1 STRATEGY PATTERN + TEMPLATE METHOD PATTERN

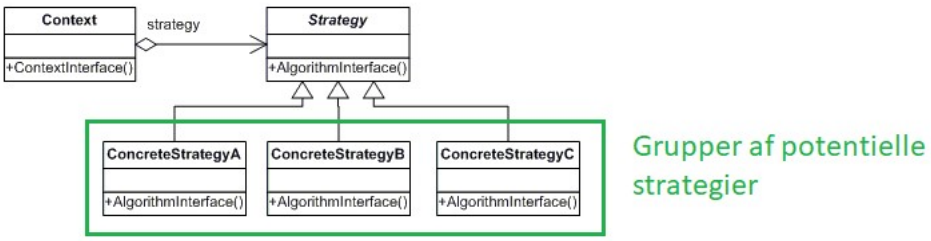
Hvad er et software design pattern?

- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

Strategy Pattern:

Behavioral pattern: Ændre måden et bestemt stykke kode opfører sig.

- Det bliver muligt at ændre (vælge) strategien run-time.
- Laver objekter som repræsenterer forskellige strategier.

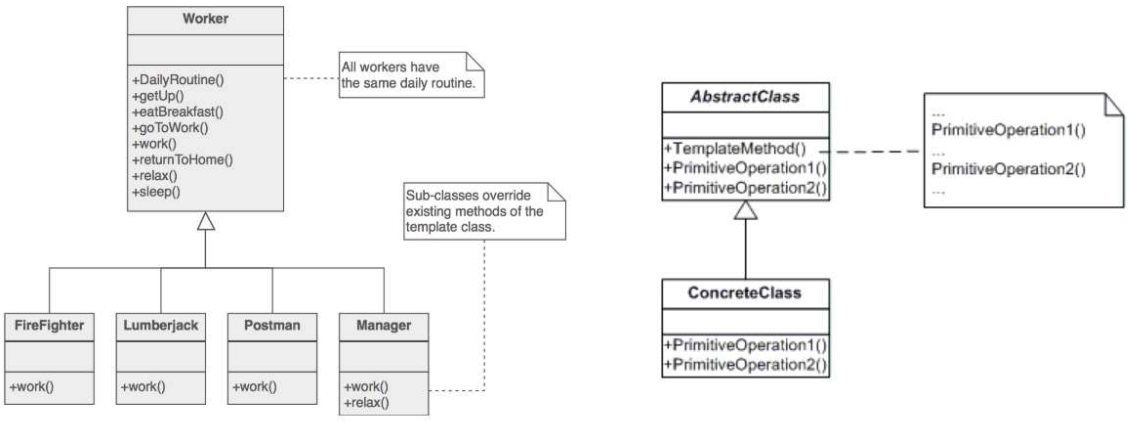


- Strategimønsteret lader "brugeren" vælge den ønskede strategi.
- Context objekt: Hvis opførsel varierer alt efter hvilken strategi som vælges. (Udskifte implementering)
- Strategi interface → definer en aktion, og derefter skal de konkrete strategier implementer disse.

Eksempel: Ønsker at gemme et billede, ved runtime vælges format som billedet skal gemmes i (JPG, PNG…)

Template Method Pattern:

- Definer en skabelon for en algoritme, og uddeler nogle skridt til subklasser.
- Dette mønster lader subklasser (underklasser) redefinere enkelte skridt.
- Mindsker duplikering → fordi abstrakte klasse definerer valg alle konkrete klasser har til fælles.
- Godt design: System af klasser hvor funktionalitet kun afviger let af hinanden



Eksempel: En arbejdsdag → Stå op, spis, arbejde, afslapning og sove. (disse er fælles for alle arbejdere, men forskellen, arbejde, implementeres i de konkrete underklasser.

Sammenlign Strategy og Method Template mønster:

- Strategi: Man kan ændre hele strategien run-time.
    - Interface definerer aktionen
        - Afledte klasser implementeres hver sin version af denne aktion.
    - Anvendes når program-flowet ikke er fastlagt.
        - Skifte implementering run-time.

- Template: Kun nogle ting ændres og resten forbliver det samme
    - ”Konstante” step implementeres i en abstrakt basis klasse.
    - Varierende steps implementeres i de afledte klasser
    - Anvendes når der er et bestemt flow alle klasse skal overholde
        - Opførsel ændres compile time.

SOLID principper (Strategy):

- Open-Closed Principle (OCP): Implementering i afledte klasser, ingen ændring hvis der tilføjes flere strategier.
- Single Responsible Principle (SRP): Ansvaret for en bestemt strategi er afkoblet fra context klassen som benytter disse.
- Liskov Substitution principle (LSP):
- Interface Segregation (ISP): Konkrete strategier implementer et interface som kun giver de bestemte nødvendigheder
- Dependency Inversion Principle (DIP): Context og konkrete strategier afhænger af et inteface og er ikke koblet direkte.

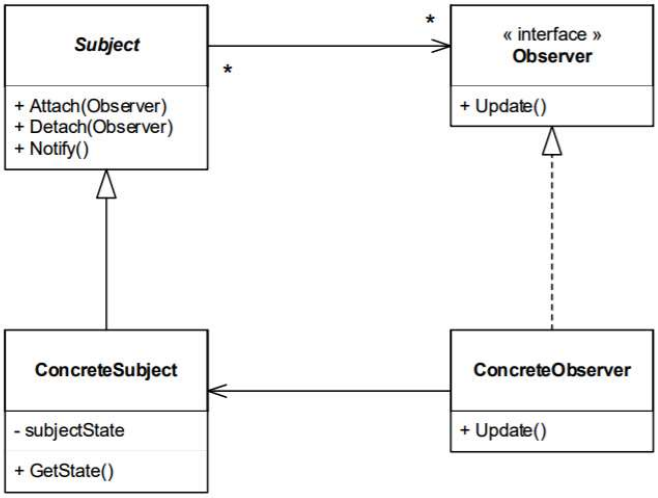
2 OBSERVER PATTERN

Hvad er et software design pattern?

- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

Observer Pattern:

- Behavioral pattern.
- Anvendes når der er et *en til mange* forhold mellem objekter. En vejs kommunikation
- Observer ønsker at holde øje med subjekt abonnere på tilstandsændringer i subjekt.



Subject: Abstrakt basis klasse, for alle data subjekter
ConcreteSubject: Arver fra 'Subject'. Det er den konkrete klasse, som skal overvåges af observers
Observer: Interface, som skal implementeres af alle klasse, som ønsker kan få ændringer af vide.
ConcreteObserver: Implementer interfacet 'Observer'. Det er den konkrete klasse som modtager beskeder.

