

UNIVERSITY OF AARHUS

Faculty of Science

Department of Engineering

**Eksamensdispositioner
Software Design**

Bjørn Nørgaard
IKT
201370248
bjornnorgaard@post.au.dk

Joachim Andersen
IKT
20137032
joachimdam@post.au.dk

Sidste ændring: January 29, 2016 at 11:36

L^AT_EX-koden kan findes [her](#)

<https://github.com/BjornNorgaard/I4SWD/tree/bjorn/Eksamen/Disposition>

Todo list

uløst problem med OnExit() Andreas/Denned??	26
GUIPERSON: what is dis?	38
GUIPERSON: hvordan sender den events?	38
er databinding ikke ét ord?	39
hvordan kan noget være til nytte hvis det ikke er muligt?	39
en ordforklaring til dette afsnit ville være nice til os uden GUI.	39
”” antyder indforstået betydning, kender vi andre ikke noget til.	39
forstår ikke helt linje 14 i det nedenstående..	43

Indholdsfortegnelse

1 Solid 1 - SRP, ISP og DIP	1
1.1 Fokuspunkter	1
1.2 Single Responsibility Principle (SRP)	1
1.2.1 Brud på SRP	1
1.3 Interface Segregation Principle (ISP)	1
1.4 Dependency Inversion Principle (DIP)	3
1.5 Hvordan fremmes godt SW design?	4
1.5.1 SRP	4
1.5.2 ISP	4
1.5.3 DIP	5
1.6 Eksempel	5
1.7 Redegør for ulemper	5
2 Solid 2 - OCP, LSP og DIP	6
2.1 Fokuspunkter	6
2.2 Open-Closed Principle (OCP)	6
2.2.1 HOW TO OCP	6
2.2.2 Perspektiver	7
2.3 Liskov's Substitution Principle (LSP)	7
2.3.1 Pre -og postconditions	7
2.3.2 LSP overholdt	7
2.3.3 Brud på LSP	8
2.4 Dependency Inversion Principle (DIP)	8
2.5 Hvordan fremmes godt SW design?	9
2.6 Eksempel	10
2.7 Redegør for ulemper	10
3 Patterns 1 - GoF Strategy + GoF Template Method	11
3.1 Fokuspunkter	11
3.2 Hvad er et software pattern?	11
3.3 Sammenlign de to design patterns GoF Strategy og GoF Template Method - hvornår vil du anvende hvilket, og hvorfor?	11
3.3.1 GoF Strategy Pattern	11
3.3.2 GoF Template	11
3.4 Vis et designeksempel på anvendelsen af GoF Strategy	12
3.5 Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Strategy understøtter	13
3.6 Redegør for, hvordan anvendelsen af GoF Template fremmer godt SW design	13
3.6.1 Eksempel på Template pattern	14

3.6.2	Hvornår vil du anvende hvilket, og hvorfor?	15
4	Patterns 2 - GoF Observer	16
4.1	Fokuspunkter	16
4.2	Hvad er et software pattern?	16
4.3	Redegør for opbygningen af GoF Observer	16
4.4	Sammenlign de forskellige varianter, af GoF Observer - hvilken vil du anvende hvornår?	17
4.4.1	Pull	17
4.4.2	Push	17
4.4.3	Many observers to many subjects	17
4.5	Who triggers the update?	17
4.6	Redegør for fordele og ulemper ved anvendelsen af GoF Observer	18
4.6.1	Fordele	18
4.6.2	Ulemper	18
4.6.3	Making sure the subject state is self-consistent	18
4.7	Redegør for, hvordan anvendelsen af GoF Observer fremmer godt software design	18
4.8	Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Observer undersøger	18
5	Patterns 3 - GoF Factory Method/Abstarct Factory	19
5.1	Fokuspunkter	19
5.2	Hvad er et software pattern?	19
5.3	Redegør for opbygningen af GoF Factory Method	19
5.4	Redegør for opbygningen af GoF Abstract Factory	20
5.4.1	Problem	20
5.4.2	Abstract Factory	21
5.5	Giv et designeksempel på anvendelsen af GoF Abstract Factory	21
6	Patterns 4 - State patterns	23
6.1	Fokuspunkter	23
6.2	Hvad er et software pattern?	23
6.3	Redegør for strukturen i et state pattern	23
6.3.1	Kodeeksempel	24
6.4	Sammenlign switch/case-implementering med GoF State.	25
6.4.1	Switch case implementering af STM	25
6.4.2	GoF State implementering af STM	26
6.5	Redegør for fordele og ulemper ved anvendelsen af GoF State.	26
6.5.1	Fordele	26
6.5.2	Ulemper	26
6.6	Redegør for, hvordan et UML (SysML) state machine diagram mapper til GoF State	26
6.6.1	Simpel UML	26
6.6.2	Nestede States	27
6.6.3	Ortogonal States	28
7	Patterns 5 - Model View Controller og Model View Presenter	31
7.1	Fokuspunkter	31
7.2	Hvad er et software pattern?	31
7.3	Redegør for Model-View-Control mønstret og dets variationer	31
7.3.1	MVC's opbygning	32
7.3.2	Passive model	32
7.3.3	Active model	33
7.3.4	Fordele ved MVC	34

