

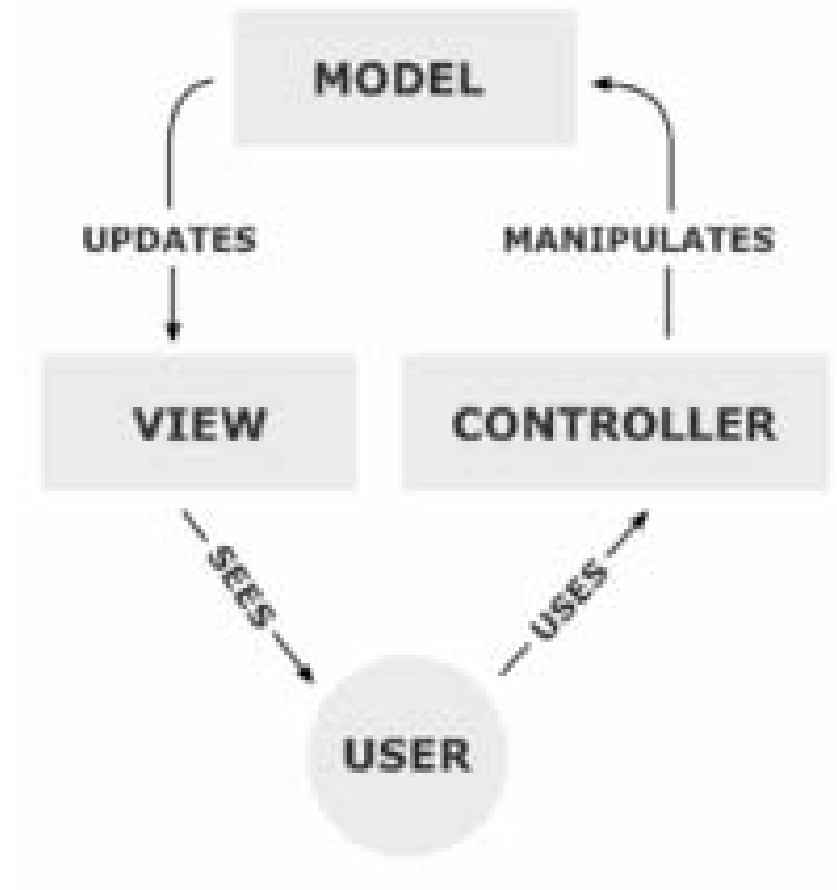
MVC

.. Og SOLID designprincipperne

MVC

- Opdel kode i brugergrænseflade (View), funktionslag (Controller) og Modellag (Model)
- Ansvarsfordelinger:
- View
 - Tager sig udelukkende af rendering af brugergrænseflade og interaktion med brugeren.
 - Layout, håndtering af bruger-input (menu-valg, button klik, liste-valg, etc), view navigation, feedback til brugeren.
- Controller
 - Implementerer systemets funktioner
 - Kender ikke til en konkret brugergrænseflade
- Model
 - Modellerer problemområdet

Model View Controller (MVC)



Brugsmønstre (Use Cases)

- Et systems funktionalitet udgøres af mængden af brugsmønstre
- “A use case captures the visible sequence of events that a system goes through in response to a single user stimulus.” *
- Eksempel: Hæv penge i Dankort automat
 1. Indsæt kort
 2. Indtast kode
 1. Hvis ugyldig – forsøg igen
 3. Indtast beløb
 4. Kvittering?
 5. Udbetal beløb
 6. Udskriv kvittering, hvis ønsket

