# OO programmerings-opgave

Denne opgave anvender kode der allerede er lavet på forhånd. Koden kan ses og hentes på følgende adresse: https://github.com/Andreas-Severin-Nielsen/OO-Loan

Filerne kan også hentes som en zip-fil: https://github.com/Andreas-Severin-Nielsen/OO-Loan/archive/refs/heads/master.zip

Opgaven er inddelt i flere underopgaver, hvor de starter i den lette ende, og gradvist bliver sværere. Opgaverne omfatter både at skrive ny kode og nye filer, og at rette i eksisterende kode og kodefiler. Det er derfor nødvendigt at forstå det der allerede er kodet, og dette dokument vil derfor først gennemgå de principper der er anvendt i den eksisterende kode.

## Nødvendig viden

### Om den eksisterende kode

Kodeprojektet er lavet i C# og Visual Studio, og fungerer i den form som den eksisterer i skrivende stund. Projektet er en simpel løsning for et udlåns-system, der har registrerede brugere og enheder, og kan oprette og afslutte udlån. Det er kun muligt for en bruger at låne en enkelt enhed ad gangen, og der er ikke lavet nogen løsning for "resistent data", dvs. data gemmes ikke når programmet lukkes.

### Klasser og objekter

Som titlen antyder, er løsningen lavet som en objektorienteret løsning, dvs. der anvendes klasser og metoder, frem for datafelter og funktioner. Der findes online træningsmaterialer om OO-programmering i C# på følgende links:

- https://www.guru99.com/c-sharp-class-object.html
- https://www.w3schools.com/cs/cs_oop.asp
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/objects
- https://www.codecademy.com/learn/learn-c-sharp/modules/learn-csharp-classes

Det er ikke nødvendigt at lave alt der nævnes i de ovenstående vejledninger, men det er en god idé at se dem igennem, og nærlæse og arbejde med det der er nyt eller man ikke forstår. Da forståelse af klasser og objekter er grundlaget for alt det der følger i opgaven, skal dette være på plads først.

### Forklaring på 3 tier inddeling

Måden man vælger at genskabe en "begrebsverden" i koden afhænger af:

- Selve begreberne
- Måden programmøren anskuer begreberne på
- Behovet for detaljer og kompleksitet for at få løsningen til at fungere

Hvilke klasser man ender med at anvende, kan derfor variere fra person til person, og fra situation til situation. En klassisk måde at anskue og opdele klasserne på er 3-tier inddelingen (eller *Three-tier architecture*), hvor klasserne i en applikation inddeles i 3 ”lag”:

1. Præsentations-laget, der indeholder grænseflade til brugeren
2. Forretnings-laget, der indeholder logikken og kontrol over processerne der forandrer data
3. Data-laget (eller DAL – Data Access Layer), som indeholder de data der arbejder med

## Præsentationslaget

Dette lag samler alt hvad der har med brugerens interaktion med systemet at gøre. Det er uanset om der er tale om et programvindue, en programkonsol, stemme / lys / lyd / følelse / neural interaktion mellem systemet og brugeren. Der er en række fordele ved at samle alt hvad der har med bruger-interaktion i sit eget lag:

- Brugere skifter ofte karakter, behov, tilgang eller mening om måden de interagerer med systemerne på. Det er lettere at finde de steder i koden der skal ændres, hvis de er samlet eet sted.
- Der er ofte behov for omfattende kode for både at sikre en god brugeroplevelse og sikre at der ikke kommer forkert eller ødelæggende data ind i systemet. Det er igen lettere at administrere, hvis det bliver holdt eet sted.
- Når brugere ønsker at ændre på noget for at få en bedre oplevelse, er det ikke nødvendigt at ændre på bagvedliggende data eller processer, og de bliver holdt udenfor. Man risikerer ikke at lave om på dem ved en fejl.