7.4	Redegør for Model-View-Presenter mønstret og dets variationer	35
7.4.1	MVP's opbygning	35
7.4.2	Fordele ved MVP	36
7.4.3	Variationer af MVP	36
8	Patterns 6 - Model-View-ViewModel (MVVM)	37
8.1	Fokuspunkter	37
8.2	Hvad er et software pattern?	37
8.3	Redegør for Model-View-ViewModel mønstret og dets variationer	37
8.3.1	View	37
8.3.2	ViewModel	38
8.3.3	Model	38
8.3.4	Commands	39
8.3.5	Oprettelse af Views og ViewModels	39
8.3.6	Hvorfor er det smart?	40
9	Patterns 7 - Redegør for følgende concurrency mønstre	41
9.1	Fokuspunkter	41
9.2	Task Parallel Library	41
9.3	Parallel loops	41
9.3.1	Hvornår giver det mening med Parallel Loops?	41
9.3.2	Oversubscription	41
9.3.3	Partitinerer	42
9.3.4	Kodeeksempler	42
9.4	Parallel tasks	42
9.4.1	Kodeeksempler	43
9.5	Thread Pool	44
9.5.1	Den simpel og dårlige løsning	44
9.5.2	Default Task Scheduler	45
10	Patterns 8 - Redegør for følgende concurrency mønstre	47
10.1	Fokuspunkter	47
10.2	Parallel Aggregation	47
10.2.1	Concurrent Collection in .NET 4.0	47
10.2.2	Eksempel med summering	47
10.3	MapReduce	48
10.3.1	4 steps	48
11	Patterns 9 - Redegør for følgende concurrency mønstre	50
11.1	Fokuspunkter	50
11.2	Future	50
11.2.1	Fordele og ulemper	51
11.3	Pipelines	51
11.3.1	Fordele og ulemper	52
12	Software arkitektur	53
12.1	Fokuspunkter	53
12.2	Redegør for begrebet softwarearkitektur	53
12.2.1	Hvad er en software arkitektur?	53
12.2.2	Hvorfor er det vigtigt?	53
12.2.3	Generelle overvejelser	54
12.2.4	Architectural Styles	54

12.3	Giv et eksempel på en typisk softwarearkitektur og dens anvendelse	55
12.3.1	Layer	55
12.4	Hvordan udarbejdes en software arkitektur	55
12.4.1	Brug af Iterativ process	56
12.5	Hvordan dokumenteres en software arkitektur	58
12.5.1	UML og N+1	59
13	Obligatorisk Opgave (Mediator Pattern)	61
13.1	Valgte fokuspunkter	61
13.2	Hvad er et software pattern?	61
13.3	Redegør for opbygningen af Mediator Pattern	61
13.4	Sammenlign de to Design Patterns GoF Observer og GoF Mediator - hvornår vil du anvende hvilket og hvorfor?	63
13.5	Redegør for fordele og ulemper ved anvendelsen af GoF Mediator	63
13.5.1	Fordele	63
13.5.2	Ulemper	63
13.6	Redegør for hvilke(t) SOLID-princip(er) du mener anvendelsen af GoF Mediator understøtter	63

List of Figures

1	Flere klienter afhængige af samme "store" interface	2
2	ISP anvendt på eksempel fra figur 1	2
3	DIP eksempel 1	3
4	DIP eksempel 2	3
5	OCP brudt.	6
6	DIP eksempel 1	9
7	DIP eksempel 2	9
8	UML for et strategy pattern	12
9	Klassediagram for Template pattern	14
10	GoF Observer klassediagram	17
11	Klassediagram for FactoryMethod	19
12	Klassediagram for compressionstocking systemet.	21
13	UML for et GoF state pattern	24
14	Et UML State Machine Diagram	26
15	Et UML Klassediagram for STD	27
16	Et UML State Machine Diagram for Nested states	27
17	Et UML Klassediagram for Nested states	28
18	Et UML State Machine Diagram for ortogonale states	29
19	Et UML Klassediagram for Nested states	30
20	MVC Passiv variation, eksemplificeret med en musik afspiller.	31
21	Passiv variation af MVC.	32
22	MVC active, observer pattern.	33
23	Aktiv variation af MVC.	34
24	Programmets flow i en MVP applikation	35
25	Variationer af MVP.	36
26	De forskellige GUI patterns og deres variationer	37
27	Sammenhæng mellem Model-View-ViewModel	38
28	Intern adfærd i MVVM pattern	39
29	Simple implementering af threadpool	45
30	Faktisk implementering af threadpool, i .NET.	45

31	Forskellen på seriel og parallel reducereing.	48
32	MapReduce illustreret.	48
33	Node D skal køres før A, B og C kan køres parallelt	50
34	Data dependancies	50
35	Brug af Futures	51
36	Brug af pipelining	51
37	Standard pipelining med uneven stages	52
38	Pipelining med uneven stages + Ekstra stage	52
39	Arkitektur for et system med GUI, BLL og DAB logik.	56
40	Netværksarkitektur, Open Systems Interconnection Model (OSI model).	57
41	Process for iterativ arkitekturudvikling.	58
42	De forskellige N+1 views.	59
43	De forskellige N+1 views.	60
44	Et mediator pattern	62
45	GoF Observer klassediagram	63

1 Solid 1 - SRP, ISP og DIP

1.1 Fokuspunkter

- Redegør for designprincipperne:
 - Single Responsibility Principle (SRP).
 - Interface Segregation Principle (ISP).
 - Dependency Inversion Principle (DIP).
- Redegør for, hvordan du mener anvendelsen af principperne fremmer godt SW design.
- Vis et eksempel på anvendelsen af et eller flere af principperne i SW design.
- Redegør for konsekvenserne ved anvendelsen af principperne - har det nogle ulemper?

1.2 Single Responsibility Principle (SRP)

En klasse skal kun have ét ansvar. Derved undgår vi at skulle *rebuild*, *retest* and *redploy* funktionalitet, som ikke er ændret. På samme tid skal det selvfølgelig heller ikke "overgøres" sådan at vi får *needless complexity*.

"An axis of change is an axis of change, only if changes occur"

"Dont apply SRP if there is no symptom"

Modem eksemplet fra side 118 i bogen¹. Skal vi dele modem klassen op? Det kommer an på hvordan applikationen ændrer sig. Hvis connection-delen ændres skal resten af klassen (data-transfer) også recompile.

1.2.1 Brud på SRP

Brud på SRP kan ses på figur 1 og 2 under ISP i section 1.3. Hvis dette interface har grund til at blive opdelt så er ansvaret muligvis også så anderledes at det burde være i separate klasser.

1.3 Interface Segregation Principle (ISP)

"No client should be forced to depend on methods it doesn't use".

Når vi har flere klienter som alle bruger samme klasse gennem et interface *IDoThings* som det kan ses på figur 1 vil nogle klienter blive afhængige af metoder som de ikke bruger.

