OBLIGATORISK OPGAVE

I4SWD

Gruppe 8

Navn	Studienummer	E-mailadresse
Frederik Andersen	201508251	201508251@post.au.dk
Nina Wiborg Mølgaard	201505613	201505613@post.au.dk
Theis Egsgaard Hansen	201506770	201506770@post.au.dk

Visitor Pattern

14. MAJ 2017

AU ENGINEERING

1 INDHOLDSFORTEGNELSE

2 Formål 2
3 Visitor Pattern 2
4 Sammenligning 3
4.1 Command Pattern 3
4.2 Servant Pattern 3
5 Resultater 4
5.1 Tax 4
5.2 Computerparts 7
6 Konklusion 8

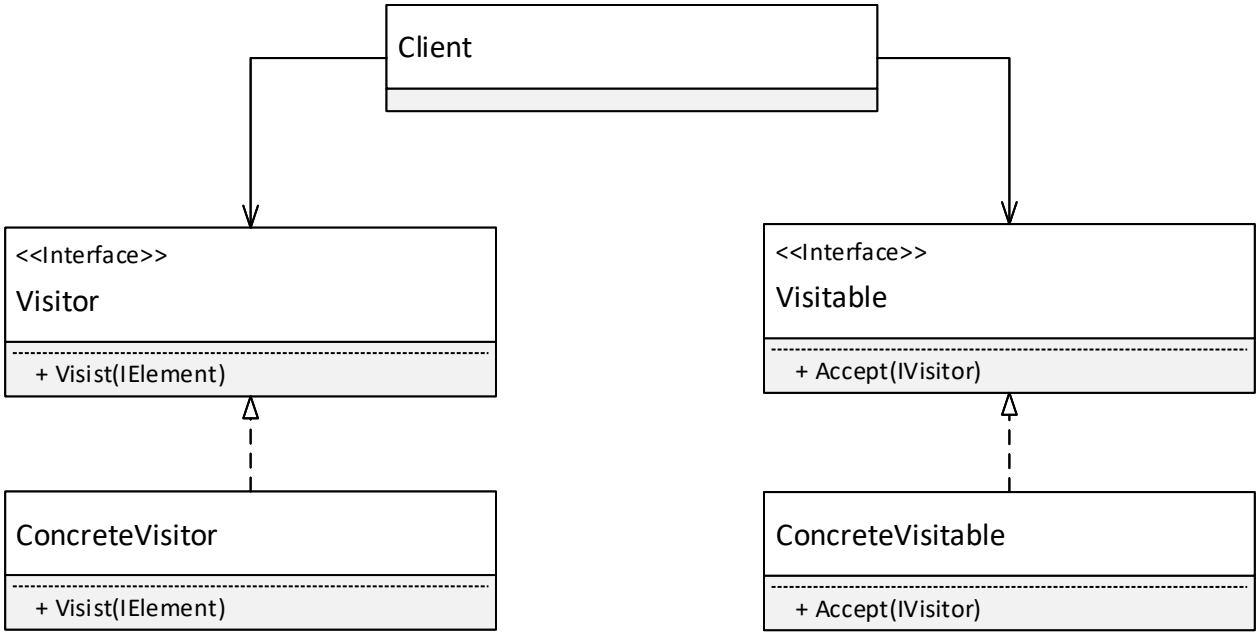
2 FORMÅL

Denne opgave har til formål at beskrive et design pattern. Gruppen har valgt at, og har fået godkendt at snakke om Visitor Pattern.

Visitor Pattern tillader at når nye metoder til en klasse af forskellige type, skal tilføjes, så kan dette gøres uden, at ændre meget i disse klasser. Dette pattern hjælper dermed med at realisere SOLID princippet Open/Closed [1].

3 VISITOR PATTERN

Visitor pattern er et *Behavioral pattern*. *Behavioral patterns* omhandler patterns, som har en intern kommunikation eller realisering. Dette øger fleksibiliteten og ofte gør, at programmet ofte overholder SOLID principperne mere.



Figur 3-1 - Overordnet klassediagram over Visitor pattern

Som der ses på Figur 3-1 har *Visitor* pattern altid to interfaces. Interfaces *Visitor* hedder altid dette, men navnet på Interfaces *Visitable* kan ændres – i denne rapport kalder vi dog denne klasse for *Visitable*.

