# Design og implementering

## Overoverdnet design

*Deployment*-diagrammet i **Figur 1** viser, hvordan de interne dele af Smart Fridge interagerer. Som det fremgår, interagerer Bruger kun med enten en *Fridge app* eller *Web app*, og har altså ingen direkte interaktion med den bagvedliggende logik.



**Figur 1** Deployment Diagram for hele SmartFridge-systemet

For et mere detaljeret *Deployment*-diagram, henvises til **bilag XX**.

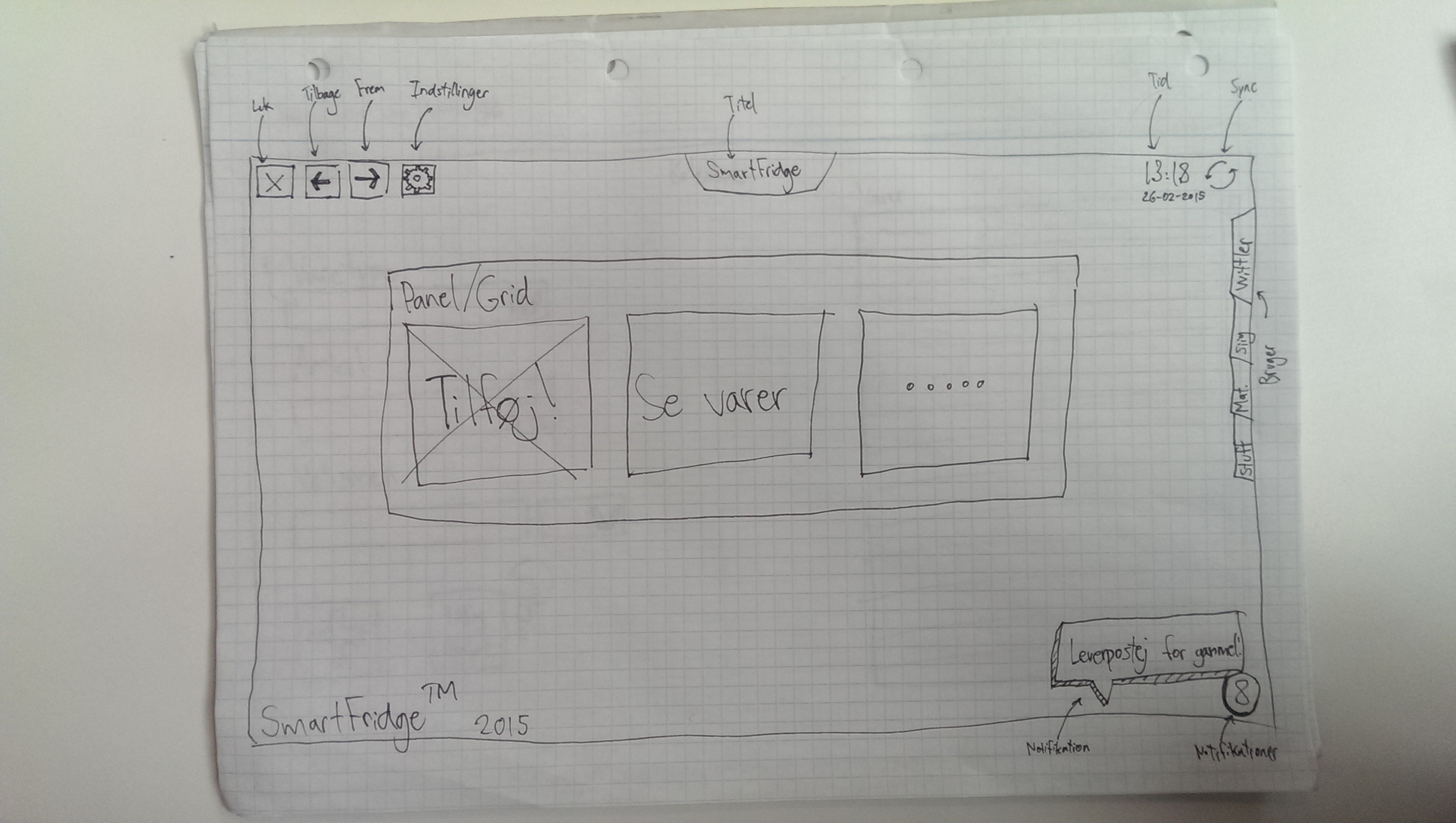
## Fridge app

### Design

Dette afsnit indeholder de grafiske og kodemæssige designovervejelser for de forskellige vinduer i Fridge App. Overvejelserne er baseret på en brainstorm (**bilag XX**). For større billeder, henvises til **bilag XX**.

#### Hovedmenu og ramme

Designet af hovedmenuen og rammen er blevet skitseret som i **Figur 2**.