¹Bogen til kurset: Agile Principles, Patterns, and Practices in C#

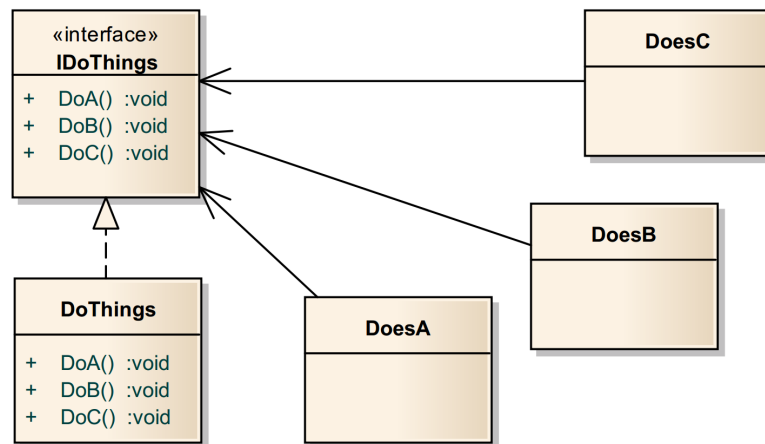


Figure 1: Flere klienter afhængige af samme "store" interface

Derfor kan vi ved hjælp af ISP princippet dele interfacet op i flere mindre interfaces. Således undgår vi store og uoverskuelige interfaces, et eksempel er på figur 2.

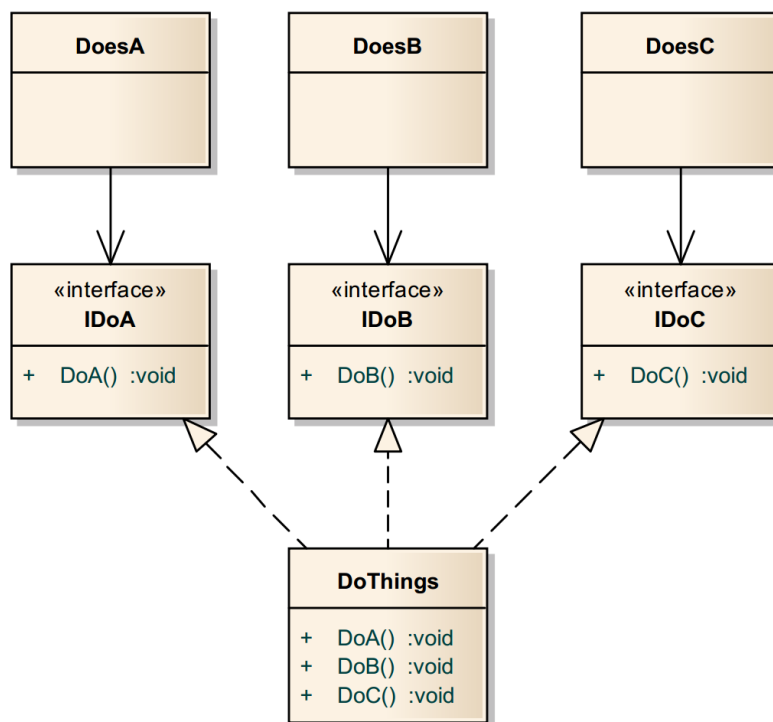


Figure 2: ISP anvendt på eksempel fra figur 1

Et eksempel kan være rectangle eksemplet på [denne](#) side, som også har gode forklaring til de øvrige SOLID principper. Klassen bør ikke indeholde funktion til både `draw()` og `calc()`. Dette gør at fx GUI includen skal bygges samtidig med `calc`.

Endeligt omkring brud på ISP skal section 1.2.1 ses på side 1.

1.4 Dependency Inversion Principle (DIP)

DIP har følgende centrale punkter:

"High level modules should not depend on low level modules. Both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Det hedder dependency inversion fordi vi invertere afhængigheden for klasserne på figurene på side 9. Sådan at afhængigheden fra figur 6 bliver inverteret, som vist på figur 7.

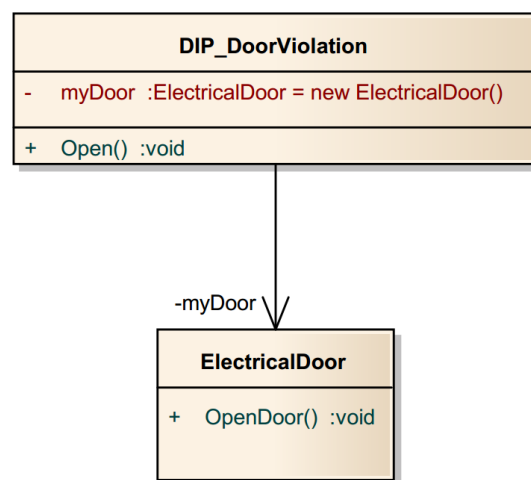


Figure 3: Eksempel på klasser hvor DIP ikke er anvendt.

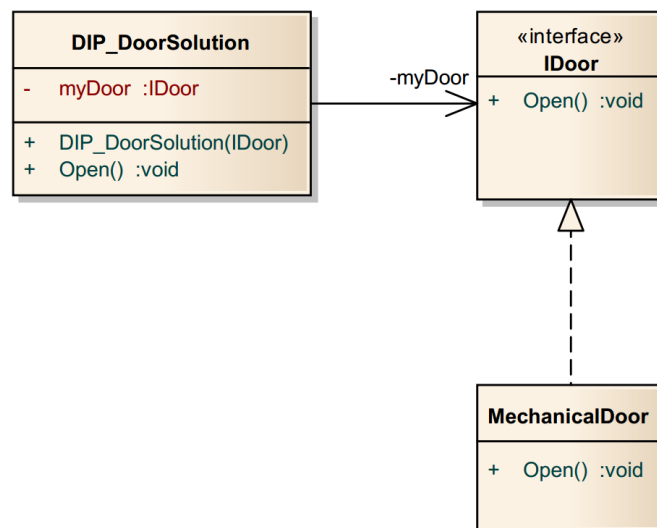


Figure 4: Eksempel på hvor DIP er anvendt.

Med andre ord så er det ikke vores high-level modul som opretter/indeholder low-level modulet. I stedet har den et interface, som dette low-level modul implementere. På denne måde kan den underliggende funktionalitet let ændres/skiftes ud senere fordi high-level modulet ikke længere siger: "tænd for lampe", men i stedet siger "lav lys" og så er det nu vores implementering af dette interface som afgør om det er en lampe der tændes eller et stearinlys.