Eksempel: Avis, der laves mange eksemplarer af den samme avis, som sendes ud til alle som abonnerer.

- Udgiver (subjekt) har et abstrakt forhold til abonnenterne (observerne).
- Udgiver (subjekt) kender kun navn/email/adresse
- Abonnenterne (observers) modtager kun avisen fordi de har henvendt sig til udgiveren.

Push og pull variant: Der er to teknikker til at opnå viden om den konkrete tilstandsændring (update)
Push:

- Her "pusher" subjekt de ønskede informationer til alle registrerede observers.
- Observers får den data de behøver, når der sker en ændring i subject.

Pull:

- Subjekt ved ikke hvilken observer som behøver hvilen tilstand
- Publisher notificerer blot observerne at tilstanden er ændret.
- Observers egen opgave at hente de ønskede informationer.
- Denne version anvendes mest, så der ikke sker unødvendige ændringer og meddelelser fra subjekt.

Mange observers til mange subjekter:

- Her skal det også gøres opmærksom på hvilket subjekt ændringen er sket.
    - Skal sendes et id

Fordele og ulemper ved observer pattern:

Fordele:

- Lav kobling, dette fremmer et godt software design
    - Gør det muligt for klasser at reagere på ændringer i andre klasser → uden høj kobling
    - Fleksibelt
- Let at udvide, dette fremmer et godt software design
- Subjekt skal ikke vide hvor mange observer den har, eller ret mange informationer om denne

Ulemper:

- Kan tilføje unødvendig kompleksitet til koden
- Observers kan få meget unødvendigt information ved push varianten.

Hvornår skal det anvendes:

Når flere objekter er afhængige af tilstanden af et objekt.

- Events er baseret på observer pattern
- Sociale medier
- Email subscription (følge mediet og modtage nyeste nyheder)
- Brugeren af en App får en besked når en ny opdatering udkommer.

SOLID principper:

- Open-Closed Principle (OCP): Let at tilføje nye observers og subjekt uden at ændre i subjekt eller anden kode.
- (LSP): De klasser som arver har samme funktionaliteter
- (DIP): Kommunikerer igennem interfaces

3 FACTORY PATTERNS

Hvad er et software design pattern?

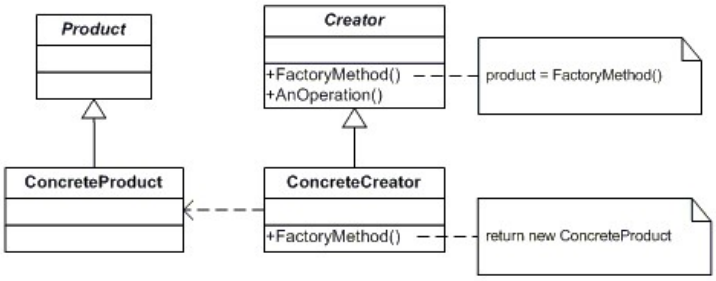
- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

Factory Patterns hører under kategorien *Creational Pattern,* disse har ansvar for effektivt at lave objekter. Den generelle ide omkring fabriker er at opdele udvikling af et objekt fra brugen.

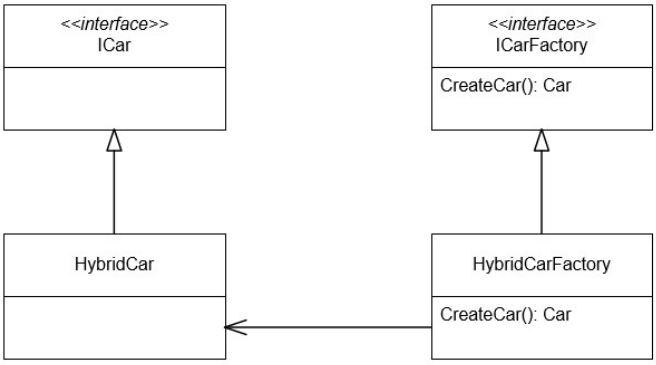
Factory Method Pattern:

Definer et interface til at oprette et objekt, men lader underklasserne bestemme hvilken klasse om skal instantieres. Dette mønster lader en klasse udsætte instantieringen til underklasserne.

- Lave en ”wrapper” omkring new
- Anvendes når vi ikke ved hvad vi har brug for/skal lave.
- Bestemt måde at lave noget på
    - Hvis vi har to ConcreteCreators → har vi to måder at lave vores product på.



- **ConcreteCreator:** Implementerer FactoryMethod(), som er metoden som producer produktet.
- **ConcreteProduct:** Er de faktiske produkter.
- **Creator:** Erklærer en FactoryMethod() som returner et produkt
- **Product:** Erklærer et interface for alle produkter som kan laves.

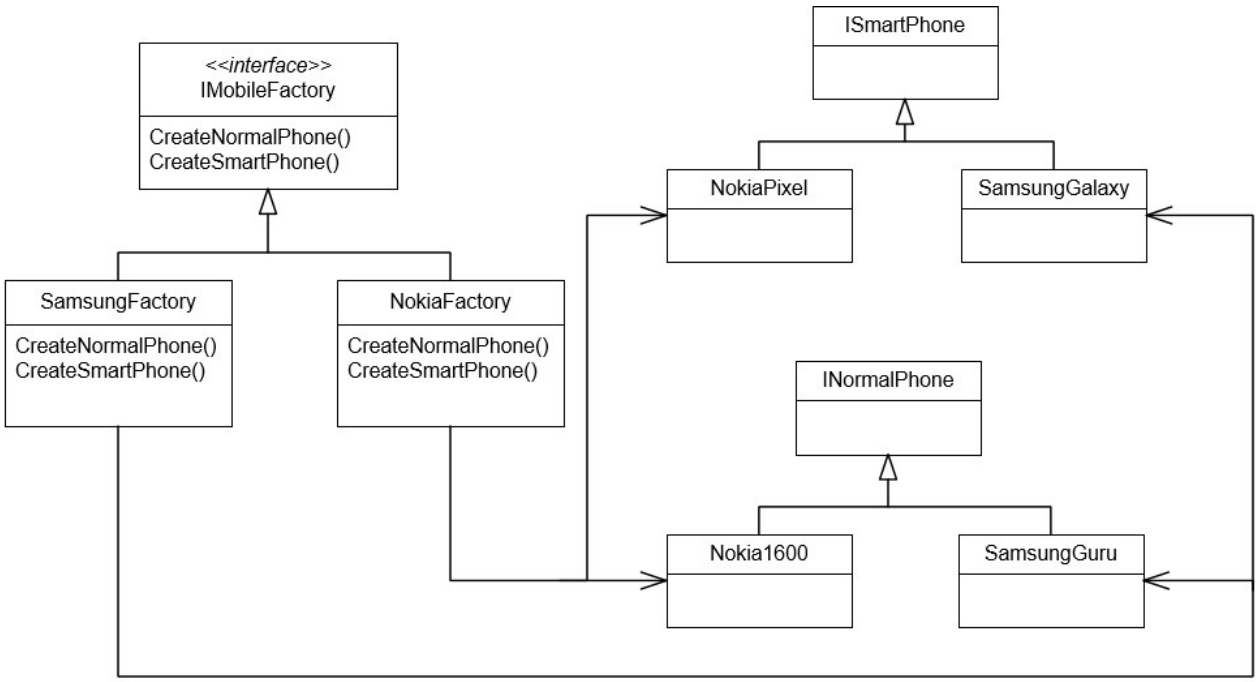


SOLID principper:

- Open-Closed Principle (OCP): Nemt tilføje flere produkter, eller måder at lave disse
- Dependency Inversion Principle (DIP): Depend upon abstractions. Do not depend upon concrete classes

Abstract Factory Pattern:
Definerer et interface til at oprette gruppe af beslægtede eller afhængige genstande uden at specificere deres konkrete klasse.

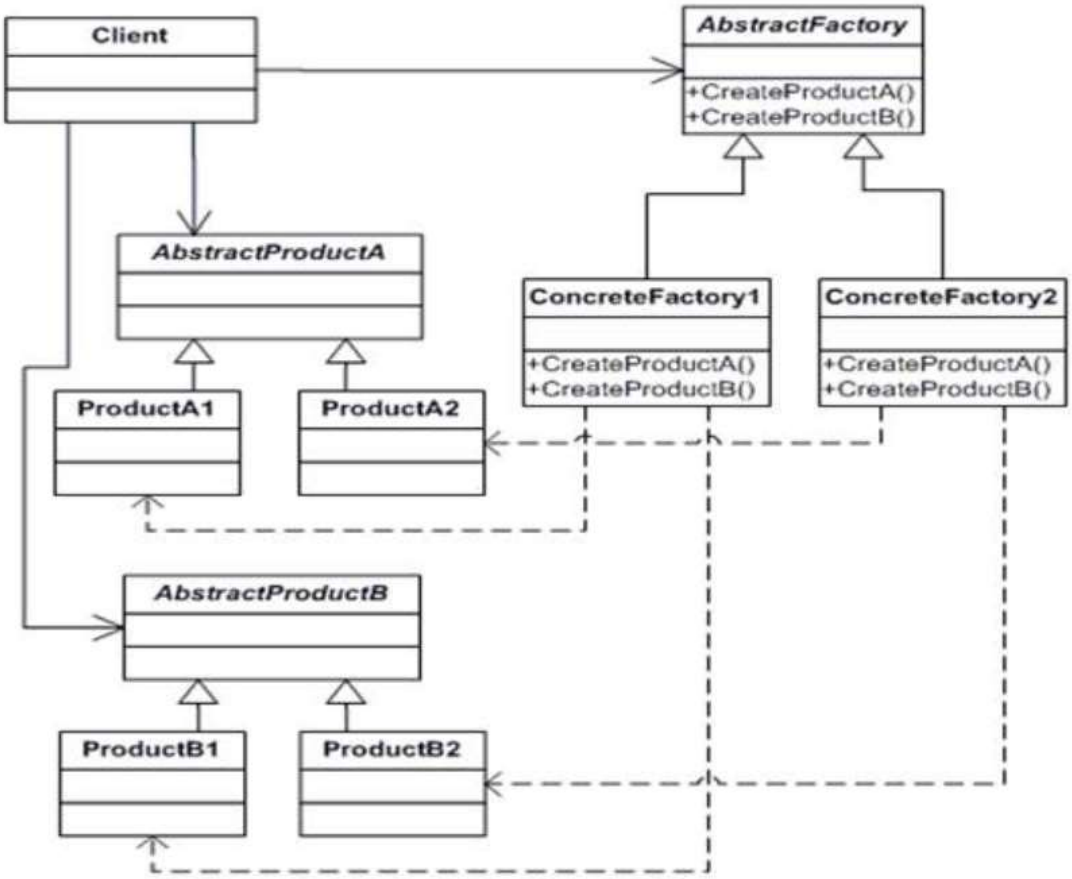
- Interface åbner mulighed for flere metoder, som hver kan lave et objekt (product)
- Minder om Factory Method Pattern, her kan der dog laves grupper af objekter (produkter)



- **AbstrackFactory:** IMobileFactory
- **ConcreteFactory**: Nokia, Samsung (fx Nokia laver en gruppe Nokia produkter)
- **AbstractProduct:** ISmartPhone, INormalPhone (Produkt gruppen – hver konkret fabrik kan lave set af produkter)
- **ConcreteProduct**: NokiaPixel, Nokia1600, SamsungGalaxym SamsungGuru

SOLID principper:

- Open-Closed Principle (OCP): Nemt tilføje flere produkter, eller måder at lave disse
- Dependency Inversion Principle (DIP): Depend upon abstractions. Do not depend upon concrete classes


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

-

4 STATE MACHINE PATTERNS

Hvad er et software design pattern?

- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

State Machine: Beskriver tilstande og overgange i et system

- Befinder sig altid i et af de begrænsede antal stadier.

Disse kan implementeres på 3 måder: Switch-Case, State/event Table og State Pattern.

Når STM bliver komplekse, vil en switch-case eller tabel implementering blive vanskelig, svær at teste og svær at vedligeholde (tilføjer ny tilstand?!) → Derfor skal State Machine Patterns anvendes

Switch-Case:

- Cases repræsenterer forskellige tilstande, og switches på nuværende tilstand.
- Ok til simpel STM

Tabel: Kan give et overblik

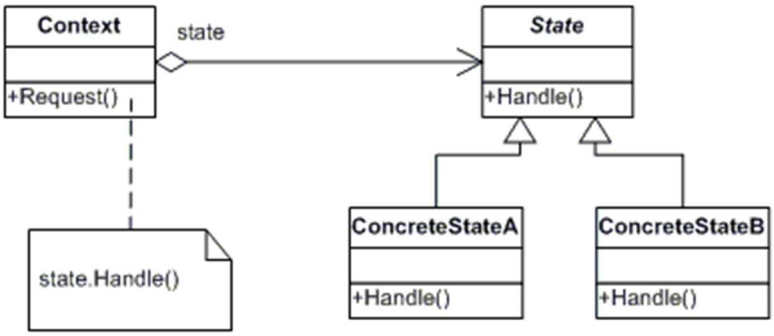
- Når state og event passer sammen udføres en overgang

State Pattern:

State Pattern er et Behavior Pattern, og er en metode til at implementere State Machines (Tilstande)

Opbygning:

- Tillader et objekt at ændre dens opførsel, når den interne tilstand ændres.
    - Dette gøres uden switch cases og if-statements



- **Context klassen**
    - ”Ejeren” af State Machinen.
    - Har en instans af ConcreteState, der kan definere nuværende tilstand
- **State klassen**
    - Et fælles interface som alle konkrete tilstande skal implementere
- **ConcreteState**
    - Håndtere forespørgsler fra Context
    - Hver konkret tilstand får sin egen implementering for en forespørgsel

Fordele:

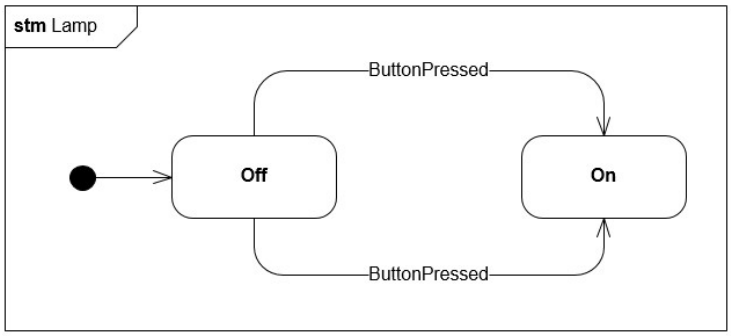
- Let at teste på grund af mange klasser (uddeling af ansvar)
- Let at tilføje nye tilstande
- Eliminerer brugen af swith cases og if-statements
- Lettere at vedlige holde

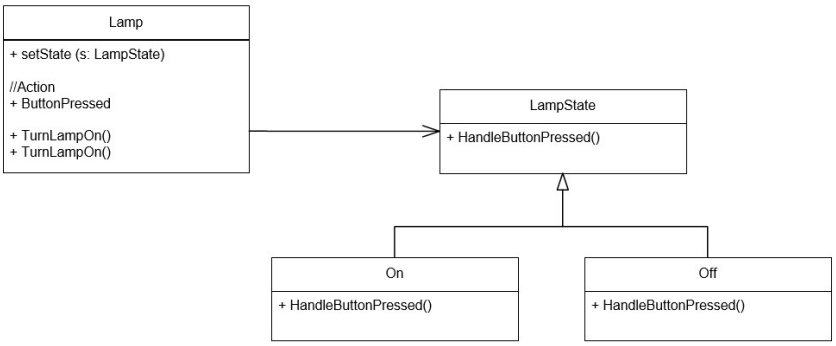
Ulemper

- Skal oprettes mange klasser (derfor anvendes ikke til små simple STM)

Eksempel:

En Lampe kan befinde sig i to tilstande, On og Off.





onExit(): Ting der sker når man går ud at en tilstand
onEnter): Ting der sker når man går ind i en tilstand

SOLID principper:

- Open-Closed Principle (OCP): Let at tilføje nye tilstande
- Single Responsible Principle (SRP): Ikke overholdt pga. Context klasse.
- Liskov Substitution principle (LSP): Subklasser må ikke gøre andet end superklasser
- Interface Segregation (ISP): Aldrig være afhængig af et interface som ikke bruges.
- Dependency Inversion Principle (DIP): OCntext klassen er afhængig af en abstraktklasse
    - Implementer til interface/abstraktklasse og ikke konkrete objekter

5 GUI DESIGN PATTERNS

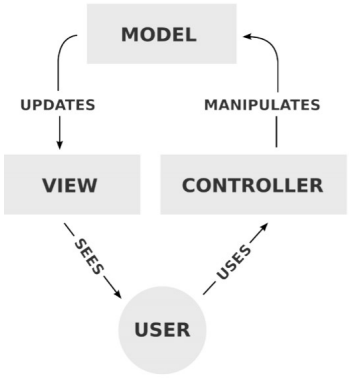
Hvad er et software design pattern?

- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

GUI patterns anvendes for at opdele koden i mindre og mere overskuelige dele.

MVC (Model-View-Controller):

- Software pattern som bruges til at implementerer GUI applikationer.
- Fortæller hvor forskellige kodestykker skal være
    - Opdeler applikation funktionalitet i 3 elementer; Model, View og Controller.



- **Model:** Indeholder logik, applikation data og tilstande for programmet.
    - Giver besked til view om at den har skiftet stadie.
- **View:** Indeholder præsentationslogik (Det eneste som brugeren ser og interagerer med)
    - Fortæller controller om user input
- **Controller:** Bindeledet mellem Model og View
    - Indeholder logik til at håndtere brugerens aktion med viewet
    - Kan håndtere information fra en database, og snakke med denne
    - Modtage informationer fra en database

Eksempel: Bruger benytter interface og handlinger sendes til controller

Fordele:

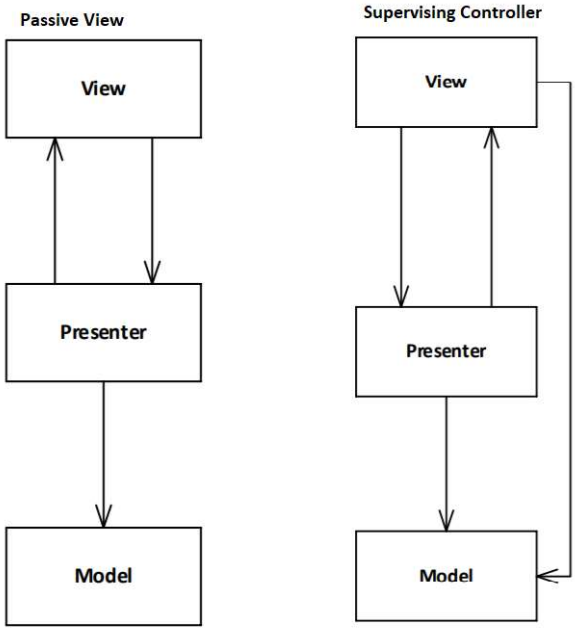
- Høj separation mellem præsentationen (controller og View) og domæneobjektet (modellen)
    - Overholder SRP ved denne separation
- Modellen (domæneobjektet) kan i mange tilfælde genbruges
- Lettere at teste og vedligeholde
- Organiseret måde at uddele kode.