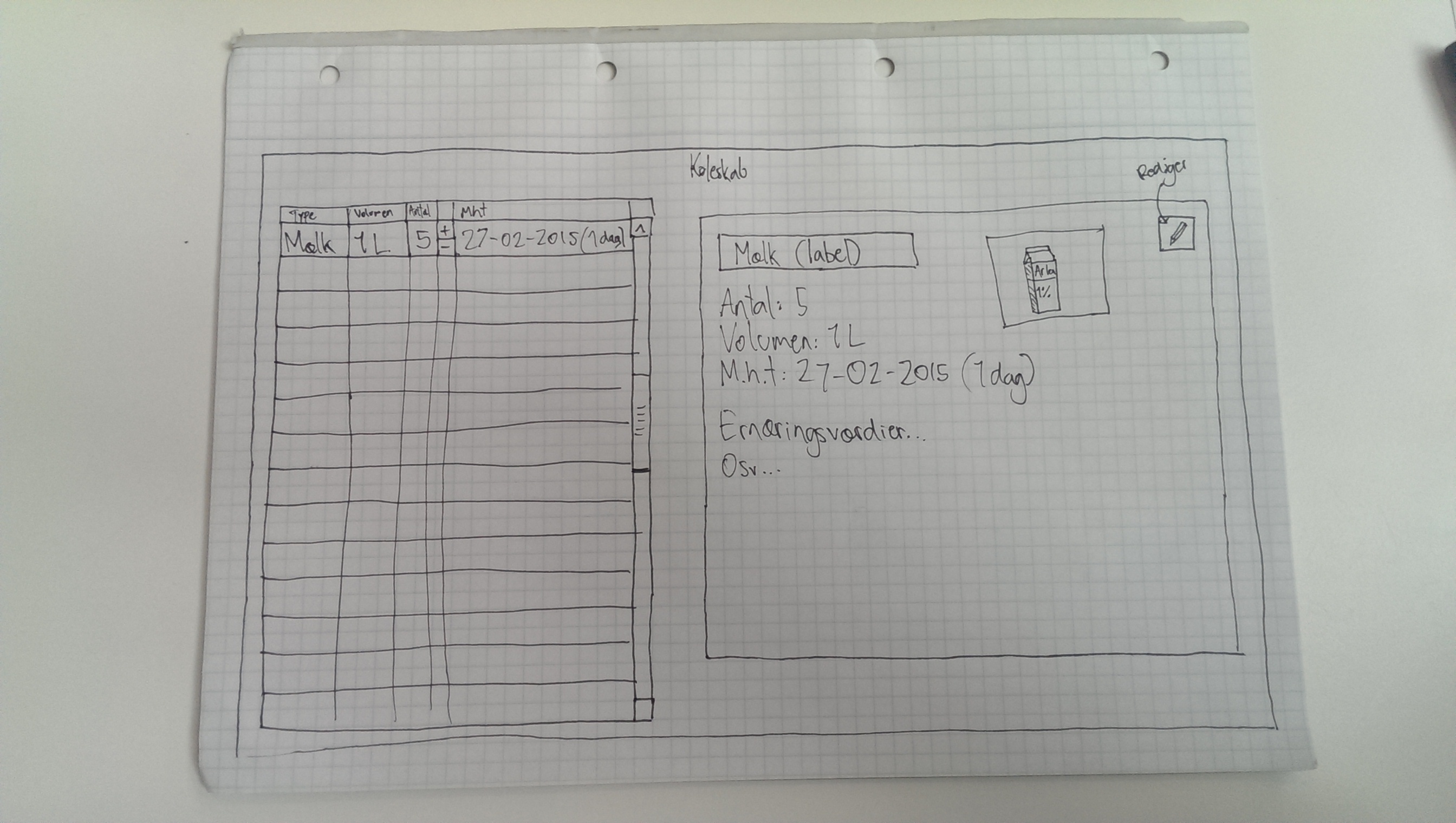
**Figur 2** Skitse af hovedmenu og ramme

Rammen, som er alt omkring firkanten *Panel/Grid*, vil ikke ændres, når der ændres kontekst.

En senere ændring, i forhold til skitsen, har været at knapperne i *Panel/Grid* vil være direkte henvisende til de eksisterende lister, hvorfra det vil være muligt at tilføje, redigere og fjerne varer direkte.

#### Se varer

Designet af menuen for “Se varer” er blevet skitseret som i **Figur 3**.



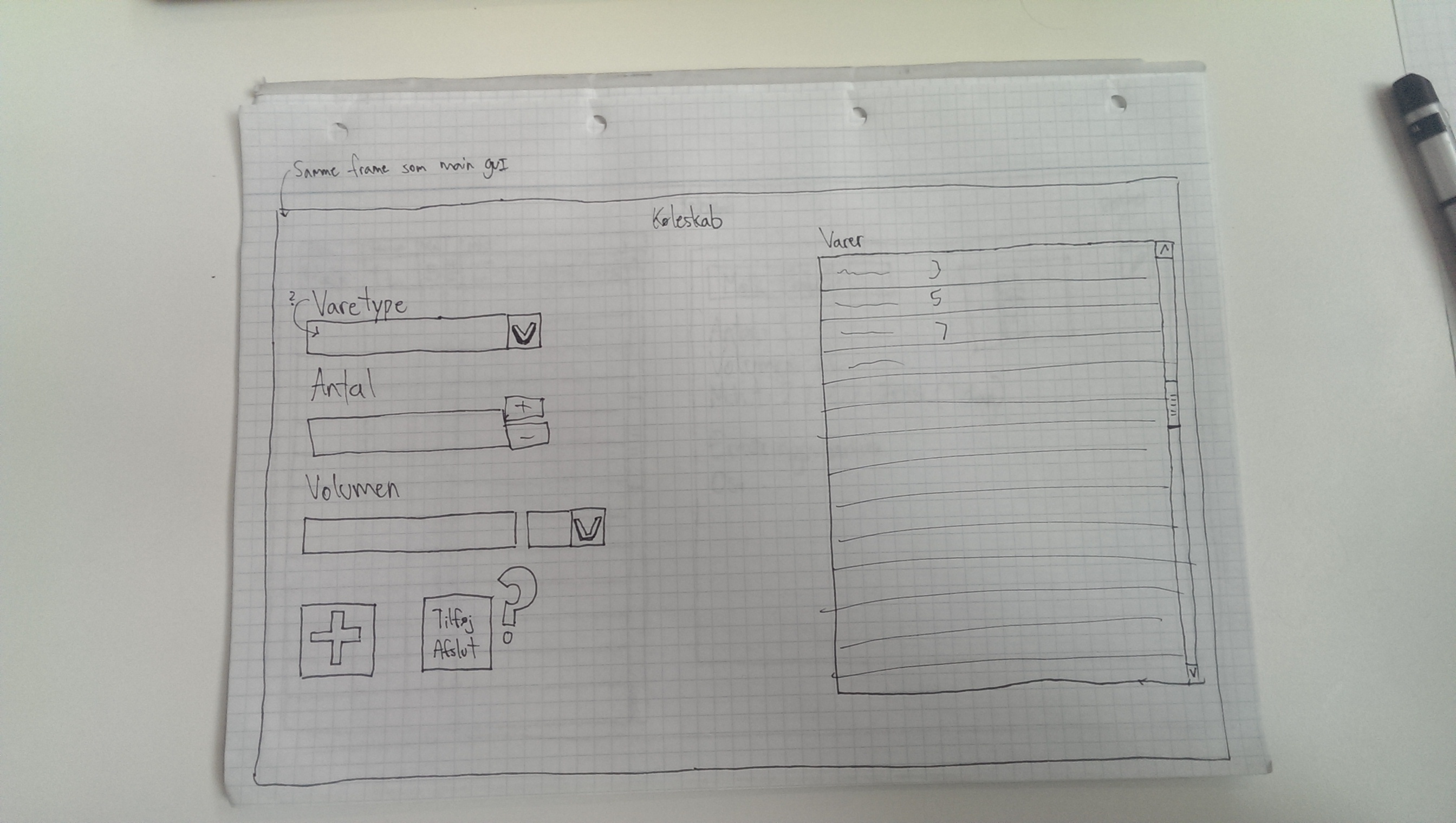
**Figur 3** Skitse af "Se varer"

Siden skitsen, er desuden tilføjet en *Slet*-knap til venstre for hver vare, hvorfra det er muligt at fjerne en vare fra listen. Det skal desuden være muligt inkrementere/dekrementere antallet af vare med én ved tryk på ’+/-’-knapper, som vil befinde sig under knappen *Rediger*.