Klassediagrammet kan ikke være på én side så derfor kan billedet for øvelsen kan ses på [dette billede](#) fra github repo'et.

1.5 Hvordan fremmes godt SW design?

Alle tre design principper har deres fordele og nogen, deres ulemper.

Undgå Design Smells:

- Ridity.
Svært at ændre i koden, simple ændring skaber kaskader af andre lndringer.
- Fragility.
Programmet går i stykker, ved simple ændringer.
- Immobility.
Immobil at overføre til nyt system.
- Viscosity.
Hacking af løsninger.
- Needless complexity.
Siger sig selv.
- Needless repetition.
Genbrug af kode - hellere lav en abstraktion!
- Opacity.
Opacity er når koden er svær at forstå.

1.5.1 SRP

At separere klassers ansvar kan være en god ide, idet ethvert ansvar har dets egen "Axis of change". Flere ansvar i klasser medfører høj kobling blandt dem. Hvis en enkelt af disse ansvar skal ændres kan det medføre at resten af klassen bliver negativt påvirket eller "bare" skal genkompileeres, gen-testes og genimplementeres.

Man skal også passe på ikke at distribuere funktionalitet ud så meget at man får "needless complexity". Hvis en klasse ikke har "brug" for at ændre sig, er der ingen grund til at bruge SRP.

1.5.2 ISP

Ved at gøre klasser uafhængige af metoder som de ikke bruger opnår vi en række fordele:

- Øget sikkerhed.
Klassen *DoesA* kan ikke længere kalde *DoB()*.
- Lavere kobling mellem klasserne.
Nemmere at vedligeholde og ændre.

1.5.3 DIP

Det lugter meget af OCP¹, men i modsætning til OCP så taler vi her om høj -og lavniveau moduler samt lagdeling af moduler.

Ulig OCP går DIP ikke ud på at gøre softwaren “udvidbar”, men løsere koblet.

1.6 Eksempel

Gode SOLID eksempler på blog.gauffin.org.

1.7 Redegør for ulemper

Alting er ikke et søm, bare fordi man har en hammer! Forkert og/eller overdrevet brug af SOLID principperne, kan føre til needless complexity. Eksempelvis kan overdrevet klasseinddeling (SRP) give anledning til unødvendig kompleksitet - An axis of change is only an axis of change, if change occurs.

¹Se afsnit 2.2 om Open-Closed Principle.

2 Solid 2 - OCP, LSP og DIP

2.1 Fokuspunkter

- Redegør for:
 - Open-Closed Principle (OCP).
 - Lisskov's Substitution Principle (LSP).
 - Dependency Inversion Principle (DIP).
- Redegør for, hvordan du mener anvendelsen af principperne fremmer godt SW design.
- Vis et eksempel på anvendelsen af et eller flere af principperne i SW design.
- Redegør for konsekvenserne ved anvendelsen af OCP, LSP og/eller DIP - har det nogle ulemper?

2.2 Open-Closed Principle (OCP)

"Open for extension, closed for modification"

OCP siger at man bør refaktorere således at yderligere ændringer ikke skaber problemer¹ i resten af programmet.

Når OCP er "well applied" betyder det at vi kan tilføje ny kode uden at behøve ændre i eksisterende kode, som er testet og virker.

De 2 primære attributter:

1. Open for extension - Modulet kan udvides i takt med at krav udvides.
2. Closed for modification - Ved kun at udvide programmet, behøves vi ikke at røre ved eksterne filer, DLL'er og library filer.

2.2.1 HOW TO OCP

- Brug abstrakte klasser, som base.
- Eksempel - Client der bruger Server.
- To konkrete klasser er lort → brug et interface.



Figure 5: OCP brudt.

¹rekompilering, retest, redeploy.

2.2.2 Perspektiver

Strategy pattern En måde at opnå OCP på, kunne være brugen af et strategy pattern (eksternalisering af klasseansvar til strategier), i stedet af begynde at modificere en allerede eksisterende klasse.

2.3 Liskov's Substitution Principle (LSP)

"Subtypes must be substitutable for their base types"

Hvis en Tesla er en specialisering af en bil, så bør jeg kunne bruge bilens `drive()` metode på teslaen.

Lad os sige at en base klassen bil, har en `drive()` og en `shiftGearUp()` metode. Teslaen kan sagtens implementere `drive()` metoden men fordi det er en Tesla får vi et problem med gearskiftet! (et **throwException**) vil bryde OCP! Et `//Do Nothing` er sikkert fint men ikke særligt sikkert. Altså er en Tesla ikke en bil i Barbara Liskovs optik.

LSP handler dermed i bund og grund om ikke at bryde "er en" kontrakten med klient koden, og altså lave et arvehieraki der opfylder en ægte specialisering.

2.3.1 Pre -og postconditions

1. An overriding method may [only] **weaken the precondition**. This means that the overriding precondition should be logically "or-ed" with the overridden precondition.
2. An overriding method may [only] **strengthen the postcondition**. This means that the overriding postcondition should be logically "and-ed" with the overridden postcondition.

2.3.2 LSP overholdt

Hvis vi har følgende klasse Vehicle:

```
1 class Vehicle {
2     public void StartEngine() {
3         // Default engine start functionality
4     }
5     public void Accelerate() {
6         // Default acceleration functionality
7     }
8 }
```

Og vi så vil aflede to klasser, Car og ElectricCar.

```
1 class Car : Vehicle {
2     public void StartEngine() {
3         engageIgnition();
4     }
5     private void engageIgnition() {
6         // Ignition procedure
7     }
8 }
9
10 class ElectricCar : Vehicle {
11     public void accelerate() {
12         increaseVoltage();
13     }
14     private void increaseVoltage() {
15         // Electric logic
16     }
17 }
```

```
16     }  
17 }
```

Så skal begge være lavet så de kan skiftes ud med Car klassen. Således vil følgende funktionskald ikke give fejl og stadig virke som de skal, som set fra clientens side.

```
1 class Driver {  
2     public void Drive(Vehicle v) {  
3         v.StartEngine();  
4         v.Accelerate();  
5     }  
6 }
```

