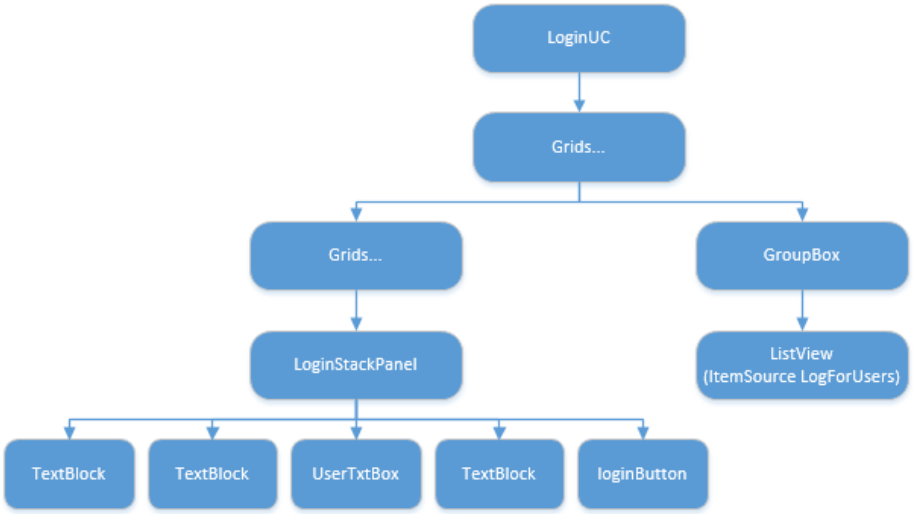


Figur 11: Admin vinduet (AdminUC).

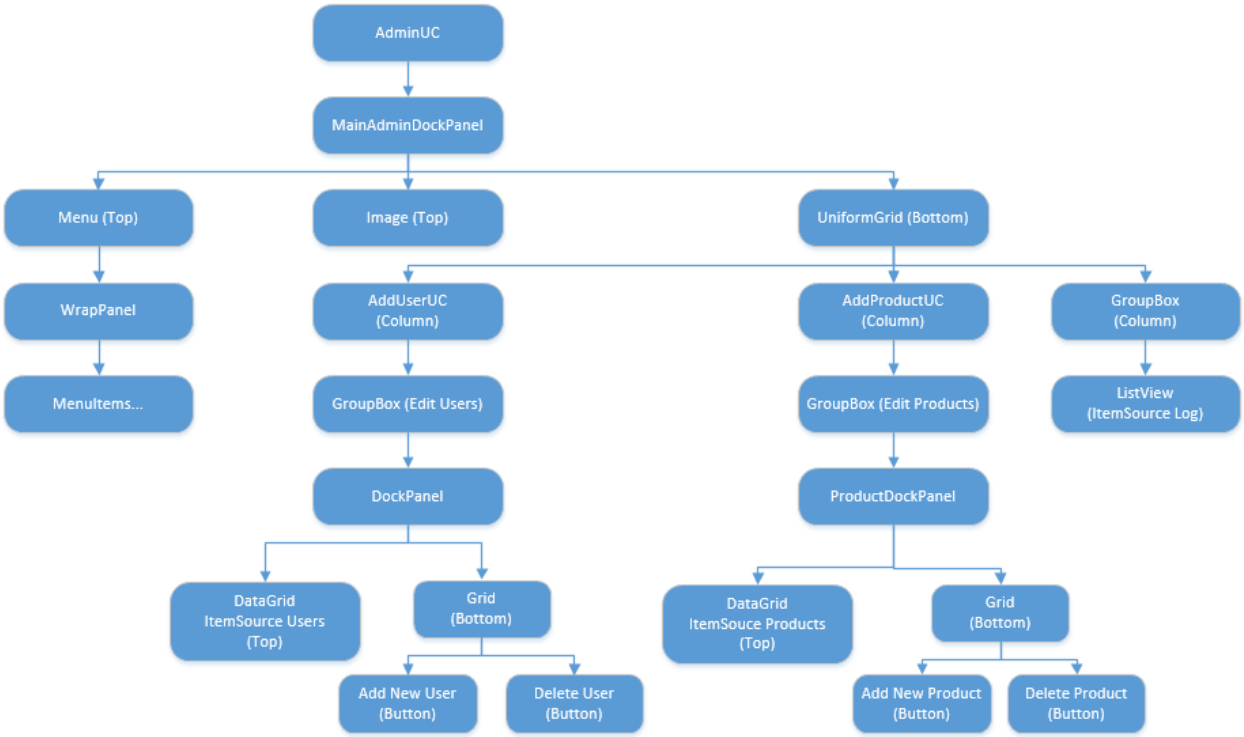


Figur 12: Mockup af brugervinduet (UserUC).