*Ernæringsværdier* er et eksempel på hvor ekstra information fra eventuelle udvidelser vil befinde sig. Det vil altså ikke være en del af kernefunktionaliteten.

#### Tilføj vare

Designet af menuen for ”Tilføj varer” er blevet skitseret som i **Figur 4**.



**Figur 4** Skitse af "Tilføj varer"

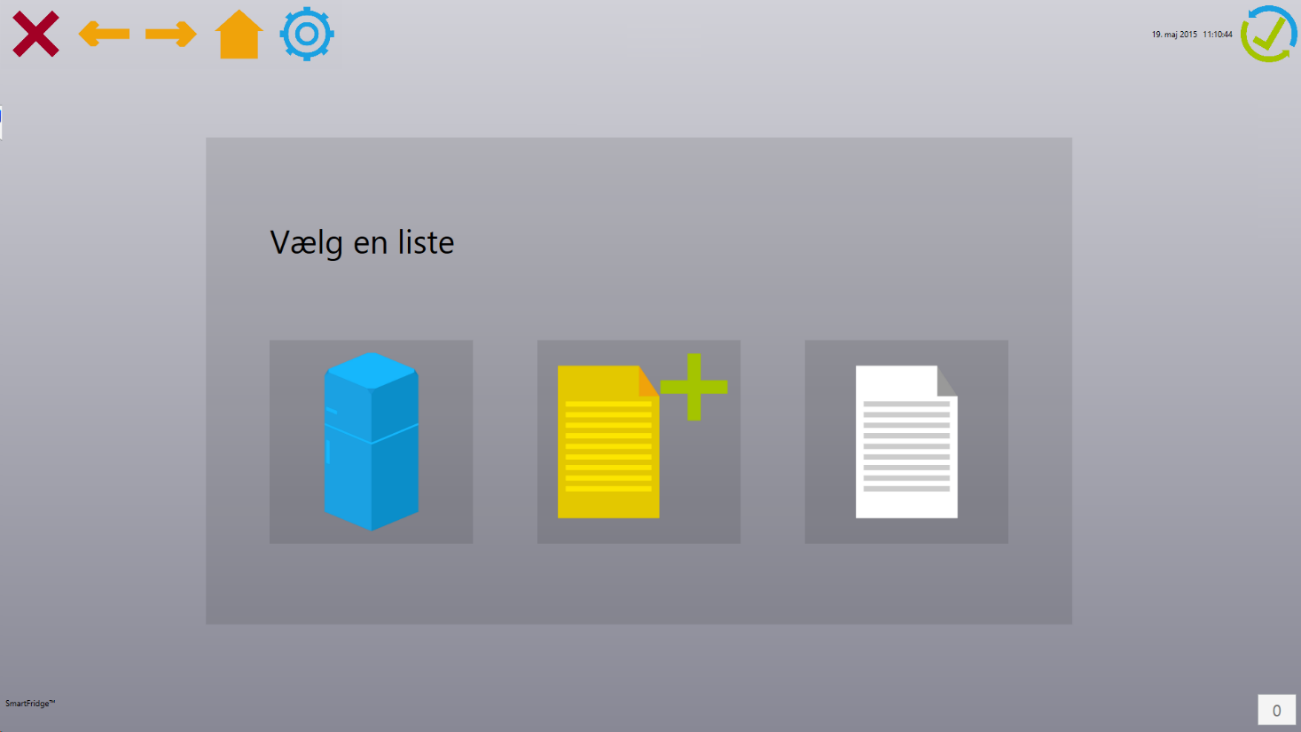
Med implementering af mulighed for at indtaste holdbarhedsdato, er dette felt siden skitsen blevet tilføjet under *Volumen*-feltet.

### Implementering

#### View Layer

##### Hovedvindue

Øverste lag i *View Layer* er hovedvinduet (*MainWindow*), som består af en ramme omkring en *UserControl*, som det fremgår af **Figur 5**.



**Figur 5** Hovedmenu

|  |
| --- |
| **Figur 6** Klassen MainWindow |

Rammen indeholder en mængde funktionalitet, som på et hvilket som helst givet tidspunkt i programmet vil være tilgængeligt.

Det røde kryds øverst til venstre repræsenterer muligheden for at afslutte programmet (*Close\_Button\_Clicked*).

De to orange pile tillader navigering frem (*Button\_Forward\_Clicked*) og tilbage (*Button\_Back\_Clicked*) blandt de tidligere besøgte sider. Selve funktionaliteten for disse funktioner er implementeret i *CtrlTemplate*-klassen, og vil blive uddybet i det følgende afsnit.

Det orange hus repræsenterer *Hjem*-knappen, som til enhver tid tillader brugeren at returnere til hovedmenuen; altså indlæser *UserControl*-klassen *ShowListSelection*.

Det blå tandhjul repræsenterer muligheden for at ændre på indstillinger. Da dette endnu ikke er implementeret, er den i første omgang blot en *placeholder*.

I øverste højre hjørne, ved siden af uret, som også styres internt i klassen (*timer\_Tick*), ses status på synkronisering (*syncStatus*). Når alle lokale data er synkroniseret med den eksterne database, vises et grønt flueben i knappen. Hvert 10. minut forsøges en synkronisering automatisk (*eventT*), og i mellemtiden kan en synkronisering initieres ved et tryk på knappen (*SyncButton\_OnClick*). I begge tilfælde skifter synkroniseringsknappens billede (*ChangeSyncImage*) til kun at indeholde to roterende pile. Hvis en synkronisering er fejlet, ændres billedet til at indeholde et rødt kryds.

Nederst til højre er en knap, som illustrerer antallet af notifikationer. Såfremt en vare er for gammel, vil der komme en notifikation om dette, samt et knap som tillader brugeren at fjerne notifikationen. Trykkes der et andet sted på skærmen, minimeres notifikationerne igen.