2.3.3 Brud på LSP

Hvis vi allerede har lavet en klasse *Rectangle*:

```
1 class Rectangle {  
2     int width, height;  
3     public void setHeight(int h){}  
4     public void getHeight(int h){}  
5     public void setWidth (int w){}  
6     public void getWidth (int w){}  
7 }
```

Og vi så vil lave en afledt klasse *Square*. Så burde dette være ligetil, men er en *Square* i programmering det samme som en *Rectangle*?

```
1 class Square : Rectangle {  
2     public void setHeight(int h){}  
3     public void setWidth (int w){}  
4 }
```

Her vil vi få et program da højde og bredde vil blive sat til det samme. Men hvad så hvis clienten forventer følgende test kan gennemføres?

```
1 class Client {  
2     public void AreaVerifier(Rectangle r) {  
3         r.setHeight(5);  
4         r.setWidth(4);  
5  
6         if(r.area() != 20) {  
7             System.Console.WriteLine("FUCK!");  
8         }  
9     }  
10 }
```

2.4 Dependency Inversion Principle (DIP)

DIP har følgende centrale punkter:

"High level modules should not depend on low level modules. Both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Det hedder dependency inversion fordi vi invertere afhængigheden for klasserne på figurene på side 9. Sådan at afhængigheden fra figur 6 bliver inverteret, som vist på figur 7.

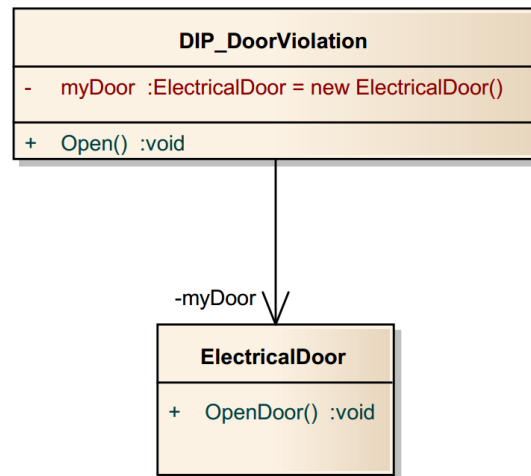


Figure 6: Eksempel på klasser hvor DIP ikke er anvendt.

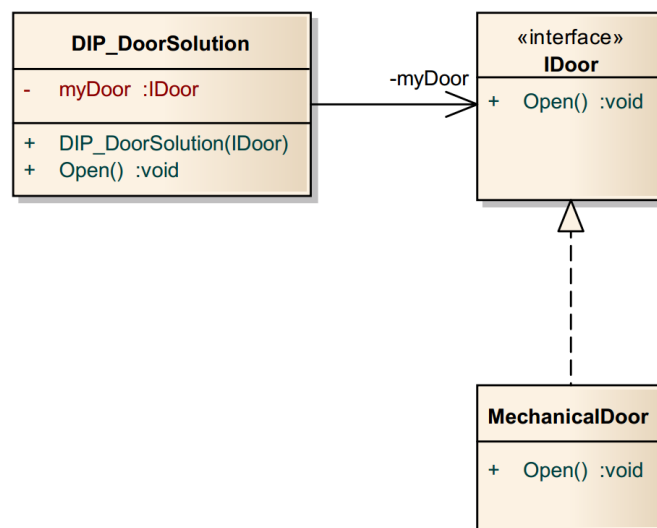


Figure 7: Eksempel på hvor DIP er anvendt.

Med andre ord så er det ikke vores high-level modul som opretter/indeholder low-level modulet. I stedet har den et interface, som dette low-level modul implementere. På denne måde kan den underliggende funktionalitet let ændres/skiftes ud senere fordi high-level modulet ikke længere siger: “tænd for lampe”, men i stedet siger “lav lys” og så er det nu vores implementering af dette interface som afgør om det er en lampe der tændes eller et stearinlys.

Klassediagrammet kan ikke være på én side så derfor kan billedet for øvelsen kan ses på [dette billede](#) fra github repo’et.

2.5 Hvordan fremmes godt SW design?

Alle tre design principper har deres fordele og nogen, deres ulemper.

Undgå Design Smells:

- Ridity.
Svært at ændre i koden, simple ændring skaber kaskader af andre lndringer.
- Fragility.
Programmet går i stykker, ved simple ændringer.
- Immobility.
Immobil at overføre til nyt system.
- Viscosity.
Hacking af løsninger.
- Needless complexity.
Siger sig selv.
- Needless repetition.
Genbrug af kode - hellere lav en abstraktion!
- Opacity.
Opacity er når koden er svær at forstå.

Godt software design fremmes, ved at implementere sit program, med henblik på at overholde eks. SOLID principperne. Til dette kan der specialiseres generelle løsninger (Design Patterns). Basically er keywords til godt software design: *Lav kobling, høj samhørighed, stærk indkapsling Testability, Reuseability, Readability* osv.

2.6 Eksempel

Gode SOLID eksempler på blog.gauffin.org.

2.7 Redegør for ulemper

Alting er ikke et søm, bare fordi man har en hammer! Forkert og eller overdrevet brug af SOLID principperne, kan føre til needless complexity. Eksempelvis kan overdrevet klasseinddeling (SRP) give anledning til unødvendig kompleksitet - An axis of change is only an axis of change, if change occurs.

3 Patterns 1 - GoF Strategy + GoF Template Method

3.1 Fokuspunkter

- Redegør for, hvad et Software Design Pattern er.
- Sammenlign de to design patterns GoF Strategy og GoF Template Method - hvornår vil du anvende hvilket, og hvorfor?
- Vis et designeksempel på anvendelsen af GoF Strategy.
- Redegør for, hvordan anvendelsen af GoF Template fremmer godt SW design.
- Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Strategy understøtter.

3.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](#)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

3.3 Sammenlign de to design patterns GoF Strategy og GoF Template Method - hvornår vil du anvende hvilket, og hvorfor?

Sidste del om hvilket vi vil anvende og hvornår er beskrevet afslutningsvis i denne section i afsnit [3.6.2](#) på side [15](#).