Controllers

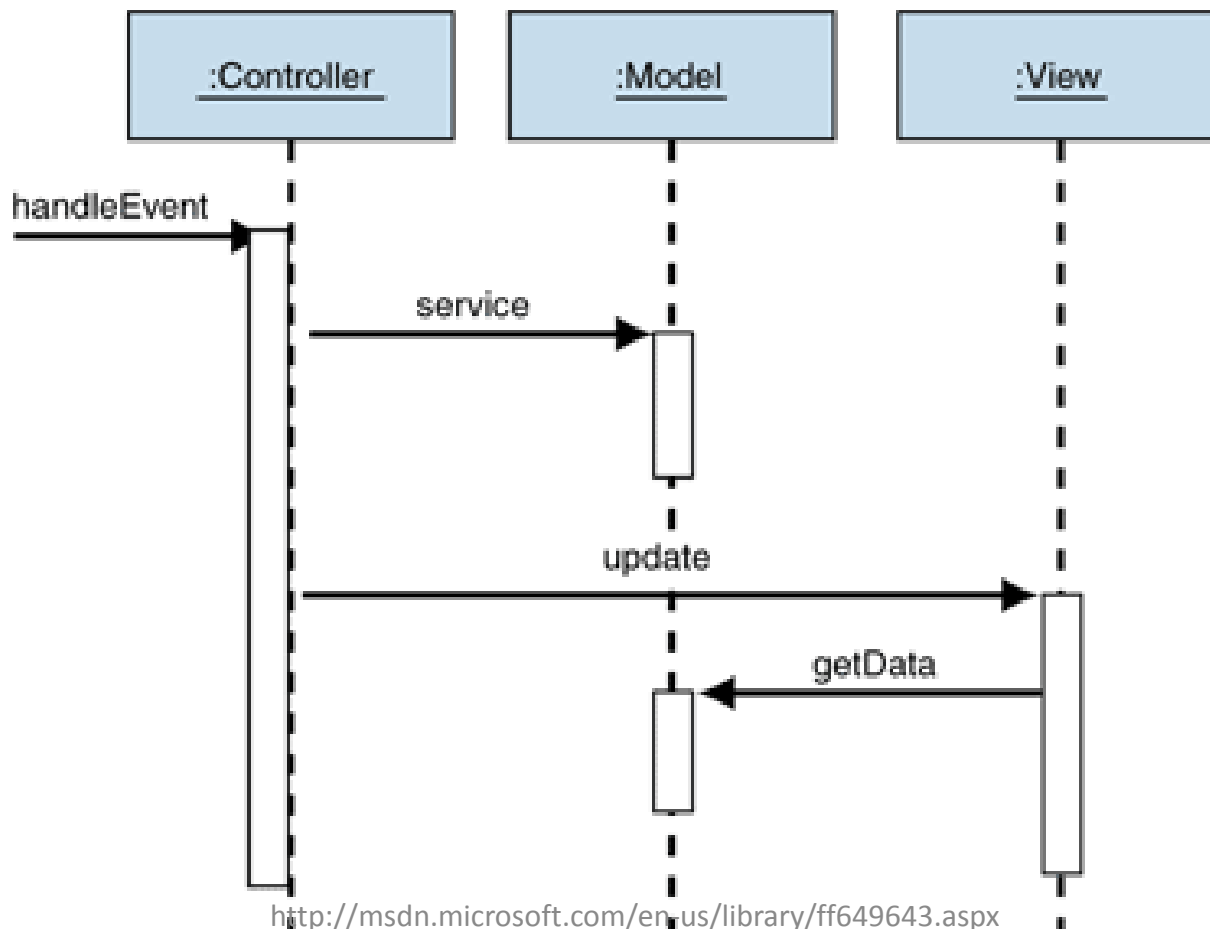
- Controllers implementerer systemets brugsmønstre.
- En controller klasse indkapsler logisk sammenhørende funktioner.
- Udgangspunkt: Lav én controller per brugsmønster - controlleren implementerer brugsmønstrets funktioner
- Dette tillader at controlleren kan holde styr på logisk tilstand (hvilke metoder er enabled)
 - F.eks.: #1. Indsæt kort, #2. Indtast Kode, #3. Indtast beløb, etc.
- En grænseflade (view) for det pågældende brugsmønster refererer til controlleren der udfører funktionaliteten.
- Ofte 1:1 forhold mellem View og Controller, men kan variere, f.eks. Aggregeret grænseflade (1 view – flere controllers) eller wizard (flere views – én controller)
- Controlleren kan notificere om dens logiske tilstand hvilket grænsefladen kan bruge til at lave tilhørende View tilstand.

Kode opdeling

- Lav Model som klassebibliotek
- Lav Controller som klassebibliotek
 - Controller importerer Model biblioteket
- Lav grænseflade
 - Importerer både Controller og Model bibliotek.
- Fordele:
- Promoverer lav kobling – høj samhørighed.
- Gør det ekstremt nemt at udskifte brugergrænseflade.
- Understøtter forskellige projekttyper (Win Forms, Console Application, GTK#, ...)
- Gør det nemt at teste
 - Controller og Model: Unit Tests
 - UI: Manuel aftestning eller [Coded UI Tests](#)