##### Control Template

|  |
| --- |
| **Figur 7** Klassen CtrlTemplate |

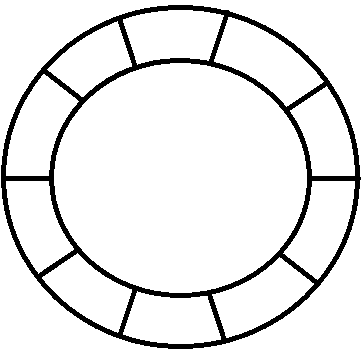
Indholdet i hovedvinduets *UserControl* styres af *UserControl-*klassen *CtrlTemplate* (**Figur 7**), som ved initialisering indlæser oversigten over tilgængelige lister. På samme tid oprettes en instans af objektet *BLL* (Business Logic Layer), som håndterer al ikke grafisk relateret *code behind*.

Constructoren for enhver af de implementerede *UserControls* tager imod en instans af klassen *CtrlTemplate*. Når et skifte ønskes, kaldes funktionen *ChangeGridContent* i *CtrlTemplate* med den ønskede *UserControl*, som indlæses i stedet for den nuværende, som set i **Kodestump 1**.

|  |
| --- |
| private UserControl \_uc;  public void ChangeGridContent(UserControl uc)  {  \_uc = uc;  CtrlTempGrid.Children.Clear();  CtrlTempGrid.Children.Add(\_uc);  [...]  } |

**Kodestump 1** Skift af UserControl i CtrlTemplate

*CtrlTemplate* lagrer desuden de foregående *UserControls*, som programmet har vist, i en cirkulær buffer (**Figur 8**). Dette muliggør brugen af *Frem*- og *Tilbage*-knapperne, som ses øverst til venstre på **Figur 5**.



[0]

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]

**Figur 8** Cirkulær buffer

Den cirkulære buffer initieres som i **Kodestump 2**. Bemærk at attributternes navne er forkortet her af pladshensyn. For den fulde implementering, henvises til **bilag XX**.

|  |
| --- |
| public UserControl[] NavHisCol { get; private set; }  private int NavHisColPos;  private int NavHisColOrgPos;  public CtrlTemplate()  {  [...]  \_uc = new CtrlShowListSelection(this);  [...]  NavHisCol = new UserControl[10];  NavHisCol[0] = \_uc;  NavHisColPos = 0;  NavHisColOrgPos = NavHisColPos;  } |

**Kodestump 2** Initiering af cirkulær buffer

*NavigationHistoryCollection* er selve arrayet, hvori bufferen lagres.

*NavigationHistoryCollectionPosition* holder styr på hvor den nuværende *UserControl* er placeret, mens *NavigationHistoryCollectionOriginalPosition* holder styr på den foregående *UserControl*. Disse benyttes at at sikre at vi kan finde den korrekte *UserControl*, og at der ikke bevæges til *UserControl*, som ikke længere er gyldig.

Efter dette, lagres hver ny *UserControl* på den næste plads i bufferen, så snart den indlæses, og med funktionerne *NavigateBack* og *NavigateForwards*, benyttes *Frem-* og *Tilbage*-knapperne til at navigere i bufferen.

##### Show List Selection

|  |
| --- |
| **Figur 9** Klassen CtrlShowListSelection |

Klassen *CtrlShowListSelection* (**Figur 9**) er den første *UserControl*, som indlæses i *CtrlTemplate*, når sidstnævnte initieres. Den indeholder en oversigt over de tilgængelige lister, og sørger for at oprette den rette *UserControl*, alt efter brugerens valg.

Constructoren for *CtrlShowListSelection* tager imod *CtrlTemplate*-klassen, som har kaldt den, således at den kan benytte *ChangeGridContent*-funktionen til at skifte *UserControl*, når brugeren har trykket på en af knapperne, som set i **Figur 10**.

|  |
| --- |
| private CtrlTemplate \_ctrlTemp;  private void BtnInFridge\_Click(object sender, RoutedEventArgs e)  {  \_ctrlTemp.ChangeGridContent(new CtrlItemList("Køleskab", \_ctrlTemp));  } |

**Figur 10** Skift af UserControl fra CtrlShowListSelection

Samme fremgangsmåde benyttes ved tryk på de andre lister.

Bemærk at listen identificeres på baggrund af en *string*. Denne umiddelbart hårde kodning er foretaget, med øje for muligheden for at udvide systemet til at lade brugeren oprette nye lister, som hver især kan identificeres på baggrund af deres navne. Alternativt kunne listerne oprettes som objekter, men da de alligevel ikke skulle indeholde andet end et navn, blev det valgt at en *string* var tilstrækkelig.