Klasserne, som nedarver fra *Visitable*, ønsker at udvide deres funktionalitet derfor opretter man en *Visitor* klasse, som implementerer den nye ønskede funktion. Dermed, når man kalder "Accept(IVisitor)" så bliver den nye funktionalitet kørt.

Visitor Pattern giver en mulighed, som ingen andre behavioral patterns [2] har, og de gør det nemmere at implementerer programmer, hvor der er mange objekter, der kan vælge imellem mange valgmuligheder.

Nogle af de konsekvenser der er ved at bruge *visitor pattern* er at man får mange klasser, hvor der skal meget forberedelse til, for at få de enkelte klasser til at gøre noget. Disse klasser fortæller heller ikke

eksplicit hvad det nødvendigvis er der kommer til at ske, når man kalder *accept* funktionen. Koden bliver derved mere uoverskuelig [3].

4 SAMMENLIGNING

4.1 COMMAND PATTERN

Command Pattern omkranser en enkelt handling funktionalitet, ligesom *Visitor Pattern*.

Command Pattern omkranser dog også funktionaliteten, således at denne funktion kan blive udført når der er behov for det. Det eneste man skal gøre er at kalde ”Execute()” [4].

Command pattern er derfor lidt mere, som en Black Box, hvor man kun ved, at man skal kalde ”Execute()”, og man ikke har nogen idé om, hvad der sker inde i den. *Visitor Pattern* her skal kende den *Visitor*, som man gerne vil bruge.

På grund af denne funktionalitet kan Command Pattern ofte bruges til at undo eller redo funktionalitet. I modsætning bruges Visitor pattern mere til at implementerer nye funktioner uden at ændre for meget i den originale klasse.

4.2 SERVANT PATTERN

Servant pattern bruges til at udføre én specifik funktionalitet til en bestemt gruppe af klasser. Så i stedet for at implementerer denne funktionalitet i hver klasse, så defineres det i vores Servant [5].

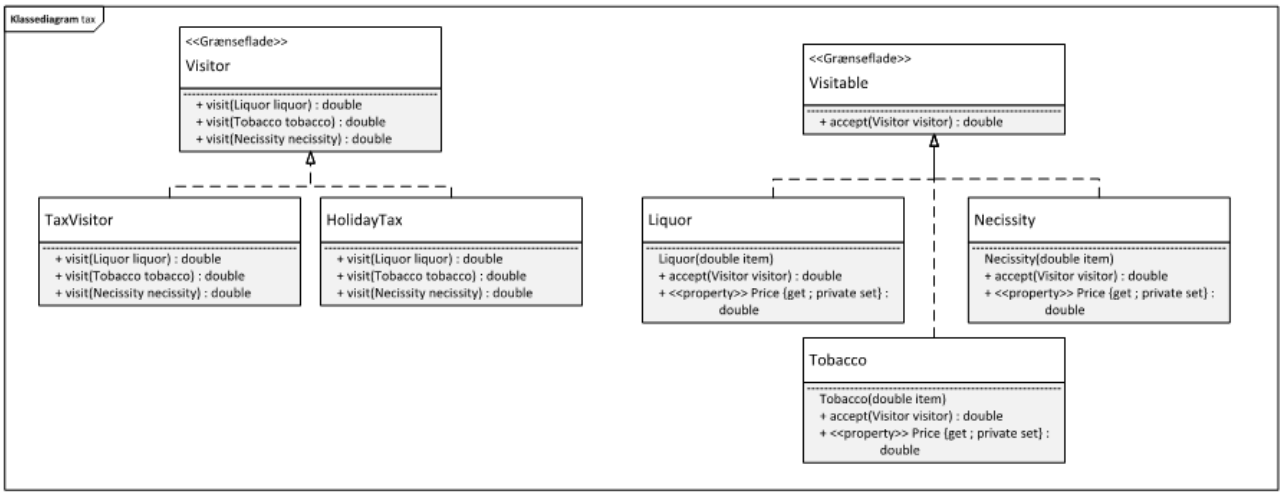
Denne ide findes også i *Visitor*, hvor vi undgår at udvide den bestemte gruppe af klasser og i opretter funktionaliteten i en ny klasse.