Ulemper:

- Ikke god til mindre applikationer → gør det for komplekst

MVP (Model-View-Presenter):

- Software pattern som bruges til at implementerer GUI applikationer.
- Findes i to varianter; Passive View og Supervising Controller.
- Opdeler ligeledes funktionalitet i 3 dele; View, Presenter og Model



- **Model:** Indeholder det data som skal vises på skærmen.
- **View:** Viser modellen på skærmen (præsentationslogik)
- **Presenter:** Binder modellen til Viewet
    - Logik som anvendes til behandling af user inputs.
    - Hente data fra modellen → behandle det således det kan sendes til View
    - Ansvarlig for største delen af opdateringer til modellen

Variationer Passsive og Supervisning: Hvordan view opdateres

Passive View:

- View har ingen direkte association til modellen
    - View viser data på skærmen
        - Overgiver alle user input til presenter
- Presenter indeholder GUI logik → opdater model og view

Supervisning View:

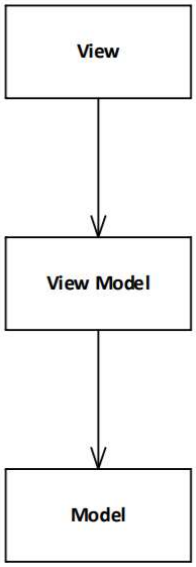
- View kan kommunikere direkte med modellen og forespørge data
- View bruger data binding eller observer synkronisering til at få data from modellen
- Mere komplekse relationer håndteres af presenter

Forskellige mellem MVC og MVP

1. I MVP går alle bruger interaktioner gennem view
2. I MVC skal controlleren håndtere alle user input, og opdatering er kun en af controllerens ansvarI MVP er presenters hovedansvar at opdatere modellen

MVVM(Model-View-ViewModel):

- Software pattern som bruges til at implementerer GUI applikationer.
- En specialisering af MVP lavet af Microsoft
- Opdeler ligeledes funktionalitet i 3 dele; View, ViewModel og Model
- **Struktur:**
    - View er forbundet til ViewModel gennem data binding
        - ViewModel kender ikke til view
    - ViewModel snakker med modellen gennem properties, metodekald
        - Modellen kender ikke til ViewModel



- **Model:** Indeholder den faktiske data og/eller den information som programmet skal kunne håndtere
    - Indeholder udelukkende informationer (ikke adfærd eller services som kan manipulere data)
- **View:** Den eneste del som brugeren ser og interagerer med
    - Data fra modellen præsenteres og formateres på en præsentabel måde
- **ViewModel:** Er en model af Viewet
    - Fungerer som bindeled mellem Model og View

Hvorfor anvende MVVM?

- Uddeling af ansvar
    - UI-design→ View
    - Præsentation logik → ViewModel
    - Business logik → Model
- Lettere at teste
- Lav kobling
- Godt work flow (samarbejde)
    - Designer skriver View
    - Udvikler skriver ViewModel

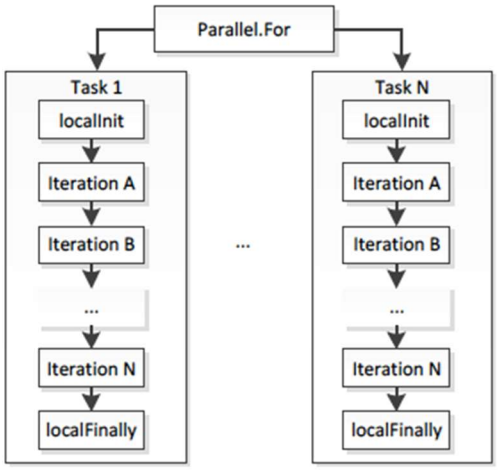
6 PARALLEL AGGREGATION + MAPREDUCE

Concurrency patterns er de mønstre som håndterer multi-threaded programmering.

I parallelle systemer, uddeles arbejde og behandlet parallelt, delresultater kombineres for at opnå et endeligt resultat.

Parallel Aggregation:
- Langvarige og tidskrævende udregninger eller processor → som skal køres ønskes at opdeles
- Eksekveringen opdeles mellem tråde → for at øge hastigheden
- Vigtigt at kende endelige resultat for at kunne sammenligne resultatet med den aggregated version.

Parallel Aggregation pattern anvender
- Unshared lokale variabler som sammensættes til sidst for give det endelige resultat
    - Kombinerer flere inputs til et enkelt output



- Et delresultat kan udregnes i hver Task
- I slutningen kombineres alle disse delresultater til et samlet

Parallel Aggregation kan anvendes hvis følgende betingelser er opfyldt
1. Opgaven kan opdeles i flere uafhængige dele → som hver producere et delresultat
2. Resultatet fra den ene del er ikke afhængig af resultatet fra den anden del (delresultater må ikke være afhængige af hinanden)
3. Sættene af delresultater fra hver Task kan kombineres efter hver Task er færdig med opgaven
4. Kommunikation mellem Tasks skal være så lille som mulig.

Eksempel:
Adder alle tal fra 0 til 1.000.000.000 (Milliard)
- Hver Task kan udregne en del af den store udregning
- Til sidst kombineres alle Task delresultater til det endelige resultat

Divide and Conquer!

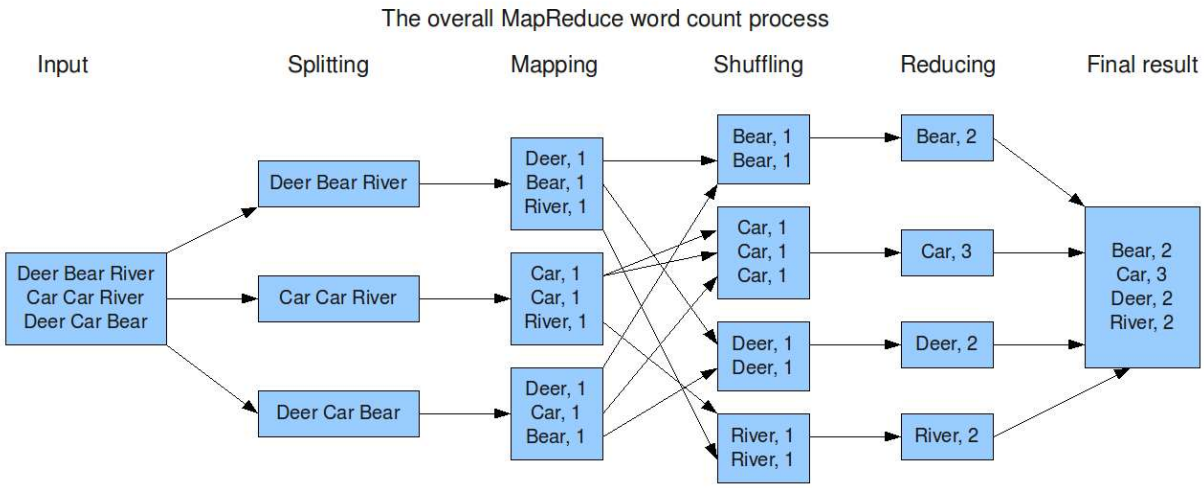
MapReduce:

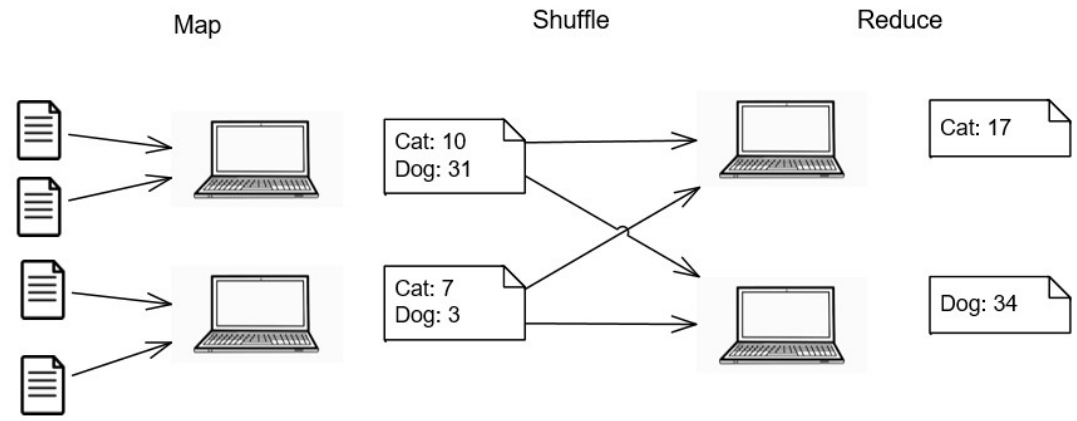
- Dette er en metode til at bearbejde store datasæt.
- Der arbejdes parallelt med data → således effektiviteten øges

Ved denne teknik tvinges man til at opdele "det man gerne vil opnå" ind i 3 dele; Map, Shuffle og Reduce.

- Map: Arbejdet med at splittes data/input
- Shuffle: Processen i at udveksle data fra Maps til der hvor de skal anvendes
- Reduce: Tager resultater fra 'Mapper' og kombiner dem, for at få det endelig resultat.

Eksempel: Tal antal gange et ord optræder i et dokument:





- Data uddeles til forskellige computer (nodes) → repræsenteres i key/value par
  - Hver Computer fortæller antal af ord den har fundet (Key = ord, value antal)
- Data udveksles → et bestemt ord på en bestem computer
- Data reduceres → Her optælles det endelige antal ord i dokumenter

7 FUTURES + PIPELINES

Concurrency patterns er de mønstre som håndterer multi-threaded programmering.

I parallelle systemer, uddeles arbejde og behandlet parallelt, delresultater kombineres for at opnå et endeligt resultat.

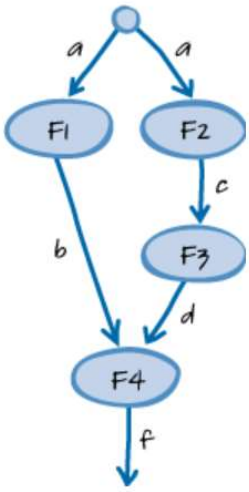
Future:

- Er et objekt som repræsenterer outputtet af en udregning som endnu ikke kendes.
- En future kan ses som en værdi der til sidst vil blive tilgængelig
- Når man prøver at tilgå en future vil programmet vente til udregning bliver færdig.
    - Når resultatet bliver tilgængeligt vil noget andet kode blive kørt.
    - Beregningen af denne værdi kan ske parallelt med andre beregninger
- Future implementering C#
    - Implementeres med Task<T> biblioteket
- 3 outputs:
    - Er tasken færdig: Værdien returneres øjeblikkeligt
    - Tasken er i gang med at blive udført: Tråden der spørger efter værdien blokeres indtil værdien er tilgængelig
    - Tasken er endnu ikke påbegyndt: Tråden der spørger efter værdien udfører tasken – hvis det er muligt.

Eksempel:

- Ønsker at uddele arbejdet af disse funktioner ud til flere cores
- Der laves en graf over koden

```
static void Main(string[] args)
{
    var a = "A";
    var b = F1(a);
    var c = F2(a);
    var d = F3(c);
    var f = F4(b, d);
    System.Console.WriteLine(f);
}
```



Indgående pile: Nødvendige inputs for funktion

Udgående pile: Værdier udregnet af hver funktion

F1 og F2 kan køre på samme tid
- men F3 skal komme efter F2

Kigger på input og output:

```
static void Main()
{
    var a = "A";
    Task<string> futureB = Task.Run(() => F1(a));
    var c = F2(a);
    var d = F3(c);
    var f = F4(futureB.Result, d);
    Console.WriteLine(f);
}
```

Continuation Tasks (Det vi har i .NET)

- Når resultatet af en funktion foreligger skal der automatisk kaldes en anden funktion
- Kunne være en funktion der opdaterede brugergrænsefladen
- Brug ContinueWithAll venter alle
- ContinueWith venter 1
- ContinueWhenAny venter til en af dem er færdige.

Fordele:

- Hurtigere eksekvering
- Kræver mindre synkronisering

Ulempe:

- Langsomste funktion bestemmer hastighed på den samlede udførselstid.

Pipeline Pattern:

- Dette mønster gør en inputsekvens parallel og der hvor rækkefølge betyder noget
- Kan ses som et samlebånd
    - Hver genstand er konstrueret i stadier
    - Den delvist samlede genstand går fra et stadie til et andet
    - Outputtet af samlebåndet optræder i den samme orden som de blev modtaget
    - Tager et objekt → modificerer det → giver det videre til den næste klasse
- Bruges når data løber igennem en sekvens af opgaver eller stadier
- Alle stadier er afhængige af outputtet af det tidligere stadige

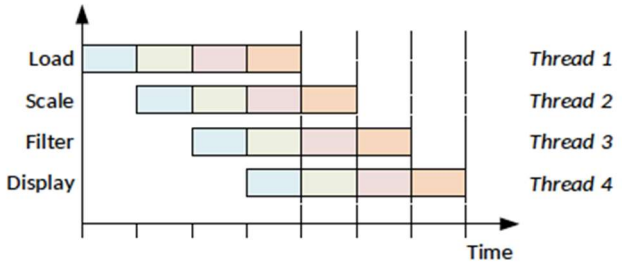
- Futures: Et stykke data
- Pipeline: Mange stykker data
- Køer mellem hvert stadie:
    - BlockingCollection
        - Blokerer så der ikke kan tages ud hvis den er tom
        - Blokerer så der ikke kan smides i hvis den er fuld
    - Producer/consumer problem
        - Consumer tager noget → sender notifikation til producer
        - Consumer sover hvis køen er tom
            - Producer sender notifikation.