##### Item List

**Figur 11** viser den grafiske repræsentation af *CtrlItemList*, uden den omkringliggende ramme.



**Figur 11** Listen ”Køleskab”

|  |
| --- |
| **Figur 12** Klassen CtrlShowListSelection |

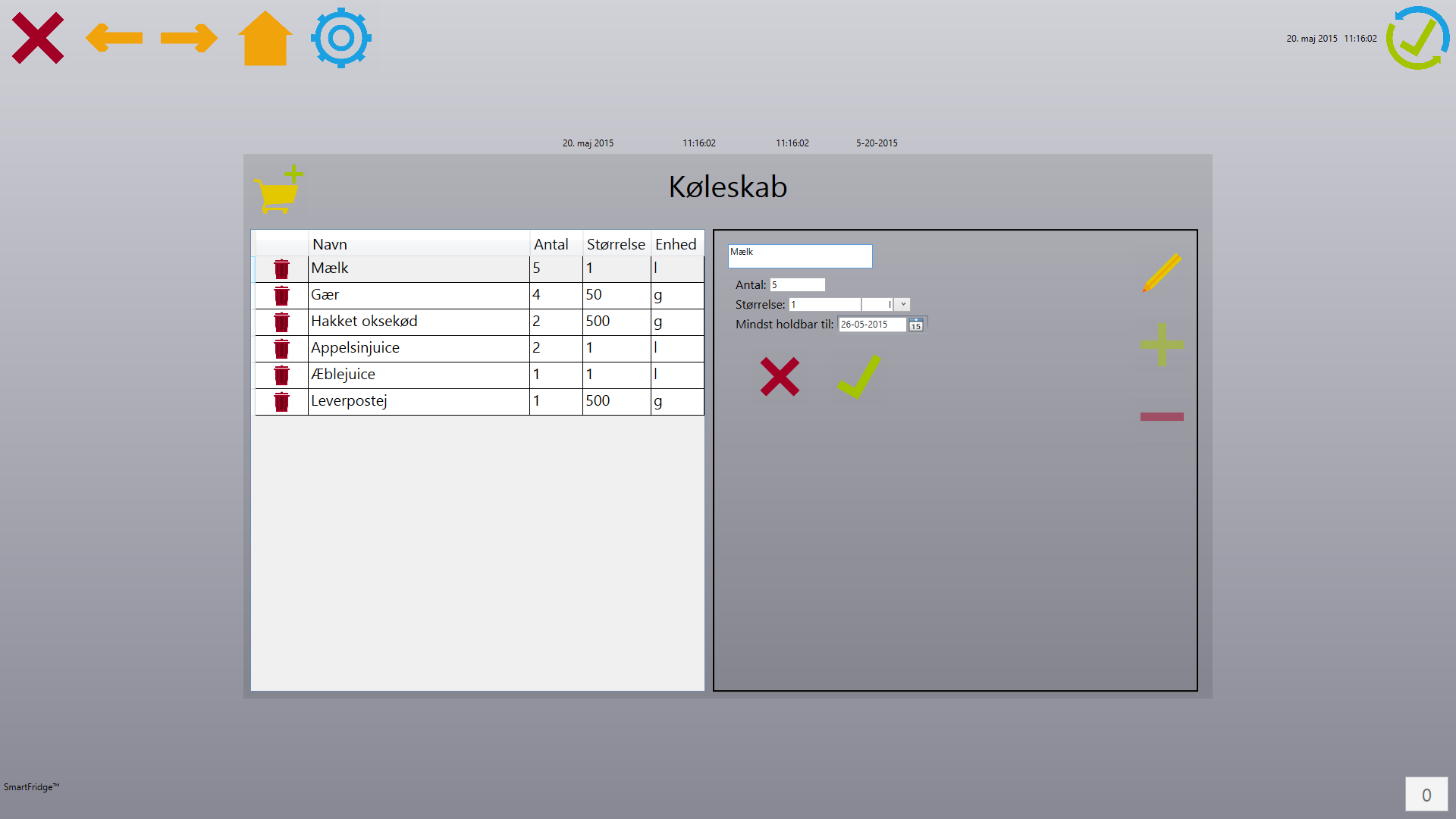
Denne klasse (**Figur 12**) indeholder en del mere funktionalitet end de foregående, og gør derfor i stor stil brug af *Business Logic Layer’*et (*BLL*), som håndterer den bagvedliggende logik, mens *CtrlItemList* selv håndterer den grafiske repræsentation.

*CtrlItemList* viser indholdet af den valgte liste, baseret på den medsendte *string*, som kædes sammen med listen af samme navn i databasen.

Når CtrlItemList oprettes, indlæses alle data (*LoadItemData*) fra den pågældende database i et *DataGrid*, og et klik på en række (*DataGridItems\_SelectionChanged*) udløser nærmere informationer om den enkelte vare, som fremkommer i informationsvinduet i højre side af vinduet.

Til venstre for hver vare, repræsenterer den røde skraldespand muligheden for at slette en vare (*BtnDelete\_Click*).

I informationsvinduet repræsenterer den gule blyant muligheden for at redigere den valgte vare (*BtnEdit\_Click*), hvilket resulterer i at vareinformationerne omdannes til redigerbare tekstblokke, som illustreret i **Figur 13**. Efter redigeringen, er det muligt at bekræfte (*BtnAccept*) ved at trykke på det grønne flueben, hvorved ændringerne persisteres, eller annullere (*BtnCancel\_Click*) ved at trykke på det røde kryds, hvorved ændringerne ignoreres.



**Figur 13** Rediger vare

Ved klik på det grønne plus er det muligt at øge (*BtnInc\_Click*) eller formindske (*BtnDec\_Click*) mængden af den valgte vare med én, uanset om der er trykket på *Rediger*. Disse funktioner syntes relevante at gøre let tilgængelige, for at gøre løbende forbrug, og registrering af ofte brugte varer, hurtigere at registrere.

Øverst til venstre i *CtrlItemList* ses en gul indkøbsvogn med et grønt plus. Denne knap tillader brugeren at tilføje nye varer (*BtnAddItem\_Click*), og fører til den sidste *UserControl*-klasse, *Additem*.

**Kodestump 3** viser hvordan der skiftes *UserControl* fra en liste til vinduet hvor varer tilføjes.

|  |
| --- |
| private void BtnAddItem\_Click(object sender, RoutedEventArgs e)  {  **\_ctrlTemp.ChangeGridContent(new AddItem(ListType, \_ctrlTemp));**  } |

**Kodestump 3** Skift af UserControl fra CtrlItemList til AddItem

*CtrlTemplate*-objektet, som blev sendt med og lagret i constructoren, benyttes til at kalde funktionen *ChangeGridContent*, hvori en ny instans af *AddItem* oprettes med listens egen type og det samme *CtrlTemplate*-objekt.