Servant pattern er dog mere simpel end *Visitor pattern*, da denne klasse kun implementerer funktionaliteten i én udgave.

5 RESULTATER

Gruppen har berørt flere forskellige eksempler i forhold til, hvad man kan bruge Visitor Pattern til.

5.1 TAX



Figur 5-1 - Klassediagram over Tax

Figur 5-1 viser et klassediagram over Tax programmet [6]. Programmet er beregnet til, at udregne den endelige pris på et produkt med en pågældende skat. Dette system kunne bruges i et land, hvor skatten ændres alt afhængig af varen og sæson, så det fungerer således, at når man køber sine dagligvare, så er der forskellig skat på alkohol kage og forskellig skat i forhold til feriesæsonen.

Et eksempel på implementering af klassen *TaxVisitor* kan dermed ses i Kode 5-1. Klassen *HolidayTax* ligner i opbygningen meget *TaxVisitor*, og dermed er dette ikke blevet indsat.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tax
{
    public class TaxVisitor : Visitor
    {
        public double visit(Liquor liquor)
        {
            Console.Write("Liquor item: Price with Tax\t");
            return (double) (liquor.Price * 0.23) + liquor.Price;
        }

        public double visit(Tobacco tobacco)
        {
            Console.Write("Tobacco item: Price with Tax\t");
            return (double) (tobacco.Price * 0.25) + tobacco.Price;
        }

        public double visit(Necessity necessity)
        {
            Console.Write("Necessity item: Price with Tax\t");
            return (double) (necessity.Price * 0.05) + necessity.Price;
        }
    }
}
```

Kode 5-1 – TaxVisitor klasse

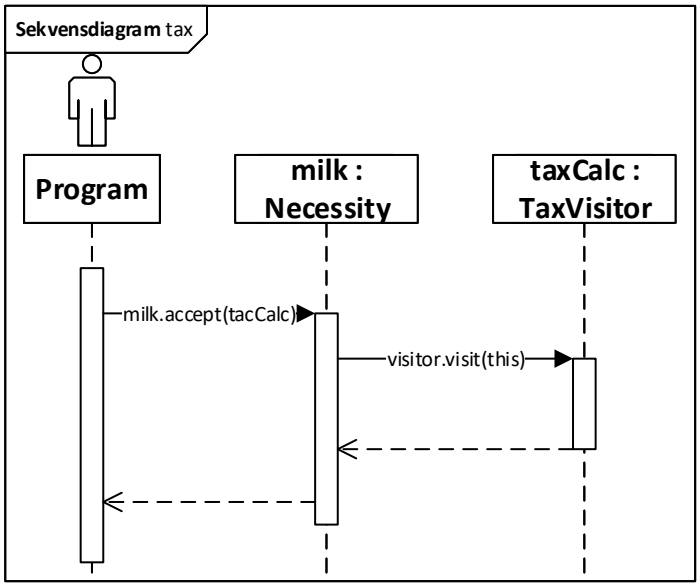
Et eksempel på en Visitable (Necissity) kan ses i Kode 5-2. Liquor og Tobacco ligner Necissity i opbygningen, dermed bliver der kun en realisering af *Visitable*, der bliver vist.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tax
{
    public class Necessity : Visitable
    {
        public double Price { get; private set; }

        public Necessity(double item)
        {
            Price = item;
        }
        public double accept(Visitor visitor)
        {
            return visitor.visit(this);
        }
    }
}
```