Eksempel:

- Hvert billede skal bearbejdes i 4 stadier:
- 1. Billedet loades fra en fil
- 2. Skaler billede
- 3. Filter på billede
- 4. Bearbejdet billede vises som næste billede

Load → scale → filter → display

Resultatet af pipeline-mønsteret kan ses herunder



Fordele: Fremskynde processor som har flere stager
Ulemper: Fremskynde dog ikke den enkelte proces, det gøres blot mere effektivt

8 SOFTWARE ARCHITECTURE

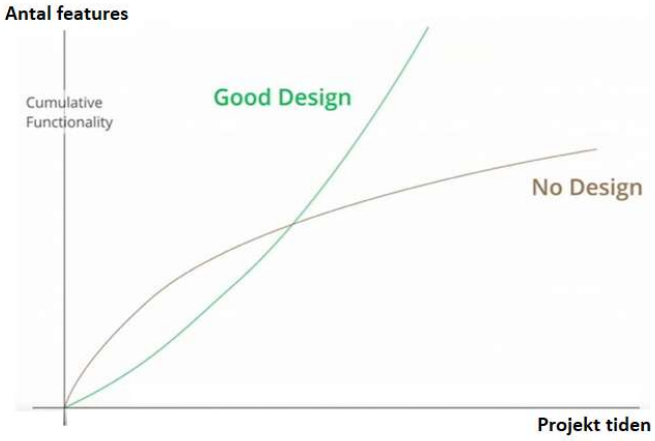
Hvad er software arkitektur?
Et sæt af nødvendige strukturer, der er påkrævet for at fremme et systems eksterne komponenter, relationen mellem dem, samt egenskaber for både relationer og komponenterne.

Hvad:
Processen hvor der defineres en struktureret løsning der imødekommer tekniske og operationelle krav.

- Når beslutninger først er taget er de rigtig svære og dyre at lave om.
- SW arkitektur er systemets skelet
- Software arkitektur er den højeste abstraktion i et system
- Målet: Identificer kravene som har en (størst) effekt på strukturen af applikationen
- Planlagt produkt → før implementering (opnå viden om system) → tidlige beslutninger
- Struktur: Hvad hører sammen og hvem er afhængige af hinanden.
- Adfærd: Flow mellem moduler og gennem software
- SOLID: På moduler. Lav kobling – høj sammenhørlighed

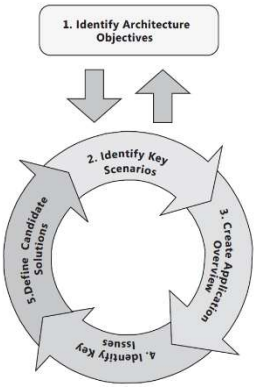
Hvorfor (er det vigtigt)?

- Vigtigt at software bygges på et solidt fundament
- Uden design bliver det mere og mere tidskrævende at tilføje ny funktionalitet
    - Betaler for bedre intern kvalitet → kan nye features på en senere tidspunkt tilføjes hurtigere



- Hvis man overser (glemmer) (huske at tage stillingen)
    - at tage hensyn til hovedscenarier
    - at designe mod typiske problemer
    - håndtere langtidsudsigten
- Dårlig arkitektur:
    - Ustabil software
    - Software som ikke støtter eksisterende eller fremtidige krav
- Kan man nå deadline, pris og mandetimer
- Når systemet bliver større → bliver arkitektur mere kompleks og abstrakt → mere vigtig.

Hvordan:



- Iterativ arkitektur proces med følgende trin
    - 1. Identify architecture Objectives
        - Mål → hjælper med at løse konkrete problemer i designet
        - Identificer begrænsninger (sikkerhed, pålidelighed, skalerbarhed, performence)
    - 2. Identify Key Scenarios (Hovedscenariet):
        - Fokuser design → hvad betyder mest
        - Hvilke mål er mest kritiske
    - 3. Create application Overview (overblik):
        - Identificer
            - Applikationstype: web, app, mobile (platforme)
            - Hvordan deployer systemet
            - Teknologier (protokoller, sprog)
    - 4. Identify Key Issues (Hovedproblem stillinger): Hvordan opnår vi disse begrænsinger
        - System kvalitet: Generelle kvalitet for systemet
        - Run-time kvalitet: Performance
        - Designkvalitet: Fleksibilitet
        - Brugerkvalitet: Hvor nemt er systemet at bruge.
    - 5. Define Canditate solution
        - Skab en prototype → udvikler og forbedre løsningen, evaluerer tidligere opstillet ting
        - Dette før næste gennemgang af processen
        - Test som viser at delen er blevet optimeret i processen.
- User/business/System
    - Alle har forskellige ideer omkring hvordan produktet skal virke.

Arkitektur stil:
Ligesom der findes forskellige design pattern, findes der ligeledes forskellige mønster af arkitektur.

Layerd: Strukturer et program og deles i mindre under grupper. Altså lag
- Højere lag kalder funktioner i lavere lag
- Lavere lag må ikke kalde funktioner i højere lag
    - Presentation layer
    - Applikation layer
    - Business Logic Layer
    - Data Access Layer

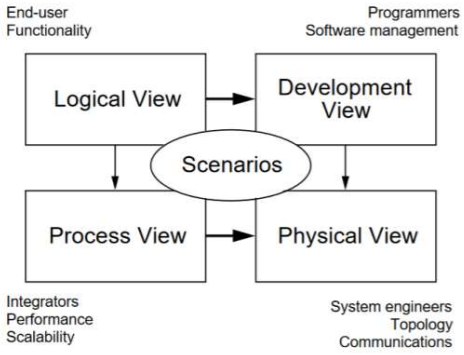
Client/server: Består af en server og flere klienter
- Serveren vil give services til flere klienter
- Client efterspørger services fra serveren.

Dokumentation af arkitektur

4+1 view, er en arkitekturmodel, som beskriver systemet baseret på flere samtidige views → vinkler.

Views bruges til at beskrive fra forskellige vinkler fra forskellige parter.

- Logical View: Funktionaliteten af systemet (end-user)
    - Klassediagrammer + tilstandsdiagrammer
- Development View: Vises fra en programmørs vinkel
- Process View: Håndterer de dynamiske aspekter af systemet.
    - System processor og hvordan de kommunikerer
    - Aktivitets diagrammer
- Physical View: Vises fra en systemingeniørs perspektiv
- Scenarios: Beskrivelsen af arkitekturen vises med en række små Use-Cases
    - Sekvens af interaktioner mellem objekter og processor.