**Kodestump 4** viser hvordan *BLL* benyttes til at håndtere funktionaliteten.

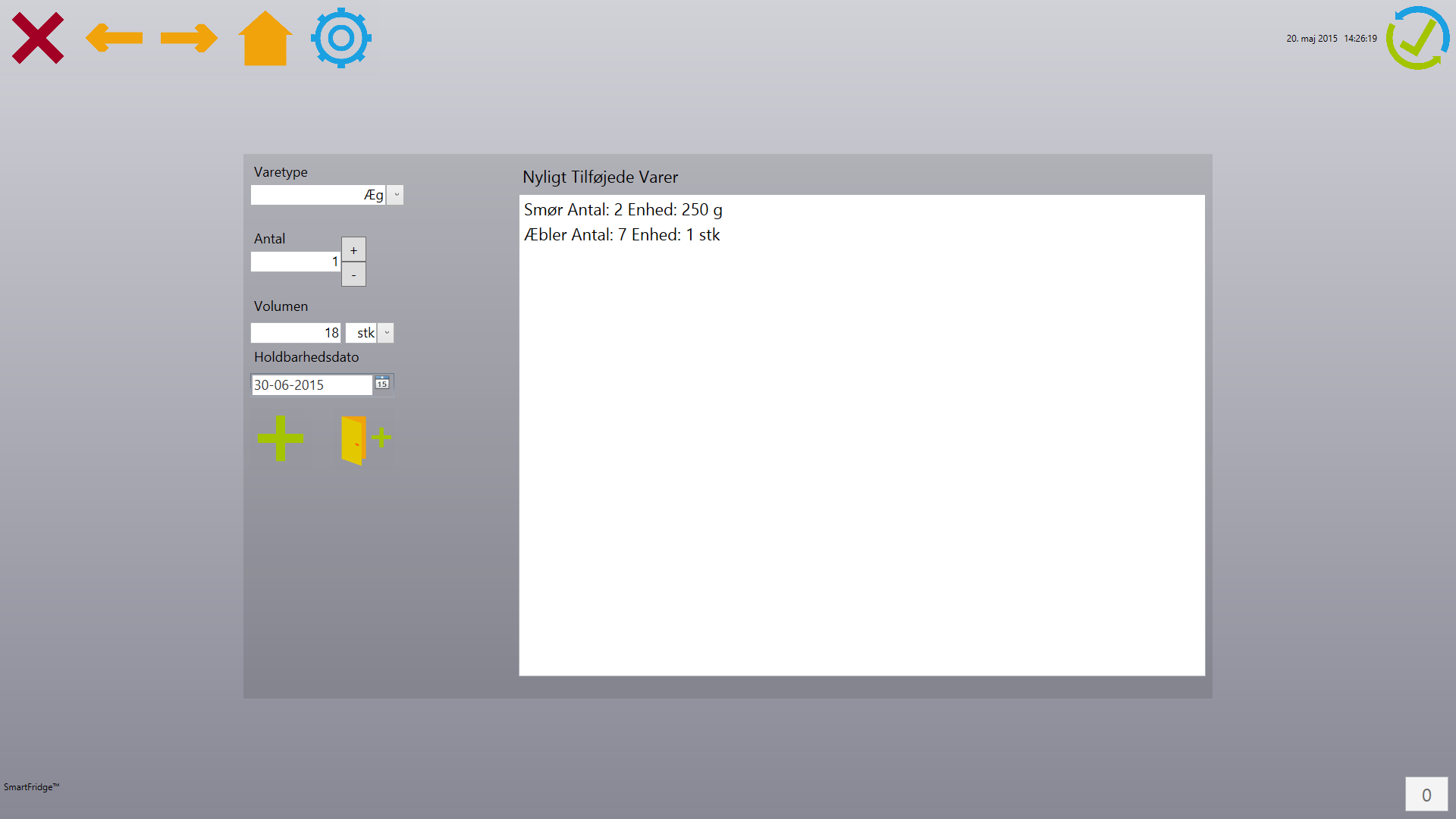
|  |
| --- |
| private async void BtnDelete\_Click(object sender, RoutedEventArgs e)  {  GUIItem itemDelete = DataGridItems.SelectedItem as GUIItem;  GUIItems.Remove(itemDelete);  **\_ctrlTemp.\_bll.DeleteItem(itemDelete);**  DataGridItems.UnselectAllCells();  DataGridItems.SelectedIndex = 0;  } |

**Kodestump 4** Sletning af vare ved hjælp af Business Logic Layer

Mens funktionen selv håndterer den grafiske fjernelse af det pågældende objekt, benyttes *BLL-objektet* i *CtrlTemplate*-objektet til at slette varen fra databasen. *,*

##### Add Item

**Figur 15** viser den grafiske repræsentation af *AddItem*, uden den omkringliggende ramme.



**Figur 14** Tilføj vare(r)

|  |
| --- |
| **Figur 15** Klassen AddItem |

Som sine forgænger, *CtrlItemList*, kontrollerer denne klasse (**Figur 15**) meget funktionalitet, som opnås gennem funktionskald til klassen *Business Logic Layer* (*BLL*), mens den grafiske repræsentation håndteres af klassen selv. Brugen af *BLL* foregår, som i *CtrlItemList*, gennem det medsendte *CtrlTemplate*-objekt.

*AddItem* tilbyder muligheden for at tilføje én eller flere varer til listen, hvorfra klassen blev kaldt.

Varens navn skrives ud for *Varetype*, antallet skrives ud for *Antal*, mængden og enheden skrives ud for *Volumen*, og holdbarhedsdatoen skrives under *Holdbarhedsdato*.

Det grønne plus repræsenterer muligheden for at tilføje varen (*AddButton\_Click*) til den midlertidige liste (*newItems*) i højre side af vinduet. Ved klik på døren med det grønne plus, tilføjes og persisteres varen (*AddExitButton\_Click*), sammen med varerne i den midlertidige liste, i databasen.

*Varetype* skrives i en *ComboBox*, hvor der ved klik kan ses forslag til eksisterende varer, som kan vælges hvis ønskes.

*Enhed* ud for *Antal* er også vist som en *ComboBox*, hvor det er muligt at vælge eksisterende enheder, eller en ny kan skrives.

Ud for *Antal* ses to ’+/-’-knapper, som kan benyttes til at øge (*PlusButton\_Click*) eller formindske (*MinusButton\_Click*) mængden af den pågældende vare med én.

#### Business Logic Layer

##### Constructor

I constructoren, der kan ses på **Kodestump 5** oprettes aller først den AdoNetContext som skal bruges til at oprette forbindelse til den lokale database med. Herefter oprettes de tre repositories, som skal bruges til at tilgå databasens forskelllige tabeller.

|  |
| --- |
| public BLL()  {  Context = new AdoNetContext(\_connectionFactory);  \_itemRepository = new ItemRepository(Context);  \_listRepository = new ListRepository(Context);  \_listItemRepository = new ListItemRepository(Context);  } |

**Kodestump 5** Business Logic Layer constructoren

##### WatchItems

WatchItems er en property, der bruges til at hente en liste med alle de relevante GUIItems for en given liste til GUIen. Det første den gør er at indlæse hele databasens indhold med LoadFromDB(); funktionen, der er beskrevet senere i dette afsnit. Herefter finder den ud af om listen den skal fylde i eksisterede på i databasen. Hvis ikke listen eksisterede i forvejen oprettes den, dette kan ses på **Kodestump 6**.

|  |
| --- |
| List<string> temp = new List<string>();  foreach (var list in Lists)  {  temp.Add(list.Name);  }  if (!temp.Contains(CurrentList))  {  CreateList();  } |

**Kodestump 6** Lists er en property,   
 der indeholder listerne.

Efter det findes alle de ListItems i databasen, der skal figurerer på den nuværende liste. For hvert ListItem der bliver fundet oprettes et tilsvarende GUIItem, hvis type er givet af den Item, det pågældende ListItem er koblet sammen med. Hele processen kan ses i **Kodestump 7**. Alle disse GUIItems ligges i en liste, som til sidst bliver returneret.

|  |
| --- |
| foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.List.ListName == CurrentList)  {  foreach (var dbItem in \_dbItems)  {  if (dbListItem.Item.ItemId == dbItem.ItemId)  {  GUIItem guiItem = new GUIItem();  guiItem.Type = dbItem.ItemName;  guiItem.Amount = (uint)dbListItem.Amount;  guiItem.Unit = dbListItem.Unit;  guiItem.Size = (uint)dbListItem.Volume;  guiItem.ShelfLife = dbListItem.ShelfLife;  guiItems.Add(guiItem);  }  }  }  } |

**Kodestump 7** Her ses, hvordan alle ListItems, der skal figurerer på   
 den nuværende liste, kobles sammen med deres respektive item i   
 et GUIItem.