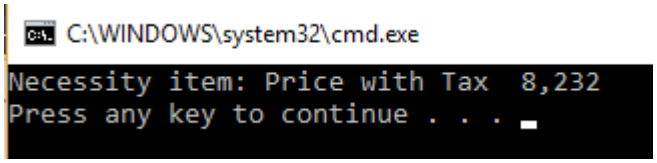
Kode 5-2 – Necessity klasse



Figur 5-2 - Eksempel på sekvensdiagram

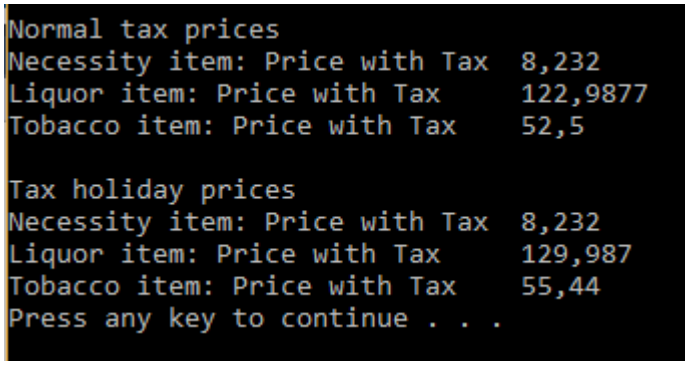
Figur 5-2 viser et eksempel på et sekvensdiagram over Tax systemet. Her er der taget udgangspunkt i, at vi har et Necessity, milk, om vi bruger accept funktionen til at tjekke skatten med.

Resultatet af Figur 5-2 kan ses på Figur 5-3.



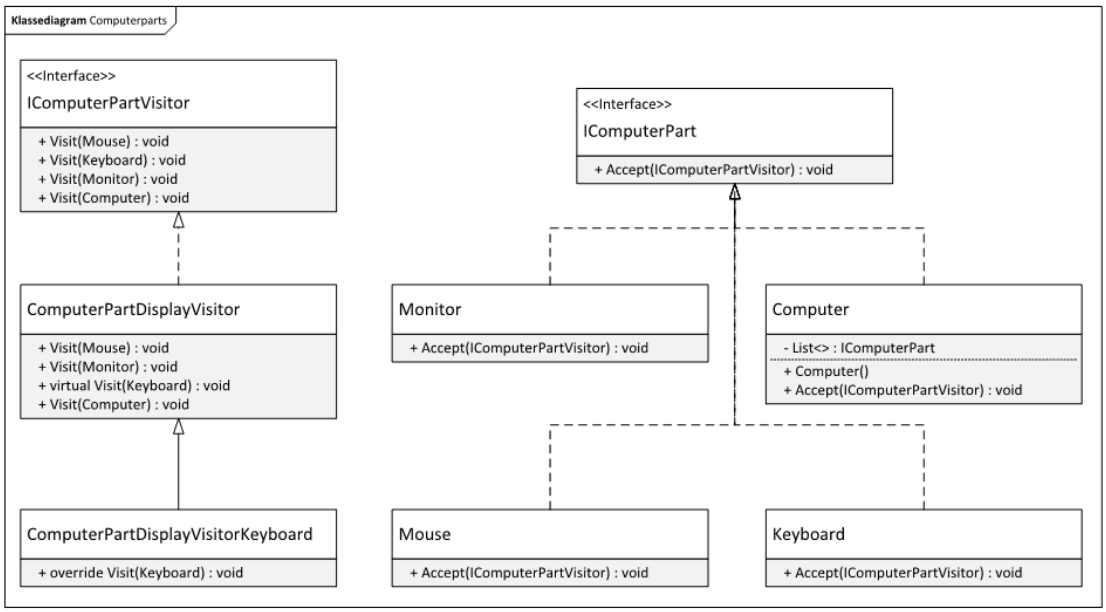
Figur 5-3 - Resultat af kun milk

Resultatet fra et samlet system kan på Figur 5-4.