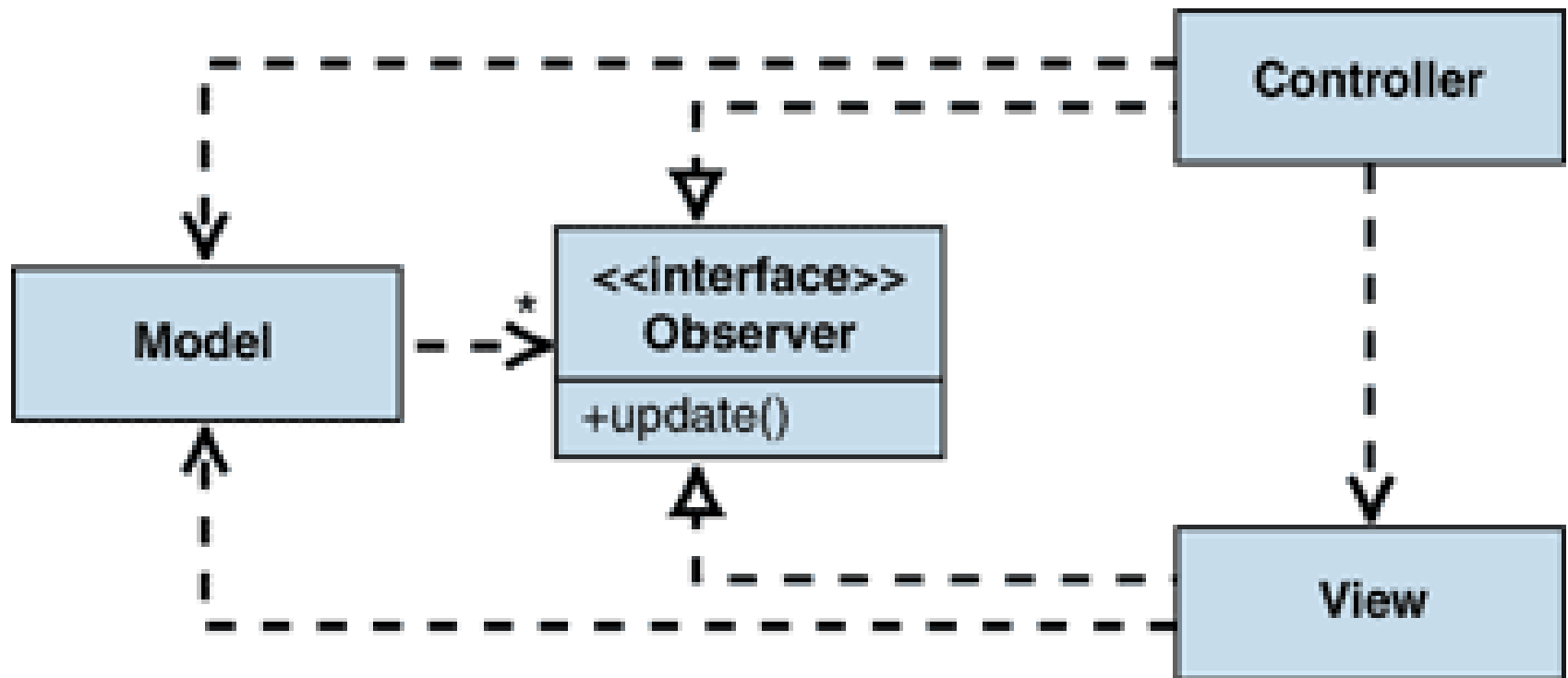
Passiv model

- Model er passiv: Signalerer ikke tilstandsændringer



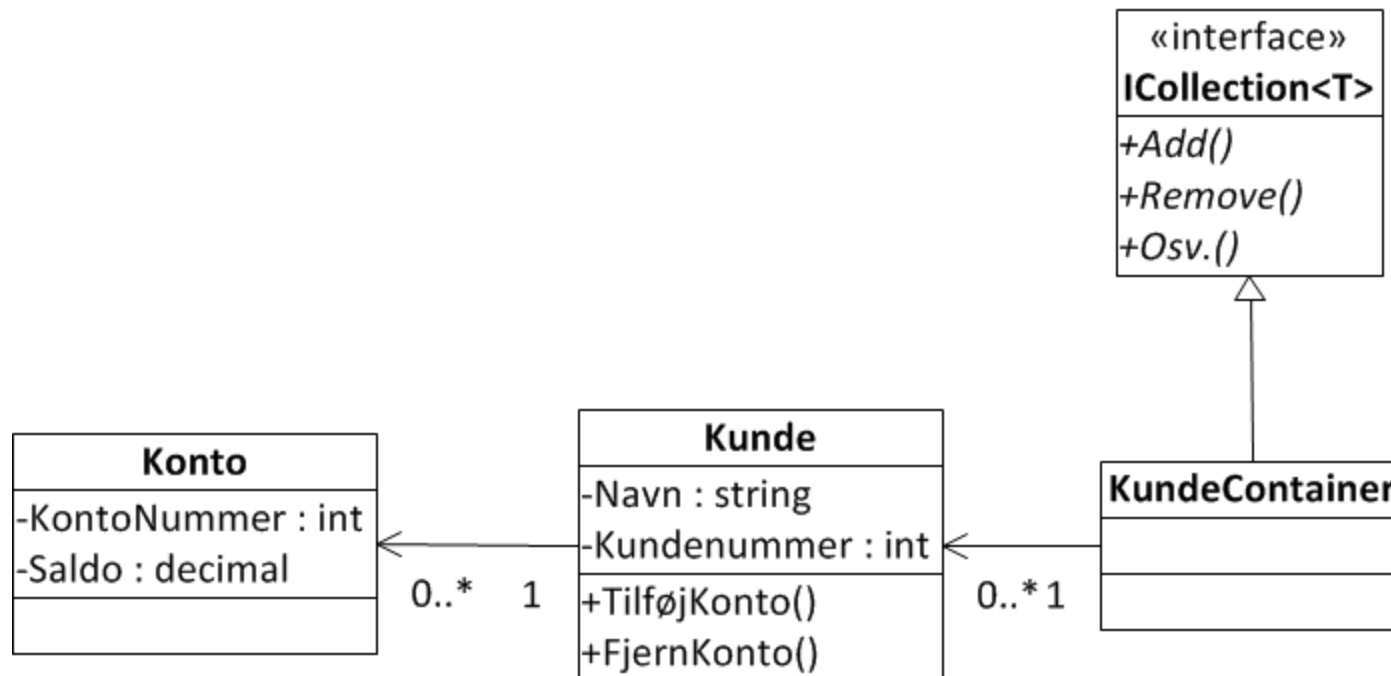
Aktiv model

- Model kan signalere tilstandsændringer



MVC By Example

- Et (simplificeret) system der administrerer kunder og deres konti.
- Her er modellen:



Brugsmønstre og funktioner

- Brugsmønstre:
 - Administrér kunder

Funktion	Kompleksitet	Type
Opret Kunde	Simpel	Opdatering/ Signalering
Slet Kunde	Simpel	Opdatering/ Signalering
Vis alle kunder	Simpel	Aflæsning
Vis rige kunder (>5k)	"Middel"	Aflæsning (/ "Beregning")

- Administrér konti for kunde

Funktion	Kompleksitet	Type
Opret Konto	Simpel	Opdatering/ Signalering
Slet Konto	Simpel	Opdatering/Signalering
Vis alle konti	Simpel	Aflæsning

Controllers

- To Controllers – én per brugsmønster:
 - **KundeAdministrationController**: Implementerer funktionerne i "Administrér Kunder" brugsmønstret.
 - **KundeKontiAdministrationController**: Implementerer funktionerne i "Administrér konti for kunde" brugsmønstret.
- (De er temmelig ens, så vi nøjes med eksemplet for KundeAdministrationController)
- Eksempler:
 - MVC med en passiv model (to varianter)
 - MVC med en aktiv model

MVC med en passiv model #1

- Controller holder en reference til et view interface.
- View interface definerer opdateringsmetoder for konkrete views ifm controller tilstande.

```
interface IView {
    void KundeOprettet(); //+ flere relevante UI metoder
}

class KundeController_Passiv1 {
    //Model data:
    private KundeContainer _Kundeliste = new KundeContainer();
    public Kunde AktuelKunde { get; set; }

    //View:
    //Setter injection, så vi kan skifte view undervejs (nyttigt v. Wizards)
    public IView View { get; set; }
    public KundeController_Passiv1(IView view) { this.View = view; }

    public void OpretKunde(Kunde k) {
        //Validering der kræver kendskab til problemdomæne-logik:
        if (_Kundeliste.ContainsKundeWithKundenummer(k.KundeNummer)) {
            throw new ArgumentException("Kunden findes allerede");
        }
        //Opdater model
        _Kundeliste.Add(k);
        //Opdater view
        View.KundeOprettet();
    }
}
```

```

class View_Passiv1 : IView
{
    private KundeController_Passiv1 _controller;

    public View_Passiv1() {
        _controller = new KundeController_Passiv1(this);
    }

    public void OpretKunde() {
        Console.WriteLine("Indtast kunde navn: ");
        string navn = Console.ReadLine();
        Console.WriteLine("Indtast kundenummer:");
        string nummer = Console.ReadLine();
        //Lav evt. simpel validering inden controller kaldes:
        try { _controller.OpretKunde(new Kunde(navn, nummer)); }
        catch (ArgumentException ex) { Console.WriteLine(ex.Message); }
    }

    public void KundeOprettet() {
        Kunde k = _controller.AktuelKunde;
        if (k != null)
            Console.WriteLine("Sidste kunde er '{0}'", k.Navn);

        this.OpretKunde();
    }
}

```

MVC med en passiv model #2

- Ulempe ved passiv model #1: Controller-forfatter skal forholde sig til View tilstande
- Controller-klienter skal implementere IView – også selvom klient ikke har nogen brugergrænseflade.
- En controller er blot et API der udstiller et systems funktionalitet. Det burde derfor ikke være nødvendigt at stille krav til klienter.
- Passiv Model #2: Controller holder ikke reference til IView.
- Controller kan i stedet angive logisk tilstand (om nødvendigt)
- Ulempe ved #2: Der placeres mere ansvar for tjek af tilstandsskift på klient. (dette er ikke nødvendigt ved en aktiv model)

Controller der ikke kender views

```
class KundeController_Passiv2
{
    //Logisk tilstand i stedet for reference til view
    public enum LogiskTilstand { KundeIkkeOprettet, KundeOprettet }
    public LogiskTilstand Tilstand { get; private set; }

    public void OpretKunde(Kunde k)
    {
        if (_Kundeliste.ContainsKundeWithKundenummer(k.KundeNummer))
        {
            throw new ArgumentException("Kunden findes allerede");
        }
        //Opdater model
        _Kundeliste.Add(k);

        //Opdater logisk tilstand (i stedet for at kalde metode på IView)
        Tilstand = LogiskTilstand.KundeOprettet;
    }
}
```



```
class View_Passiv2 {
    public void OpretKunde()
    {
        //Indlæs navn og nummer
        _controller.OpretKunde(new Kunde(navn, nummer));
        //View er nu ansvarlig for tjek på logisk tilstand
        if (_controller.Tilstand ==
            KundeController_Passiv2.LogiskTilstand.KundeOprettet)
            this.KundeOprettet();
        else
            this.KundeIkkeOprettet();
    }

    public void KundeIkkeOprettet() {
        Console.WriteLine("Prøv igen");
        this.OpretKunde();
    }

    public void KundeOprettet() {
        Kunde k = _controller.AktuelKunde;
        if (k != null)
            Console.WriteLine("Sidste kunde er '{0}'", k.Navn);

        this.OpretKunde();
    }
}
```