3.3.1 GoF Strategy Pattern

GoF strategy pattern er et Design Pattern, der gør det muligt at ændre en algoritmes opførsel, **runtime**.

Et strategy pattern På denne måde lader dette mønster disse algoritmer varieres alt efter hvad dets klienter ønsker.

3.3.2 GoF Template

Se afsnit [3.6.1](#) på side [14](#) om template pattern. Med template pattern kan opførsel ændres **compile-time** i modsætning til strategy pattern som kan gøre det på *run-time*.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

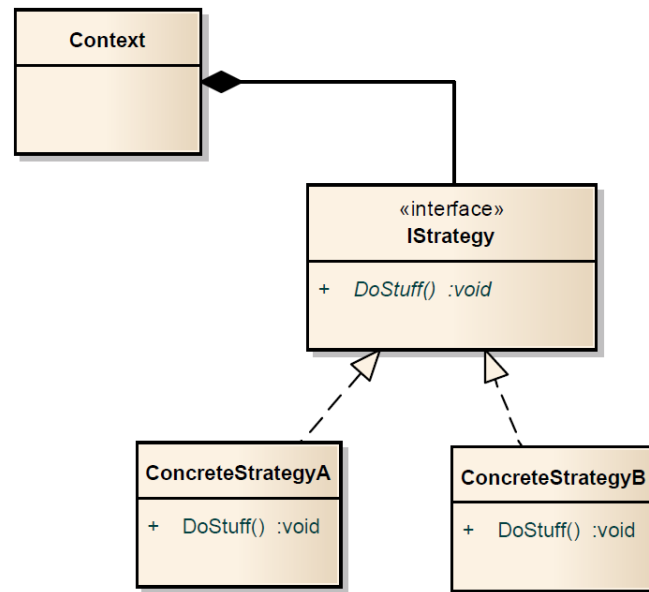


Figure 8: Simpel illustration af et strategy pattern

3.4 Vis et designeksempel på anvendelsen af GoF Strategy

Lad os tage udgangspunkt i figur 8. Vores kontekst (klient) er en calculator der opererer på 2 integers. Vi har da et ICalculator interface, med en enkelt virtuel metode, *calculate()*. Herudover har vi de 2 afledte klasser **Plus** og **Minus**.

- Definerer en *familie* af algoritmer.
- Indkapsler hver algoritme.
- Gør algoritmerne *interchangable* i dennes familie.

Klientens constructor sætter det pågældende objekts strategy. Se main, hvor et client objekt oprettes og initialiseres.

```
1 //Interface
2 public interface ICalculate {
3     int Calculate(int value1, int value2);
4 }
5
6 /*Concrete strategies*/
7
8 // Strategy 1: Minus
9 class Minus : ICalculate {
10     public int Calculate(int value1, int value2) {
11         return value1 - value2;
12     }
13 }
14
15 //Strategy 2: Plus
16 class Plus : ICalculate {
17     public int Calculate(int value1, int value2) {
18         return value1 + value2;
19     }
20 }
```

```
21
22 //klienten
23 class CalculateClient {
24     private ICalculate calculateStrategy;
25
26     //Constructor: assigns strategy to interface
27     public CalculateClient(ICalculate strategy) {
28         this.calculateStrategy = strategy;
29     }
30
31     //Executes the strategy
32     public int Calculate(int value1, int value2) {
33         return calculateStrategy.Calculate(value1, value2);
34     }
35 }
36
37 //Initialisering
38 int application(object sender, EventArgs e) {
39     CalculateClient minusClient = new CalculateClient(new Minus());
40     Response.Write("<br />Minus: " + minusClient.Calculate(7, 1).ToString());
41
42     CalculateClient plusClient = new CalculateClient(new Plus());
43     Response.Write("<br />Plus: " + plusClient.Calculate(7, 1).ToString());
44 }
```

3.5 Redegør for, hvilke(t) SOLID-princip(er) du mener anvendelsen af GoF Strategy understøtter

Overholdelse af OCP Som det kan ses på figur 8, er context (Klienten) open og closed, og OCP er herved overholdt. Klassen i sig selv bruger interfacet IStrategy, imens objekter af klienten bruger interfacets afledte klasser. Dette betyder, at hvis vi ønsker at bruge en ConcreteStrategyC skal vi ikke ændre noget i klient klassen. Vi skal derimod blot sætte klient objektet til at bruge den nye strategy.

Overholdelse af DIP Overholdelse af DIP ligger implicit i brugen af dette mønster - Bedst eksemplificeret med CompressionStocking klassediagrammet, hvor dependencies mellem Stocking-Controller klassen og Lace/AirCompressionCtrl er inverterede - Vi bruger interfacet ICompressionCtrl.

Overholdelse af ISP Strategy mønstret giver i dets beskrivelse udtryk for en opdeling af algoritmer i "familier" eg. interfaces).

Overholdelse af LSP Vi tager igen udgangspunkt i *familiebeskrivelsen* af strategy mønstret - Per definition skal alle subtyperne af samme interface være *interchangeable*. Dette er mindre relevant for netop dette mønster idet man sjældent vil bevæge sig længere ned i et arvehieraki end første lag.

3.6 Redegør for, hvordan anvendelsen af GoF Template fremmer godt SW design

Hvis man har et system bestående af nogle klasser (eller én), hvor disse klasse funktionalitet kun afviger lidt fra hinanden. Så kan *Template pattern* bruges. Via dette pattern kan et fast "programflow" defineres. Når dette "flow" så er fastlagt skal klasserne bare implementere/ændre (nok via override) de metoder som de ikke er tilfredse med. Et klassediagram kan ses på figur 9 på side 14.