I dette programeksempel er der lavet en mappe der hedder ”Userinterface”, og det indeholder følgende klasser, der sørger for interaktion med brugeren i form af styring af konsol vinduet:

- Mainscreen: Hoved skærmbillede til programmet og dets muligheder
- Loanscreen: Sørger for bruger interaktion omkring lån
- Unitscreen: Sørger for interaktion omkring administration af enheder til udlån
- Userscreen: Sørger for interaktion omkring administration af brugere der kan låne enheder
- Menuselection: En hjælpeklasse. Det er en form for et generisk menu-valg, som sætter menuvalg op på samme måde uanset hvor den kaldes fra.

## Forretningslaget

Forretningslaget indeholder ”forretningslogikken”, dvs. de regler, politikker og processer der anvendes til at behandle de data, som virksomheden eller programmet skal arbejde med. Når disse processer først er beskrevet og implementeret, får de som regel lov til at fungere længe uden at blive ændret – ændring af måden data behandles på skal normalt godkendes rundt omkring i hele virksomheden (af både ledelse, brugere, juridisk afdeling osv.) Så længe klasserne i forretningslaget opfylder deres formål, får de normalt lov til at fungere uberørt.

I programeksemplet er der følgende kontrolklasser:

- LoanManager: Sørger for korrekt behandling af data omkring lån
- UnitManager: Sørger for korrekt behandling af data omkring enheder
- UserManager: Sørger for korrekt behandling af data omkring brugere

## Datalaget

Datalaget gemmer på ”model” klasserne, dvs. de data-klasser og objekter som ”efterligner” de objekter fra den virkelige verden, som programmet skal arbejde med. Når de er på plads, er der

normalt kun behov for at ændre på dem når behovet for opbevaring af data ændrer sig. I dette programeksempel er der følgende model-klasser:

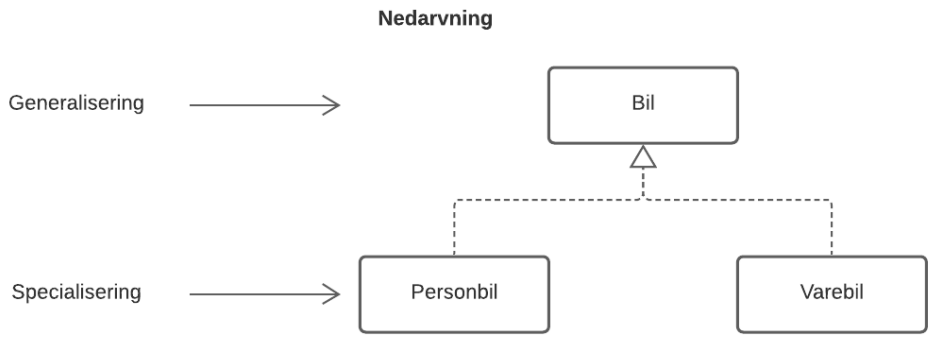
- School: Fungerer som den klasse, der indeholder alle andre model-klasser
- User: Indeholder data omkring de brugere der er registreret
- Headset: Indeholder data omkring headsets
- Laptop: Indeholder data omkring laptops
- Unit: Fungerer som en overordnet kategori, som både Headset og Laptop er en del af

## Object Interface – nedarvning, generalisering, specialisering

Når der anvendes objekt-orienteret udvikling og programmering, vil der også blive tale om begreber som nedarvning, generalisering, specialisering og interfaces.

### Nedarvning / Inheritance

Når en klasse arver fra en anden klasse, svarer det til, at den nye klasse overtager egenskaberne fra den originale klasse.




I ovenstående eksempel, nedarver Personbil og Varebil fra klassen Bil. Man kan sige følgende:

- Bil er en generalisering af Personbil og Varebil. En klasse for personbiler og en klasse for varebiler har flere ting til fælles, som kan samles i en generel klasse.
- Personbil og Varebil er specialiseringer af klassen Bil. Hvis man har en bil, kan man tilføje flere ting der gør det til en "speciel" type bil, som ikke er gældende for andre typer biler. Varebiler har f.eks. ikke de samme muligheder og begrænsninger i forhold til last eller passagerer, som personbiler har.