MVC med en aktiv model

Signalering af tilstandsændringer i model

```
public class Kunde : INotifyPropertyChanged { //Ditto for Konto klassen
    private string _Navn;

    public string Navn
    {
        set {
            if (value != _Navn) {
                this._Navn = value;
                OnNotifyPropertyChanged(new PropertyChangedEventArgs("Navn"));
            }
        }
    }

    //Ditto for tilstandsændringer i TilføjKonto() og FjernKonto()

    //View tilføjer eventhandler for at blive notificeret om ændringer
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnNotifyPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null) //controller og/eller view notificeres
            PropertyChanged(this, e);
    }
}
```

Navn på ændrede property

Tilstandsændring på collection klasse

```
public class KundeContainer : ICollection<Kunde>, INotifyCollectionChanged
{
    private Dictionary<string, Kunde> _Kunder;

    public void Add(Kunde kunde)
    {
        //noget validering

        _Kunder.Add(kunde.KundeNummer, kunde);
        OnCollectionChanged(
            new NotifyCollectionChangedEventArgs(
                NotifyCollectionChangedAction.Add, kunde));
    }

    //Ditto for Remove() og Clear()

    public event NotifyCollectionChangedEventHandler CollectionChanged;
    protected virtual void OnCollectionChanged(
        NotifyCollectionChangedEventArgs args)
    {
        if (CollectionChanged != null)
            CollectionChanged(this, args);
    }
}
```

En controller der også understøtter events

- Et view kan abonnere på tilstandsskift direkte fra modellaget
- Alternativt / som supplement kan controlleren fungere som mellemmand – kan bla. aggregere events fra flere modelklasser.

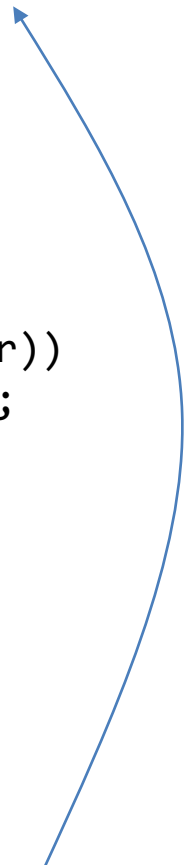
```
public class KundeAdministrationController: INotifyCollectionChanged {
    private KundeContainer _Kundeliste; //Reference til model

    public KundeAdministrationController() {
        _Kundeliste = new KundeContainer();
    }

    public void OpretKunde(Kunde k) {
        if (_Kundeliste.ContainsKundeWithKundenummer(k.KundeNummer))
            throw new ArgumentException("Kunden findes allerede");
        //Opdatering af model - medfører event i modellen
        _Kundeliste.Add(k);
    }

    public bool SletKunde(Kunde k) {
        //Opdatering af model - medfører event i modellen
        return _Kundeliste.Remove(k);
    }

    //Når et view subscriber til CollectionChanged event på denne controller
    //subscriber i virkeligheden til modellens CollectionChanged event
    public event NotifyCollectionChangedEventHandler CollectionChanged {
        add { _Kundeliste.CollectionChanged += value; }
        remove { _Kundeliste.CollectionChanged -= value; }
    }
}
```

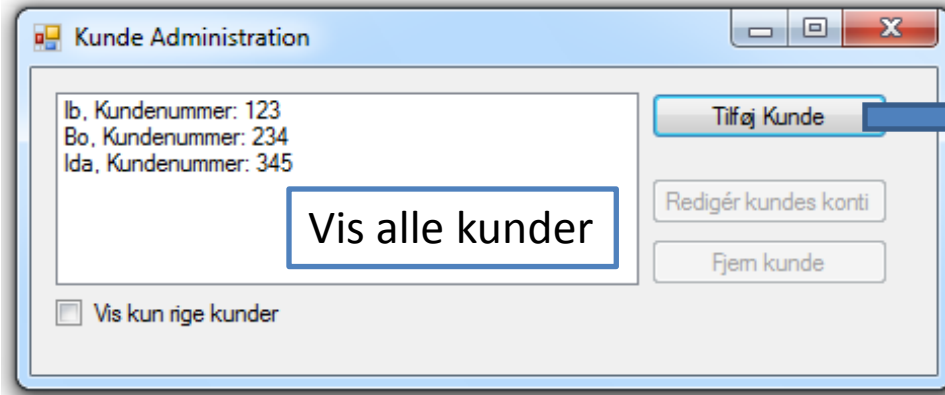


- Resten af controlleren herunder...
- Blot aflæsningsmetoder som ikke udløser events