3.6.1 Eksempel på Template pattern

Herunder er en abstrakt klasse der har metoder, som de mange spil bruger/følger. Eksemplet er taget fra [wikipedia](#) om template pattern

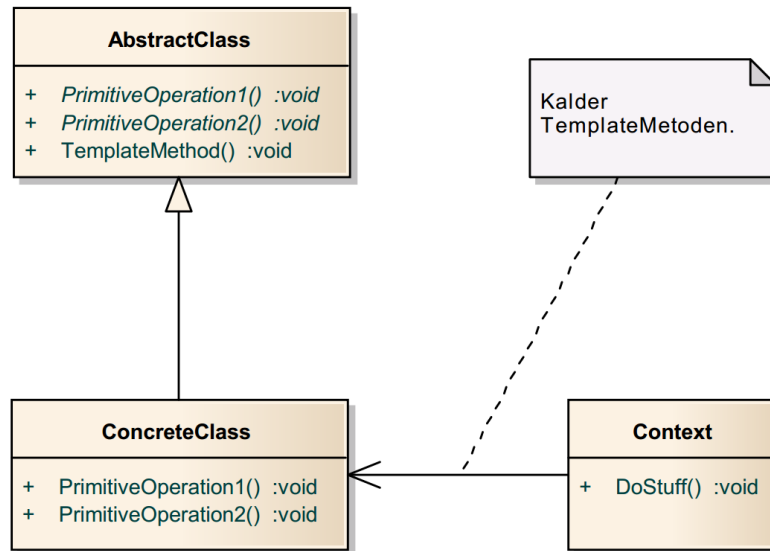


Figure 9: Klassediagram for Template pattern. Viser hvordan arv bruges til at implementere de specifikke metoder i en klasse, men programflow overlades til en abstrakt klasse

```

1 abstract class Game {
2     // Hook methods. Concrete implementation may differ in each subclass
3     protected int PlayerCount;
4     abstract void InitializeGame();
5     abstract void MakePlay(int player);
6     abstract void EndOfGame();
7     abstract void AnnounceWinner();
8
9     // A template method:
10    public final void PlayGame(int playerCount) {
11        PlayerCount = playerCount;
12        InitializeGame();
13        int j = 0;
14        while(!EndOfGame()) {
15            MakePlay(j);
16            j = (j + 1) % PlayerCount;
17        }
18        AnnounceWinner();
19    }
20 }

```

Herunder ses så hvordan den specifikke implementering af den abstrakts klasse kan laves.

```

1 class Monopoly : Game {
2     /* Implementation of necessary concrete methods */
3     void InitializeGame() {
4         // Initialize players
5         // Initialize money
6     }
7     void MakePlay(int player) {
8         // Process one turn of player
9     }
10 }

```

```
9      }
10     boolean EndOfGame() {
11         // Return true if game is over
12         // according to Monopoly rules
13     }
14     void PrintWinner() {
15         // Display who won
16     }
17
18     /* Specific declarations for the Monopoly game. */
19     // ...
20 }
```

3.6.2 Hvornår vil du anvende hvilket, og hvorfor?

Strategy Hvis man ikke har et fastlagt "program-flow" og har brug for at kunne skifte mellem flere implementeringer på *run-time* vil strategy nok være den bedste løsning.

Eksempel: Når man skal transmittere data over et netværk, kan det være en fordel at nogle gange gøre det med TCP og andre gange en UDP overførsel. Det er den samme data der sendes, men 2 forskellige "algoritmer" til at udføre transmissionen.

Template Har man et bestemt "flow" som alle klasserne i ens system skal følge, vil template pattern være en god løsning. Den gør det muligt at overlade den specifikke implementering af "underfunktioner" til andre klasser. Samtidigt vil måden disse funktioner bruges på være fast bestemt via den overliggende abstrakte klasse. Tilgængæld kan opførsel kun ændre på *compile-time*.

4 Patterns 2 - GoF Observer

4.1 Fokuspunkter

- Redegør for, hvad et Software Design Pattern er.
- Redegør for opbygningen af GoF Observer.
- Sammenlign de forskellige varianter, af GoF Observer - hvilken vil du anvende hvornår?
- Redegør for, hvordan anvendelsen af GoF Observer fremmer godt software design.
- Redegør for fordele og ulemper ved anvendelsen af GoF Observer.
- Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Observer undersøger.

4.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](https://da.wikipedia.org/wiki/Design_pattern)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

4.3 Redegør for opbygningen af GoF Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"

Opbygningen kan ses på figur 45 side 63, som viser et klassediagram for opbygning af dette pattern.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

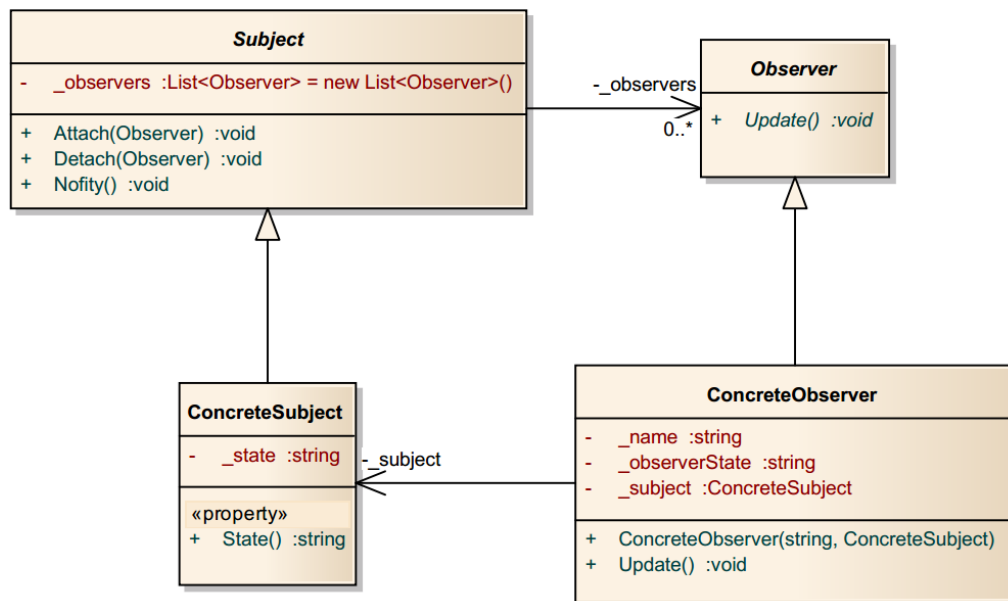


Figure 10: Klassediagram som viser GoF Observers opbygning.