Til at beskrive hvordan et system er kodet eller skal kodes, anvendes et format der kaldes UML[1]. I UML tegnes nedarvning ved at den generelle klasse står øverst, mens de specialiserede klasser står under. Der tegnes en tom pil med stiplet linje fra de specialiserede klasser til den generelle klasse.

### Interfaces

Et interface i objekt-orienteret programmering er IKKE det samme som en grænseflade (dvs. brugergrænseflade eller grænseflade til f.eks. databaser). I OO-verdenen er der tale om en form for "kontrakt", hvor en generaliseret, men ellers tom klasse, fortæller hvilke metoder der skal ligge i de specialiserede klasser. Hvis man ser på eksemplet fra før, vil det ikke give nogen mening at oprette objekter af klassen "Biler". Man bliver nødt til at have en specialiseret form af klassen, dvs. enten en personbil, varebil, lastbil osv. Men man kan stadigvæk sige, at "biler" kan bestemme, hvad de

[1] https://www.uml-diagrams.org/

specialiserede klasser skal kunne. Det kan f.eks. være starte, slukke, køre fremad eller bakke. Dermed ved man, hvad en bil skal kunne – man ved bare ikke hvordan, eller hvilken motor der er i, brændstoftype, antal passagerer, mærke, formål osv. Det kommer først i de specialiserede klasser.

### Abstrakte klasser

En abstrakt klasse er en mellemting mellem et interface og en "normal" klasse. Det er ikke en "færdig" klasse der kan anvendes, men det er muligt at indsætte data og metoder, som de specialiserede klasser har tilfældes. Hvis man f.eks. har en generaliseret klasse "Biler", og den specialiseres til henholdsvis "forbrændingsbiler" og "elbiler" kan der f.eks. være følgende forskelle

- Forbrændingsbiler
    - Brændselstype (benzin, diesel)
    - CO2 udledning
    - Antal liter på brændstoftank
    - Antal km/l
    - Gearbox og antal gear
- Elbiler
    - Opladningshastighed
    - Antal km/kw
    - Batteri antal kwh

Selvom man har specialiseret biler til ovenstående to muligheder, er det alligevel ikke nok til at man kan oprette "fornuftige" biler – der mangler fortsat information om det er SUV, sportsbiler, varebiler eller lastbiler. Begge klasser skal derfor specialiseres yderligere, før det giver mening at lave konkrete objekter af klasserne.

## Dependency Injection

Der findes mange mønstre (*design patterns)* man kan strukturere koden i. Der er fordele og ulemper ved dem alle, men nogle patterns er gode at kende, og kommer ofte til sin ret. Dependency Injection er sådan et pattern, og mange virksomheder ser gerne, at du kender og anvender det ofte.