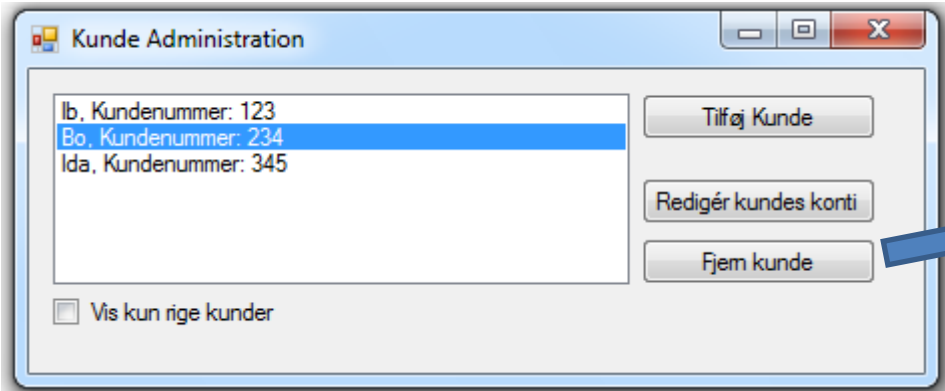
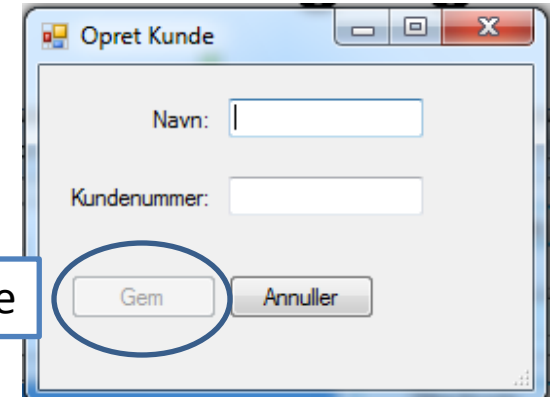
```
//...
public IEnumerable<Kunde> VisAlleKunder()
{
    return _Kundeliste;
}

public IEnumerable<Kunde> VisRigeKunder()
{
    //Aflæsning med 'Medium' kompleksitet
    foreach (Kunde kunde in _Kundeliste)
    {
        foreach (Konto konto in kunde.Konti)
        {
            if (konto.Saldo > 5000)
            {
                yield return kunde;
            }
        }
    }
}
} // KundeAdministrationController slut
```

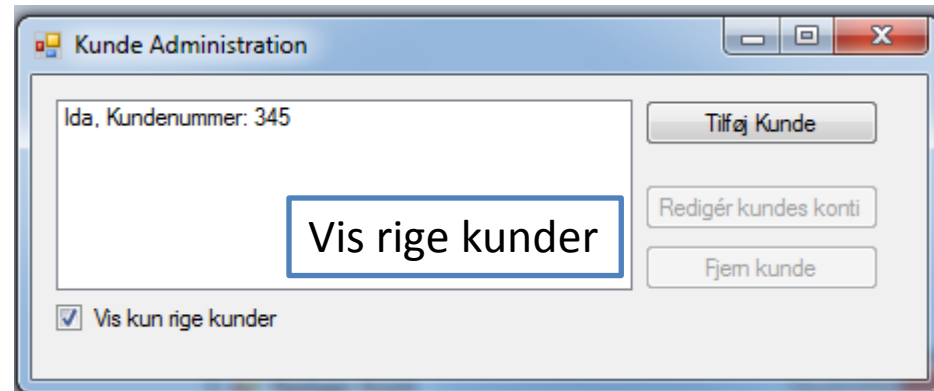
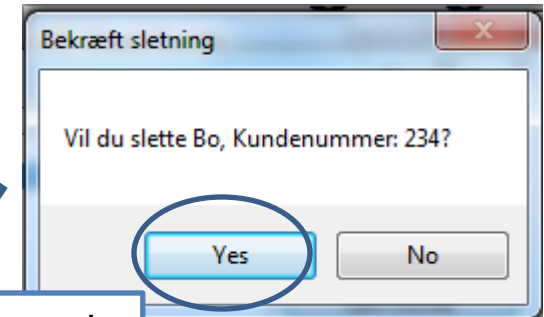
Administrér kunder



Opret Kunde



Slet Kunde



Live demo...