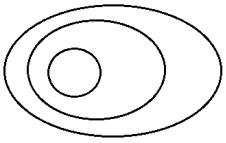
9 MANDATORY EXERCISE (DECORATOR PATTERN)

Hvad er et software design pattern?

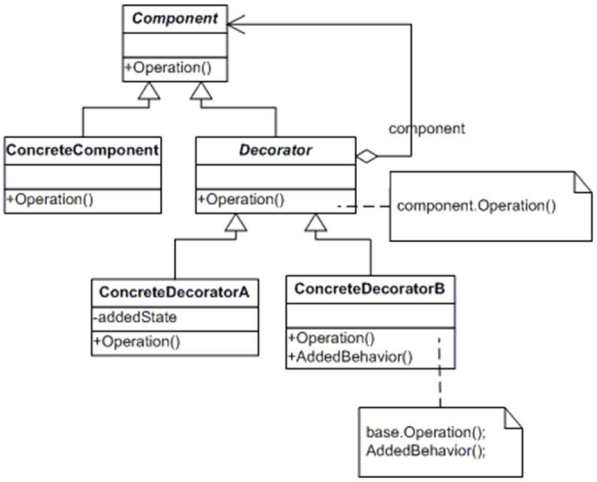
- Designmønstre er en generel løsning på en problemtype, der ofte opstår i softwareudvikling.
- Ikke et endeligt design, det er en skabelon for hvordan man løser et problem i mange forskellige situationer
- Er ikke et færdigt design som kan laves direkte om til kode.

Decorator pattern:

- Defineret som et strukturelt pattern → fordi den pakker et eksisterende objekt ind i lag af en abstraktion.

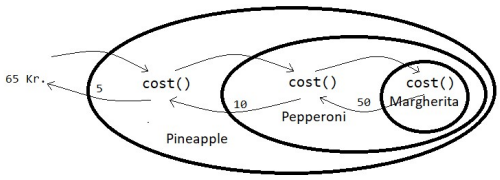


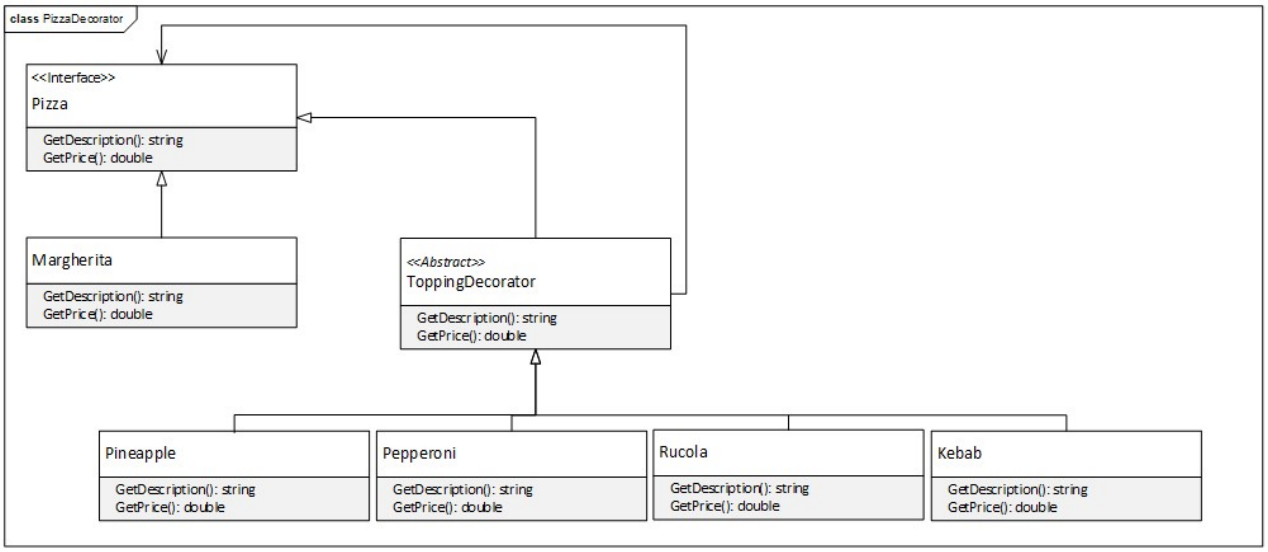
- Giver bruger mulighed for at tilføje ny funktionalitet og adfærd til allerede eksisterende obejkt.
    - Dette sker uden at modificere objektets struktur
- Lader en bruger tilføje en eller flere opførsler til et objekt run-time.



- **Component:** Interface som angiver hvilket produkt vi ønsker at dekorerer
- **ConreteComponent**: Som er det konkrete product som brugeren dekorer
- **Decorator:** Abstrakt klasse, som angiver det vi kan ”pynte” vores produkt med
- **ConcreteDecorator**: Er det konkrete genstande som kan pynte det konkrete produkt.

Eksempel: Ønsker at vælge forskellige toppings til en pizza.





Fordele og ulemper:

Fordele:

- Mere fleksibelt end arv
- Tillader at tilføje funktionalitet fun-time
- Tillader mange forskellige kombinationer (kan tilbage man decorators i forskellige mængder)

Ulemper:

- Mange små klasser med lidt kode (meget at vedlige holde

SOLID principper:

- Single Responsible Principle (SRP): Ekstra funktionalitet eller adfærd tilføjes i ny klasse
    - Hver klasse har kun et ansvar.
- Open-Closed Principle (OCP): Let at tilføje nye produkter, eller decorators uden at ændre den eksisterende kode.

10 SOLID

Single Responsible Principle (SRP):
"A class should only have one reason to change, meaning that a class should only have on job."
- En klasse skal kun have en grund til at ændre sig
- Det vil sige klasser må kun have et ansvar (et job)

Open-Closed Principle (OCP):
"Objects should be open for extension, but closed for modification"
- Det skal være let at udvide en klasse, uden at ændre klassen selv.

- Open: En klasses opførsel kan udvides hvis/når krav ændres.
- Closed: Sourcekoden for denne klasse må ikke ændres.

Opnås gennem abstraktion

Liskov Substitution principle (LSP):
"Derived classes must be substitutable for their base classes"

Interface Segregation (ISP):
"Make fine grained interfaces that are client specifik"
- Klienter skal ikke tvinges til at implementerer som de ikke bruger

Det er bedre at have flere små interfaces, en et stort.

Dependency Inversion Principle (DIP):
"Depend on abstractions, not concretions"

Højniveau moduler skal ikke være afhængig af lavniveau moduler, begge skal være afhængige af abstraktioner
Abstraktion skal ikke være afhængige af detaljer. Detaljer skal være afhængige af abstraktioner.