Figur 5-4 - Resultat fra det samlede tax system

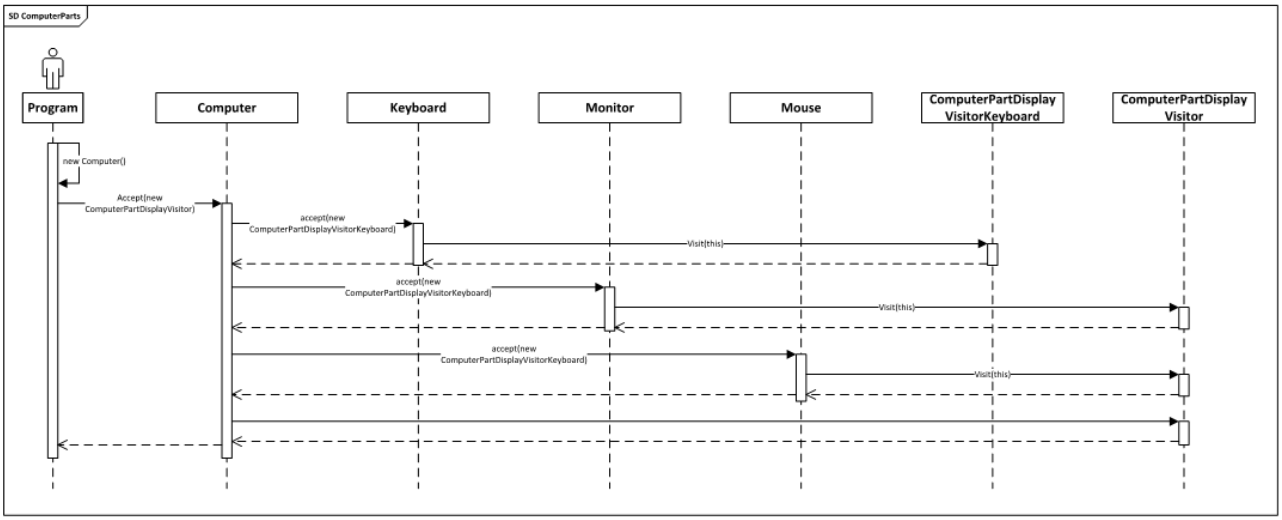
5.2 COMPUTERPARTS



Figur 5-5 – Klassediagram over eksemplet computerparts

På Figur 5-5 – Klassediagram over eksemplet computerparts ses et klassediagram over et andet eksempel, som benytter sig af visitor pattern. Denne løsning bygger over *Computerparts* [7]. Der bliver oprettet et IComputerPart, som definerer en accept funktion til alle nedarvede klasser af dette interface. Disse klasser kaldes for konkrete klasser og bruger accept funktionen til, at accepterer et visitor objekt. Der bliver nemlig også defineret et interface kaldet IComputerPartVisitor. De klasser, som nedarver fra IComputerPartVisitor kaldes for konkrete visitors, som håndterer et visitor objekt,

når der er kommet en accept fra IComputerPart. Ydermere har vi lavet en virtuel funktion til keyboard i ComputerPartDisplayVisitor, til at *override* keyboard funktionen, hvis der ønskes en anden udskrift, som kunne være hvis *keyboardet* ikke længere kunne skabe forbindelse til *computeren*, som også kan ses på Figur 5-7.



Figur 5-6 - Sekvensdiagram over ComputerParts

```
Keyboard has failed to connect. Now die.
Displaying monitor.
Displaying mouse.
Displaying computer.
Press any key to continue . . .
```

Figur 5-7 - Resultat fra Computerparts systemet

En potentiel svaghed ved dette design pattern, er at hvis man skal bruge flere forskellige *visit* metoder, for et objekt, kan man enten implementere en helt ny instans af *Visitor* interfacet, eller man kan arve fra den første implementering.

6 KONKLUSION

Visitor Pattern er en et godt pattern til at implementere polymorfisk kode, men desværre bliver dette pattern ikke brugt særlig ofte, da man skal lave en masse kode inden man rent faktisk kan komme til at implementere den funktionalitet, som man ønsker.

7 REFERENCES

[1] en.wikipedia, »en.wikipedia,« wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Visitor_pattern.

[2] sourcemaking, »sourcemaking.com,« sourcemaking, [Online]. Available: https://sourcemaking.com/design_patterns/visitor. [Senest hentet eller vist den 11 05 2017].

[3] Nice, »nice.sourceforge.net,« Nice, [Online]. Available: http://nice.sourceforge.net/visitor.html.

[4] sourcemaking, »sourcemaking.com,« sourcemaking, [Online]. Available: https://sourcemaking.com/design_patterns/command. [Senest hentet eller vist den 11 05 2017].

[5] en.wikipedia, »en.wikipedia,« wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Servant_(design_pattern). [Senest hentet eller vist den 11 05 2017].

[6] D. Banas, »Youtube,« Google, [Online]. Available: https://www.youtube.com/watch?v=pL4mOUDi54o. [Senest hentet eller vist den 11 05 2017].

[7] tutorialspoint, »tutorialspoint.com,« tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/design_pattern/visitor_pattern.htm. [Senest hentet eller vist den 11 05 2017].