N:1 forhold mellem View og Controller
(opret kunde i separat vindue)


```
public partial class KundeAdministrationForm : Form {  
    private enum ViewState { IntetValgt, KundeValgt }  
    private ViewState _ViewState;
```

UI Tilstand

```
private KundeAdministrationController _Controller;
```

```
public KundeAdministrationForm() {  
    InitializeComponent();
```

Controller Ref.

```
    _Controller = new KundeAdministrationController();  
    //View lytter til ændringer på Controller  
    _Controller.CollectionChanged += controller_KundelisteÆndret;
```

Events

```
    SetViewState(ViewState.IntetValgt);
```

```
}
```

```
private void SetViewState(ViewState newState) {
```

Skift View tilstand

```
    _ViewState = newState;
```

```
    switch (_ViewState) {
```

```
        case ViewState.IntetValgt:
```

```
            fjernKundeButton.Enabled = false;
```

```
            redigerKundeKontiButton.Enabled = false;
```

```
            break;
```

```
        case ViewState.KundeValgt:
```

```
            ...  
        }
```

```
}
```

```
//... fortsat
```

```
private void controller_KundelisteÆndret(  
    object sender, NotifyCollectionChangedEventArgs e) {  
    OpdaterKundeVisning();  
}
```

Modtager event om kunder ændret

```
private void OpdaterKundeVisning() {  
    IEnumerable<Kunde> kundeliste;  
    if (visRigeKunderCheckBox.Checked)  
        kundeliste = _Controller.VisRigeKunder();  
    else  
        kundeliste = _Controller.VisAlleKunder();
```

Controller laver arbejdet

```
//Udfyld listbox med kunder  
kunderListBox.Items.Clear();  
foreach (Kunde k in kundeliste)  
{  
    kunderListBox.Items.Add(k);  
}
```

View sørger for visning

```
private void tilføjKundeButton_Click(object sender, EventArgs e) {  
    SetViewState(ViewState.IntetValgt);
```

```
OpretKundeForm opretKundeForm = new OpretKundeForm(_Controller);  
opretKundeForm.ShowDialog();
```

```
}
```

Controller deles af flere vinduer

Generelt


- View
 - Foretager typisk simpel data validering. F.eks. Er indtastet saldo en gyldig decimal type?
 - Skal ikke foretage logisk data validring: Feks. "Findes kunde allerede i systemet?"
 - Typisk konstruerer input data typer (f.eks. Kunde) i View laget, da det ellers kan give anledning til metode-overload eksplosion i controlleren.
- Controller
 - Som udgangspunkt én per use case – men kan sagtens variere.
 - Kan levere View-venlig repræsentation af model-data – på en ikke view-specifik måde vel at mærke!
 - Eksempel: Konstruér en data type der aggregerer data fra flere klasser i modellaget / lav virtuelle properties baseret på udregninger / ...
 - Skal vi gøre det?
 - Hvis det er generelt anvendeligt – ja.
 - Hvis det kun er til ære for en specifik brugergrænseflade – nej.

Persistenslag

- Det er god skik at indkapsle persistens operationer i et **persistenslag** (aka **Data Access Layer (DAL)**).
- Herved undgår man afhængighed af én bestemt måde at persistere data på.
- Lav et DAL interface der angiver mulige persistens operationer (CRUD) for data i modellen.
- Lav derefter konkrete DAL klasser der indeholder kode til at gemme/hente fra specifikke underliggende data kilder (DB, XML, Web Service, ...).
- Controlleren tager sig af interaktionen med DAL.


Ovenstående definition af et DAL refererer, mere specifikt, til det såkaldte [Repository mønster](#)

```
interface IKundeDAL
{
    Kunde Create(Kunde nyKunde);
    Kunde Update(Kunde eksisterendeKunde);
    Kunde Delete(Kunde eksisterendeKunde);
    //Hent alle kunder
    IEnumerable<Kunde> Read();
    //Hent kunder der opfylder prædikat
    IEnumerable<Kunde> Read(Func<Kunde, bool> filter);
    //Hent den enkelte kunde der opfylder prædikatet
    Kunde ReadSingle(Func<Kunde, bool> filter);
}
```



EF implementation: KundeEntiteterDAL

```
public interface IKontoDAL
{
    //Opret konto for kunde og returner den oprettede kunde eller null
    Konto Create(Konto nyKonto, Kunde eksisterendeKunde);
    Konto Update(Konto eksisterendekonto);
    Konto Delete(Konto eksisterendeKonto);
    //Hent alle konti for kunde
    IEnumerable<Konto> Read(Kunde eksisterendeKunde);
}
```



EF implementation: KontoEntiteterDAL

Eksempel

```
public class KundeAdministrationViewModel: INotifyPropertyChanged {
    IKundeDAL _KundeDAL;
    IKontoDAL _KontoDAL;

    public KundeAdministrationViewModel(
        IKundeDAL kundeDAL, IKontoDAL kontoDAL) {
        _KundeDAL = kundeDAL;
        _KontoDAL = kontoDAL;

        IndlæsKunderFraDAL();
        //...
    }

    private void IndlæsKunderFraDAL() {
        IEnumerable<Kunde> dbKunder = _KundeDAL.Read();
        foreach (Kunde curKunde in dbKunder) {
            foreach (Konto curKonto in _KontoDAL.Read(curKunde)) {
                curKunde.TilføjKonto(curKonto);
            }
            _Kundeliste.Add(curKunde);
        }
    }
    //...
}
```