##### LoadFromDB

LoadFromDB indlæser alt fra databasen, og kommet indholdet i lokale objekter. Som det kan ses på **Kodestump 8**, gøres dette ved at kalde funktionen GetAll(); på samtlige repositories. Da denne funktion returnere en IEnumerable kaldes ToList(); for at lave dem til lister. Når alt ligger lokalt kaldes Mapper(); funktionen, der sørger for at de lokale referencer er som de skal være. Listerne fra databasen bruges til at oprette lokale GUIItemList objekter, der til forskel fra en List fra databasen har en indbygget liste af GUIItems.

|  |
| --- |
| List<List> lists = new List<List>();  using (var uow = Context.CreateUnitOfWork())  {  lists = \_listRepository.GetAll().ToList();  \_dbItems = \_itemRepository.GetAll().ToList();  \_dblistItems = \_listItemRepository.GetAll().ToList();  \_listItemRepository.Mapper(\_dbItems, lists, \_dblistItems);  } |

**Kodestump 8 Her indlæses databasens inhold.**

Når alt ligger lokalt og alle GUIItemListerne er oprettet sammenkobles alle ListItems med deres respektive Item, i et GUIItem. Alle disse GUIItems ligges i deres tilhørende lokale GUIItemList, som det kan ses på **Kodestump 9**

|  |
| --- |
| foreach (var list in Lists)  {  foreach (var listItem in \_dblistItems)  {  if (listItem.List.ListId == list.ID)  foreach (var dbItem in \_dbItems)  {  if (listItem.Item.ItemId == dbItem.ItemId)  {  GUIItem guiItem = new GUIItem()  {  Amount = (uint)listItem.Amount,  Type = dbItem.ItemName,  Size = (uint)listItem.Volume,  Unit = listItem.Unit,  ID = listItem.ItemId,  ShelfLife = listItem.ShelfLife  };  list.ItemList.Add(guiItem);  break;  }  }  }  } |

**Kodestump 9 Her ligge alle ListItems og Items sammen til GUIItems,  
 der fordeles ud på deres respektive lokale lister.**

##### CheckShelfLife

CheckShelfLife funktionen går alle items på en specifik liste igennem for at finde ud af, hvorvidt nogle af disse har overskredet deres sidste holdbarhedsdato. Den gør dette ved at sammenligne hvert GUIItems ShelfLife property, med dagens dato. For hvert GUIItem, der er blevet for gammel, tilføjes en notifikation til en liste, der til sidst returneres som, det kan ses på **Kodestump 10**.

|  |
| --- |
| var list = new List<Notification>();  foreach (var item in items.ItemList)  {  if (item.ShelfLife != null)  {  if (item.ShelfLife.Date <= DateTime.Now)  {  string message = item.Type + " blev for gammel d. " + DateTime.Now.Date;  list.Add(new Notification(message, DateTime.Now, item.ID));  }  }  }  return list; |

**Kodestump 10** Her kan ses hvordan en notifikation med information omkring hvilken vare,  
 der blevet for gammel, tilføjes for hver vare hvis holdbarhedsdato er overskreden.

##### AddItemsToTable

AddItemsToTable undersøger først, hvorvidt den liste funktionen blev fra eksistere, hvis den ikke før det bruges CreateList. Når listen eksistere findes den rigtige lokale liste dens id skal bruges når der skal tilføjes ListItems til databasen. Herefter løbes alle de GUIItems, der skal tilføjes igennem for at finde ud af om der er nogle af dem, som er af ikke i forvejen eksisterende type. Dette kan ses på **Kodestump 11**, her bruges IsNewItem til at finde ud af, hvorvidt et GUIItem er af en ny type eller ej. Bliver der tilføjet nye Items, kaldes LoadFromDB så de nye Items også ligger lokalt.

|  |
| --- |
| bool newItemAdded = false;  using (var uow = Context.CreateUnitOfWork())  {  foreach (var newGuiItem in newGuiItems)  {  if (IsNewItem(newGuiItem))  {  Item dbItem = new Item()  {  ItemName = newGuiItem.Type,  StdUnit = newGuiItem.Unit,  StdVolume = (int)newGuiItem.Size  };  \_itemRepository.Insert(dbItem);  newItemAdded = true;  }  }  uow.SaveChanges();  }  if (newItemAdded)  LoadFromDB(); |

**Kodestump 11 Her tilføjes alle nye typer af Items,   
 hvorefter der loades fra databasen igen.**

Når de nyeste opdateringer er hentet ned, tjekkes der på hver af de nye GUIItems om den kaldende liste allerede har et GUIItem af samme type. Findes der ikke et match, laves der et ListItem ud fra det nye GUIItems attributter, og det tilføjes til databasen. Eksistere der allerede et GUIItem af samme type sammenlignes der på deres størrelse/volume, enhed og udløbsdato, er der blot en af disse, som ikke stemmer overens med det i forvejen eksisterende tilføjes der et nyt ListItem. Stemmer alle parametrene overens laves der et nyt ListItem, hvor antallet er summen af det gamle og det nye item. Herefter slettes det gamle ListItem, og det nye indsættes i databasen, hvilket kan ses på **Kodestump 12**.

|  |
| --- |
| [...]  if (dbItem.ItemName == newGuiItem.Type)  {  foreach (var dbListItem in \_dblistItems)  {  if (dbItem.ItemId == dbListItem.Item.ItemId &&  dbListItem.List.ListId == currentGuiItemList.ID &&  dbListItem.Unit == newGuiItem.Unit &&  dbListItem.Volume == newGuiItem.Size)  {  int currentAmount = dbListItem.Amount;  ListItem updatedListItem = new ListItem(((int)newGuiItem.Amount +  currentAmount),  (int)newGuiItem.Size,  newGuiItem.Unit,  dbListItem.List,  dbItem,  newGuiItem.ShelfLife);  \_listItemRepository.Delete(dbListItem);  \_listItemRepository.Insert(updatedListItem);  }  [...] |

**Kodestump 12 Her kan ses hvordan det håndteres når der bliver tilføjet et allerede eksisterende  
 GUIItem**

Når alle nye GUIItems er tilføjet til databasen på den ene eller den anden måde kaldes SaveChanges(); så ændringerne gemmes i databasen, og LoadFromDB(); kaldes for at få alle disse ændringer gemt lokalt også. Når det nyeste ligger lokalt kaldes Mapper(); der sikrer at alle referencer er som de skal være. Til sidst kaldes STDToShopListControl, som er beskrevet længere nede i dette afsnit.