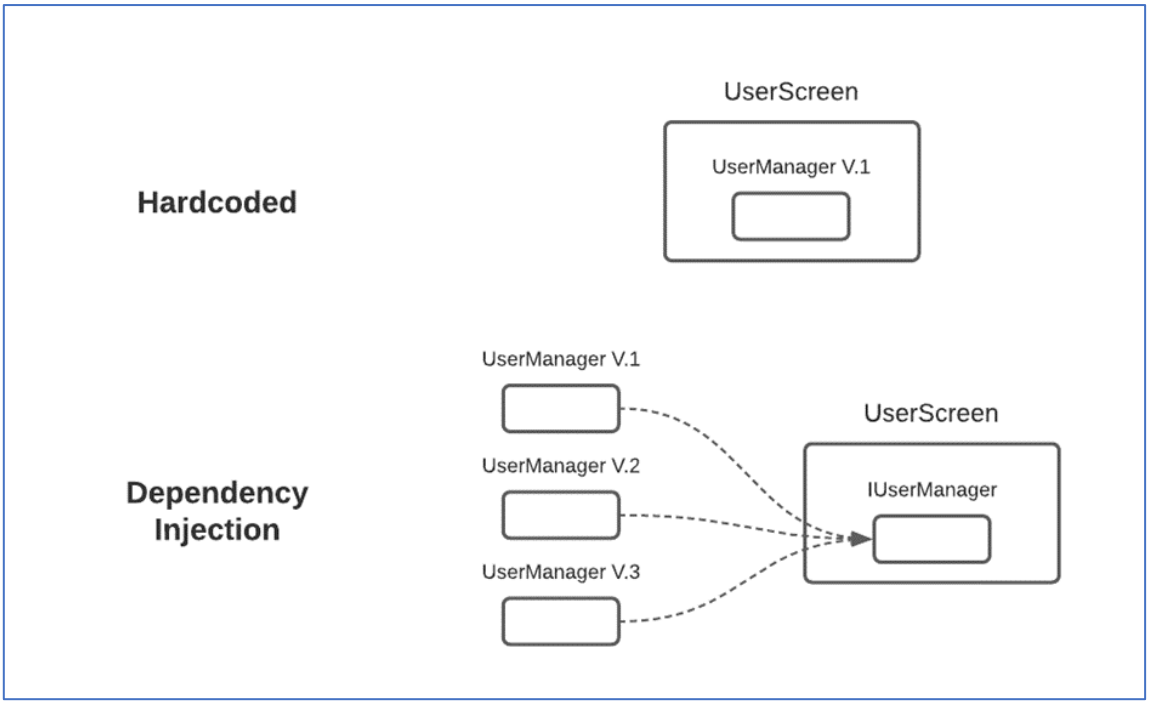
Dependency Injection er en måde at bestemme hvad et objekt skal arbejde med på kørselstidspunktet, idet objektet ikke *hard codes* til at anvende en anden klasse, men derimod får det "sendt" til sig på kørselstidspunktet. Et eksempel fra den udleverede kode:

Præsentations-laget har en klasse der hedder "UserScreen". Denne klasse sørger for at modtage og vise informationer vedr. administration af brugerne i systemet. Men forretningslogikken sker jo ikke her – den sker hos UserManagement. Så for at UserScreen kan igangsætte noget der har at gøre med brugerne, er den afhængig af (*Depending on*) et objekt af klassen UserManagement. En oplagt løsning på dette ville være at have følgende kode i klassen UserScreen:

```
UserManagement userManagement = new UserManagement()
```

Ved at indsætte denne kode direkte i "UserScreen" er den hardcoded til at anvende dette, og den har ikke noget valg.

En anden mulighed er at "indskyde" (*Injecting*) UserManagement ind i UserScreen. Dette kan gøres når UserScreen oprettes, eller når en funktion skal anvendes. Fordelen er, at man kan lave nye versioner af UserManagement, og de kan indsættes i UserScreen ved programkørsel. Derved bliver det muligt at lave nye og specielle versioner, der kan anvendes mere dynamisk.




Som det kan ses af ovenstående billede, vil der kun kunne anvendes én version af UserManager, hvis den bliver hardcoded ind i UserScreen klassen. Hvis der derimod anvendes Dependency Injection, afgøres det først under kørslen hvilken version der anvendes. Dette er praktisk hvis man skal skifte mellem f.eks. specielle test-versioner eller hvis der skal anvendes andre specialiserede klasser, der nedarver fra UserManager klassen.

# Opgave

## Opgave 1 (introduktion)

Denne opgave går ud på at tilføje en ny type enhed: Tablets. Følg nedenstående opgaverække:

- Lav en ny model-klasse under mappen `Model`, der hedder `Tablet`. Du kan hente inspiration fra f.eks. `Laptop` klassen.
    - Tablet skal indeholde følgende data:
        - Brand (mærke)
        - Model (serienavnet på modellen)
        - Size (skærmstørrelse)
- Find klassen `UnitManager` under mappen `Control`. Tilføj metoden `CreateNewTablet`. Hent evt. inspiration fra metoden `CreateNewLaptop`.
    - Husk at der skal anvendes argumenter i kaldet:
        - ID
        - Brand
        - Model
        - Size
    - ID for tablets skal starte med ”T”
    - Tabletter skal til service hver anden gang, ligesom Laptops
- Find klassen `UnitScreen` i mappen `Userinterface`.
    - Tilføj metoden `RegisterNewTablet`. Hent evt. inspiration fra `RegisterNewLaptop`.
    - Tilret `RegisterNewUnit`, så den også anvender tillader at oprette tablets. Husk at den skal kalde `RegisterNewTablet`, hvis brugeren vælger Tablet.
- Test at det virker, ved at oprette, låne og slette tabletter. Husk også at tjekke om den reserverer tabletter efter de har været udlånt 2 gange.