Objekt-Orienteret Design

- "The only constant is change" → forvent ændringer.
- Design smells
 - **Rigidt**: Det er svært at lave ændringer. En ændring kræver en kaskade af ændringer
 - **Skrøbeligt**: Programmet går i stykker mange steder når der laves en ændring
 - **Immobil**: Dele der kunne være generelt anvendelige, kan ikke udtrækkes.
 - **Viskost**: Det er lettere at gøre det forkerte (et hack) end det er at gøre det rette.
 - **Unødig kompleksitet**: Forberedelse på hændelser der aldrig sker
 - **Unødig gentagelse**: Ændringer skal foretages mange steder.
 - **Uigennemsigtighed**: Kode bliver ulæselig med alderen.
- Følg SOLID principperne for at undgå design smells

Single-Responsibility Princippet (SRP)

- En klasse skal kun have én grund til at ændre sig.
- (En klasse skal have høj samhørighed)
- Ændringer -> rebuild, retest, redeploy – og ændringer kan "cascade".
- En klassisk begynderfejl: Klasse indeholder både forretningslogik, UI logik og persistenslogik.

Open-Closed Princippet (OCP)

- *Software entiteter (klasser, moduler, funktioner, etc.) skal være åbne for udvidelse, men lukkede for modifikationer*
- Løsningen er ofte polymorfisme

```
class Customer
{
    //Skal ændres hvis vi senere tilføjer nye kundetyper

    public decimal GetDiscount()
    {
        if (this is GoldCustomer)
            return .5m;
        else //SilverCustomer
            return .25m;
    }
}

class GoldCustomer : Customer { }
class SilverCustomer : Customer { }
```

Fragile Base Class -> Open Closed Princip

```
class Garage
{
    List<Car> _cars = new List<Car>();

    public virtual void AddCar(Car c)
    {
        _cars.Add(c);
    }

    public void AddAll(List<Car> moreCars)
    {
        foreach (Car c in moreCars)
            this.AddCar(c);
        NEW: _cars.AddRange(moreCars);
    }

    public virtual int NumCars
    {
        get { return _cars.Count; }
    }
}
```

```
class GarageWithCount : Garage
{
    int _numCars;

    public override void AddCar(Car c)
    {
        base.AddCar(c);
        _numCars++;
    }

    public override int NumCars
    {
        get { return _numCars; }
    }
}
```

Liskov Substitution Princippet (LSP)

- *Subtype skal kunne erstatte deres supertyper (fornuftigt!)*
- Subtyper skal overholde IS-A forholdet ift supertype
- Subtyper skal overholde supertypens kontrakt.
- En kontrakt kan angive *preconditions* (krav til kalder), *postconditions* (krav til metode) og *objekt invarianter* (krav til objekt tilstand).
- En kontrakter kan gøres eksplicit vha. unit tests eller Design By Contract ([C# Code contracts](#))
 - Code contracts tjekker kontrakter på compile tidspunktet og er en del af dokumentationen (med Sandcastle)
- En subklasse må ikke tilføje preconditions (forstærkede krav), men må gerne tilføje postconditions (forstærkede garantier)

```
class Rectangle
{
    public static double Area(double length, double width)
    {
        if (length < 0 || width < 0)
            throw new ArgumentOutOfRangeException();

        return length * width;
    }

    public static double Area2(double length, double width)
    {
        Contract.Requires<ArgumentOutOfRangeException>(
            length >= 0 && width >= 0);
        Contract.Ensures(Contract.Result<double>() == length * width);

        return length * width;
    }
}
```

Interface Segregation Princippet (ISP)

- Interfaces (APIer) skal være slanke (have høj samhørighed)
- Kan være fristende at lave "schweizerkniv klasse" – subklasser degenererer dog mere og mere

```
class Vehicle
{
    public void MetodeDerKunErRelevantforBusserOgLastbiler() { }
    public void MetodeDerKunErRelevantForPersonbilerOgTaxier { }
    public void MetodeDerKunErRelevantForMotorcyklerOgKnallerter() { }
}
```

- Almindelige løsninger:
 1. Lav mellem-subklasser (f.eks. "BigVehicle" for Busser og Lastbiler)
 2. Lav komponent-klasser (f.eks. "BigVehicleFunctionality" og brug aggregering i relevante klasser – bus og lastbiler)
- Tilsvarende: Tilføj aldrig metoder til et interface – lav i stedet et nyt interface der arver fra det gamle.

Dependency Inversion Princippet (DIP)

- *Abstraktioner skal ikke afhænge af detaljer. Detaljer skal afhænge af abstraktioner.*
- Dependency Inversion = dependency injection
- Navn: Princippet vender den måde “nogen” folk tænker om OO design

```
public class KundeAdministrationViewModel: INotifyPropertyChanged {  
    IKundeDAL _KundeDAL; //abstraktion  
    IKontoDAL _KontoDAL; //abstraktion  
  
    public KundeAdministrationViewModel(  
        IKundeDAL kundeDAL, IKontoDAL kontoDAL) {  
        _KundeDAL = kundeDAL; //new KundeDAL_Sql();  
        _KontoDAL = kontoDAL; //new KontoDAL_Sql();  
    }  
}
```

Fin