B Elementtræer



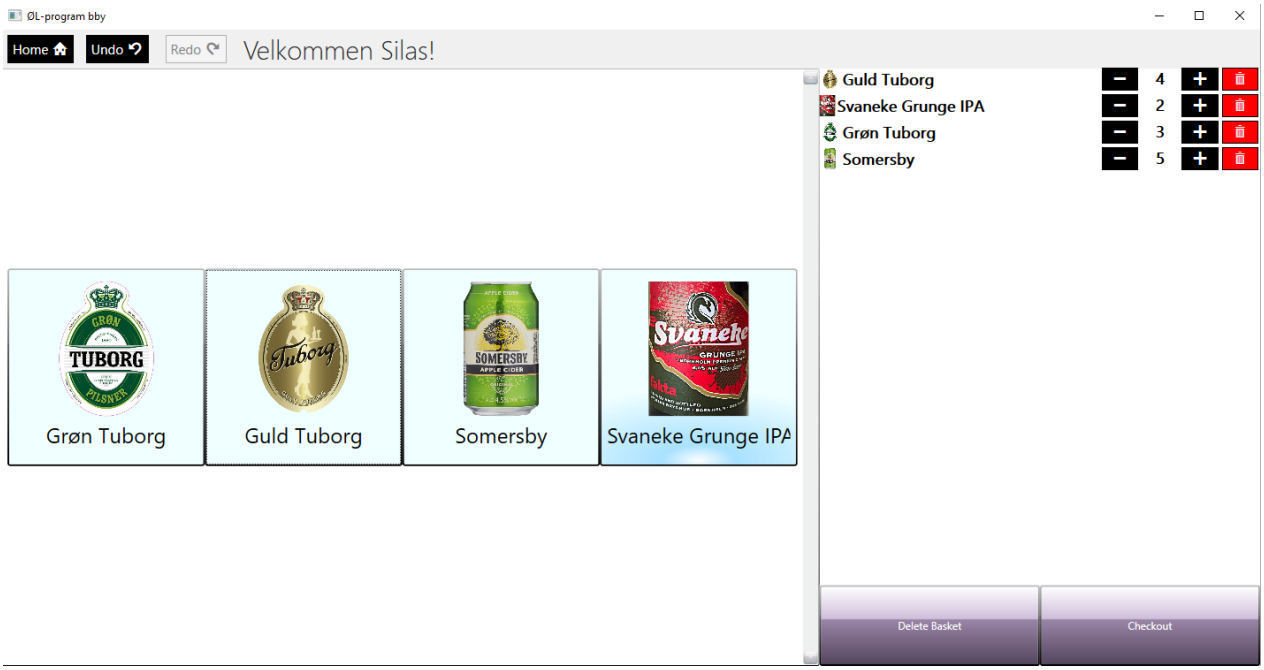
Figur 13: Visuelt element træ for LoginUC.



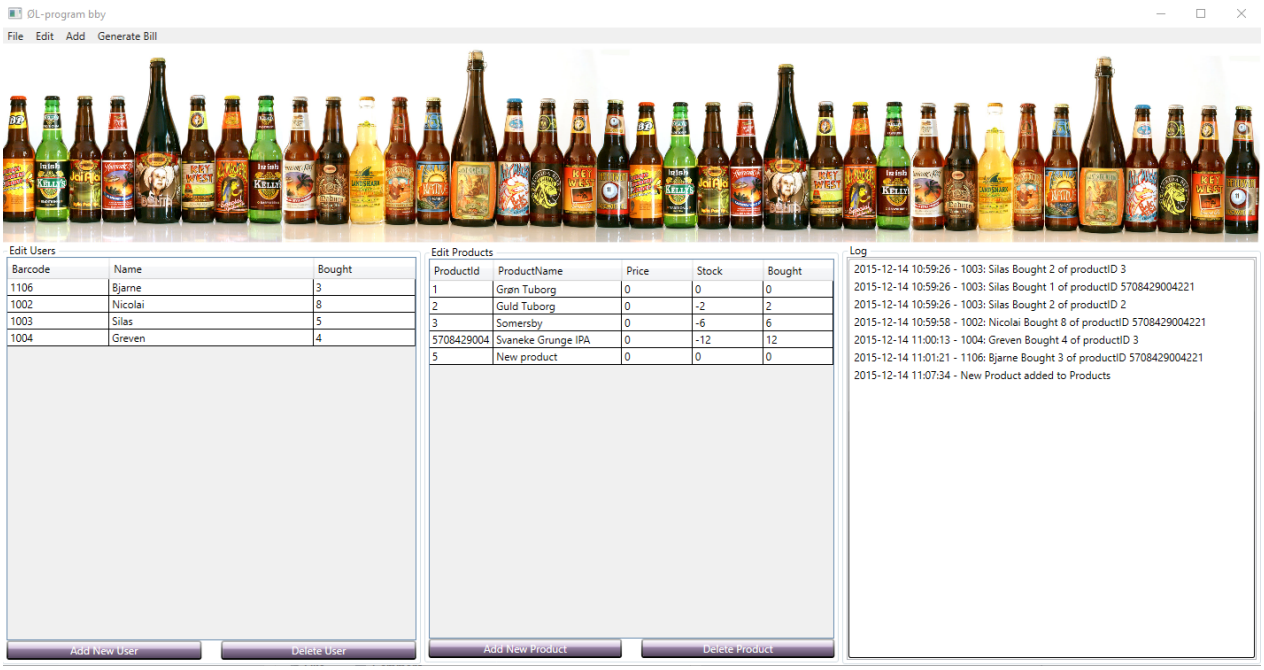
Figur 14: Visuelt elementtræ for AdminUC.

C ØLProgram

Figur 15: Log ind skærmen.



Figur 16: Bruger skærmen til indkøb.



Figur 17: Admin skærmen.