##### DeleteItem

DeleteItem finder først den rigtige lokale liste, da listens id skal bruges til at finde den ListItem, der skal slettes fra databasen. Bagefter som vist på **Kodestump 13** findes og slettes det fundne ListItem. Når det er gjort laves et kald til uow.SaveChanges, så ændringen også sker i databasen.

|  |
| --- |
| using (var uow = Context.CreateUnitOfWork())  {  foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.Item.ItemName == GUIitemToDelete.Type  && dbListItem.Amount == GUIitemToDelete.Amount  && dbListItem.Unit == GUIitemToDelete.Unit  && (uint)dbListItem.Volume == GUIitemToDelete.Size  && dbListItem.ListId == currentGuiItemList.ID)  {  \_listItemRepository.Delete(dbListItem);  break;  }  }  uow.SaveChanges();  } |

**Kodestump 13 Her kan ses hvordan et ListItem slettes fra databasen.**

##### ChangeItem

ChangeItem starte ligesom DeleteItem med først at finde den rigtige lokale liste da dennes id skal bruges til at finde det rigtige ListItem. Herefter tjekker den på om det kun er det fundne ListItem, der skal opdateres eller om den tilhørende Item også skal opdateres. Hvis Itemet skal opdateres gøres det som det kan ses på **Kodestump 14**.

|  |
| --- |
| [...]  if (oldItem.Type != newItem.Type)  {  foreach (var dbItem in \_dbItems)  {  if (dbItem.ItemName == oldItem.Type)  {  dbItem.ItemName = newItem.Type;  \_itemRepository.Update(dbItem);  }  }  }  [...] |

**Kodestump 14 Her opdateres et ListItems   
 tilhørende item.**

Når det rigtige ListItem er fundet, bliver dette opdateret ved at det allerede eksisterende ListItem slettes, og der indsættes et nyt med de opdaterede værdier som det ses på **Kodestump 15**.

|  |
| --- |
| [...]  ListItem updatedListItem = new ListItem((int)newItem.Amount,  (int)newItem.Size,  newItem.Unit,  dbListItem.List,  dbListItem.Item,  newItem.ShelfLife);  \_listItemRepository.Delete(dbListItem);  \_listItemRepository.Insert(updatedListItem);  uow.SaveChanges();  break;  [...] |

**Kodestump 15 Her "opdateres" et ListItem ved først at slette det gamle  
 ListItem, og derefter indsætte et nyt med opdaterede værdier.**

##### STDToShopListControl

STDToShopListControl tjekker først på om den er blevet kaldt fra "Standard-beholdning" listen, da det kun er ved kalde derfra den skal gøre noget. Derfor returnere den bare, hvis kaldet kom fra en af de to andre liste. På **Kodestump 16** kan det ses, hvordan den opretter to lister af ListItems med varer der henholdsvis er i køleskabet, eller som Bruger har valgt altid skal være i køleskabet.

|  |
| --- |
| foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.List.ListName == "Køleskab")  {  har.Add(dbListItem);  }  else if (dbListItem.List.ListName == "Standard-beholdning")  {  skalAltidHave.Add(dbListItem);  }  } |

**Kodestump 16 Her kan ses, hvorledes der oprettes en liste over ting, der er  
 i køleskabet, og ting der skal være i køleskabet.**

Listen over ting, der altid skal være i køleskabet kopieres over i en ny liste ved navn "mangler", og der kigges nu efter overenstemmelser mellem de to tidligere lister. Findes der et ListItem i køleskabet, hvis amount er større end eller lig med det fra standard-beholdningen, fjernes det fra mangler listen, ellers, hvis antallet er under bliver differensen mellem disse, antallet på "mangler" listen. På **Kodestump 17** kan ses, hvordan dette er implementeret. Her efter løbes "mangler" listen igennem og samtlige ListItems kobles sammen deres tilhørende Item i et GUIItem, som tilføjes til en ny liste. Denne liste tilføjes til "Inkøbsliste" med AddItemsToTable funktionen.

|  |
| --- |
| List<ListItem> mangler = new List<ListItem>(skalAltidHave);  foreach (var STDListItem in skalAltidHave)  {  foreach (var ownedListItem in har)  {  if (ownedListItem.Item.ItemName == STDListItem.Item.ItemName &&  ownedListItem.Unit == STDListItem.Unit &&  ownedListItem.Volume == STDListItem.Volume)  {  if (ownedListItem.Amount >= STDListItem.Amount)  {  mangler.Remove(STDListItem);  }  if (ownedListItem.Amount <= STDListItem.Amount)  {  STDListItem.Amount -= ownedListItem.Amount;  }  }  }  } |

**Kodestump 17 Her fjernes alle vare som allerede er i køleskabet i et korrekt antal fra  
 listen over varer, der mangler. Varer, hvor antallet er forkert, bliver stående med det  
 manglende antal.**

### Test

Som følge af det kodemæssige design, som i første omgang blev oprettet med ren *code behind*, og senere, med inkorporering af *Data Access Layer* (*DAL*)-laget, blev opdelt i *View* og *Business Logic Layer* (*BLL*), er der opnået en ringe grad af testbarhed i *Fridge app*. Fokus har ligget på test af den bagvedliggende funktionalitet, frem for at bruge for meget tid på at omdesigne et, allerede på daværende tidspunkt, velfungerende system.

Da *Fridge app* er kodet uden *interfaces*, begrænser det ligeledes muligheden for modultests, da den høje kobling har medført at de fleste funktioner har måttet kastes direkte ud i integrationstests, idet de benytter sig af underliggende lag (*View* -> *BLL* -> *DAL*). Herudover oprettes der allerede i *BLL*’s constructor forbindelse til *DAL*, hvilket betyder at ingen tests vil kunne isoleres som modultests.

For mere om systemets tests, henvises derfor til afsnittet om **integrationstests, side XX**.

## Web app

### Design

### Implementering

### Test

## Database

### Design

### Implementering

### Test