## Opgave 2 (let)

Denne opgave er et oplæg til at forbedre brugergrænsefladen, så den fungerer bedre. Det kan f.eks. være ved at ændre teksterne, farverne, rækkefølger på spørgsmål eller lignende. Alle klasser der er involveret i dette, ligger i mappen `Userinterface`. Se eventuelt nærmere på `Console.BackgroundColor` og `Console.ForegroundColor`. Baggrunde kan f.eks. inddeles efter område:

- Brugeradministration har en grønlig baggrund
- Enhedsadministration har en blålig baggrund
- Lånadministration har en gullig baggrund

## Opgave 3 (moderat)

Denne opgave skal forbedre interfacet. Det er i øjeblikket kun muligt at se listen af enheder og brugere ved at gå ind på menupunktet for sletning, hvilket ikke er intuitivt korrekt.

- Tilføj mulighed for at se lister af henholdsvis brugere og units, uden at skulle anvende menupunktet ”slet”.
    - Tilføj muligheden for at se brugere i `UserInterface.UserScreen()`. Anvend f.eks. `Usermanager.GetAllUsers()` i løsningen.
    - Tilføj muligheden for at se enheder i `UserInterface.UnitScreen`. Anvend f.eks. `UnitManager.GetAllUnits()` i løsningen.

## Opgave 4 (udfordrende)

I fortsættelse af opgave 3: Ud fra listen over brugere skal det være muligt at kunne redigere i informationerne, dvs. ændre navn osv.

- Lav om på listen for brugere der blev lavet i opgave 3, så det er muligt at vælge en af brugerne
- Lav brugergrænsefladen, den fortæller det der ændres på, den gamle værdi, samt indtastning af den nye værdi. Hvis der ikke indtastes noget og blot tastes [enter], skal den gamle information anvendes igen. Husk at ID ikke skal laves om!
- Husk at anvende 3-tier arkitekturen: data skal sendes fra Userinterface til Control til Model.

Husk på, at du ikke kan anvende f.eks. `UserManager.CreateNewUser`, da brugeren i så fald vil få et nyt ID – det er ikke meningen, når den blot skal have ændret sine andre data. Lav i stedet en ny funktion, f.eks `UserManager.EditUser(ny information)`. Den skal så finde den eksisterende post, og overskrive de gamle værdier med de nye.

## Opgave 5 (svær)

Lige nu bliver listen over enheder svær at overskue, hvis der kommer for mange enheder i systemet. Men det kan blive svært at lave en liste over enheder, der er let at overskue. I de tilfælde, hvor der skrives lister over enheder på skærmen, sørg for at listerne bliver inddelt efter hvilken type enhed der anvendes.

- En mulighed kunne være at indsætte et "Enum" datafelt, som fortæller om det er et headset, laptop, tablet osv. Der ligger allerede anvendelse af Enum i koden (den fortæller status på en enhed)
- En anden mulighed er at se på hvilken klasse der er tale om. Se nærmere på
    - `if (unit.GetType() == typeof(Laptop))`
    - https://docs.microsoft.com/en-us/dotnet/api/system.object.gettype?view=net-5.0

## Opgave 6 (svær)

I fortsættelse af opgave 5: Lav programmet om, så en bruger kan låne én af hver type enhed

## Opgave 7 (meget svær)

Tilføj en metode i `UserManager`, der sorterer listen af brugere alfabetisk efter navn. Husk at anvende metoden, så den anvendes når en liste af brugere skal vi