4.4 Sammenlign de forskellige varianter, af GoF Observer - hvilken vil du anvende hvornår?

I de fleste systemer vil det ikke være tilstrækkeligt at blot notify observeren om en ændring, men derimod også fortælle hvad der konkret er sket.

4.4.1 Pull

I pull variationen er det observeren der står for at finde ud af hvilket state change der er sket. Her vil observeren have association med subjectet således at det er muligt for observeren at få fat i det state som er ændret. Altså trækker den det ud af subjectet.

4.4.2 Push

I push variationen er det subjecten der i dens notify funktion, skal videregive dens state som parameter. Observerens `Update()` funktion skal da også indrette sig derefter.

4.4.3 Many observers to many subjects

Hvis vi har flere observers som vil observere mere end ét subject objekt, er det ikke nok at `Notify()` om ændringen. I sådant et tilfælde vil det også være nødvendigt at gøre opmærksom på hvilket subject, som ændringen er sket i.

4.5 Who triggers the update?

Hvis subjectet opdateres meget ofte og vil resultere i unødvendige ændringer af observeren, vil det være bedre om observeren stod for at *trigge* opdateringen.

4.6 Redegør for fordele og ulemper ved anvendelsen af GoF Observer

4.6.1 Fordele

Gør det muligt for klasser at reagere på ændringer i en anden klasse uden at koble dem for hørdt.

4.6.2 Ulemper

Kan give anledning til *memory leaks*. Fordi den simple implementering kræver explicit *registering* og *deregistering*. Dette koblet med at *Subject*-klassen har en *strong reference*¹ til de observers, som kan holde dem unødigt i live.

Dette kan dog undgås ved at bruge en *weak-reference*, således at *Subject*'s reference til observeren ikke beskytter mod garbage collectoren hvis det er den eneste.

4.6.3 Making sure the subject state is self-consistent

Det er vigtigt at vi ikke kalder *Notify()* funktionen før vores *state* faktisk er opdateret. En løsning er beskrevet på denne side <http://www.oodesign.com/observer-pattern.html> og går ud på at lave en template for kaldet, som også kan ses herunder:

```
1 public void final updateState(int increment)
2 {
3     doUpdateState(increment);
4     notifyObservers();
5 }
6
7 public void doUpdateState(int increment)
8 {
9     state = state + increment;
10 }
```

I dette tilfælde vil vores *ConcreteSubject* bare skulle **override** *doUpdateState(int)* funktionen og ikke skulle tænke på at kalde *Notify()* på det rigtige tidspunkt.

4.7 Redegør for, hvordan anvendelsen af GoF Observer fremmer godt software design

- Lav kobling.
- Let at udvide.

4.8 Redegør for, hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Observer undersøger

OCP Vi kan let tilføje flere observers uden at skulle ændre i subject eller anden kode.

¹Se wiki om weak reference:
https://en.wikipedia.org/wiki/Weak_reference

5 Patterns 3 - GoF Factory Method/Abstract Factory

5.1 Fokuspunkter

- Redegør for, hvad et software design pattern er.
- Redegør for opbygningen af GoF Factory Method og GoF Abstract Factory.
- Giv et designeksempel på anvendelsen af GoF Abstract Factory.

5.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](https://da.wikipedia.org/wiki/Design_pattern)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

5.3 Redegør for opbygningen af GoF Factory Method

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses"

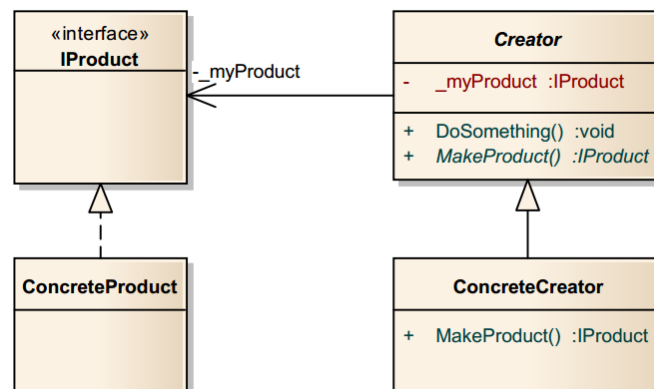


Figure 11: Klassediagram for FactoryMethod

Omkring selve implementeringen af en klasse m.m. som bruger factory method kan listing 1 ses. Den overlader kort sagt valg af konkret produkt til en subclasse, som implementerer `MakeProdukt()`.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

```
1 public abstract class Creator
2 {
3     // IProduct to be used for stuff
4     IProduct _myProduct;
5
6     // Constructor
7     Creator()
8     {
9         _myProduct = MakeProduct();
10        Console.WriteLine(_myProduct.GetType().Name + " says hello!");
11    }
12
13    // Factory method -> to be implemented in subclass...
14    public abstract IProduct MakeProduct();
15 }
16
17 public class ConcreteCreator : Creator
18 {
19     public override IProduct MakeProduct()
20     {
21         // returns some subclass of IProduct
22         return new ConcreteProduct();
23     }
24 }
```

Code listing 1: Realisering og brug af factory method.

5.4 Redegør for opbygningen af GoF Abstract Factory

5.4.1 Problem

Hvis man har et klassediagram som ser ud noget i stil med det på figur 12 og en main ligner noget ala det i listing 2 så står det meget hurtigt en klart at dette er meget rodet og uoverskueligt. For at undgå dette babushka helvede bliver løsningen **Abstract Factory** og dette er beskrevet i næste afsnit.

```
1 IPump myPump = new Pump();
2 ITightner myTightner = new Tightner();
3 ICompressionCtrl myCompressionCtrl = new LaceCompressionCtrl(myTightner);
4
5 INotificationDevice myGreenLed = new LedGreen();
6 INotificationDevice myReDevice = new LedRed();
7 INotificationDevice myVibrator = new Vibrator();
8 INotification myNotification = new Notification(myGreenLed, myReDevice, myVibrator);
9
10 IButtonHandler myStockingCtrl = new StockingCtrl(myCompressionCtrl, myNotification);
11 myCompressionCtrl.AddNotificationCenter(myEventHandler);
```

Code listing 2: Main for compressionstocking.application.

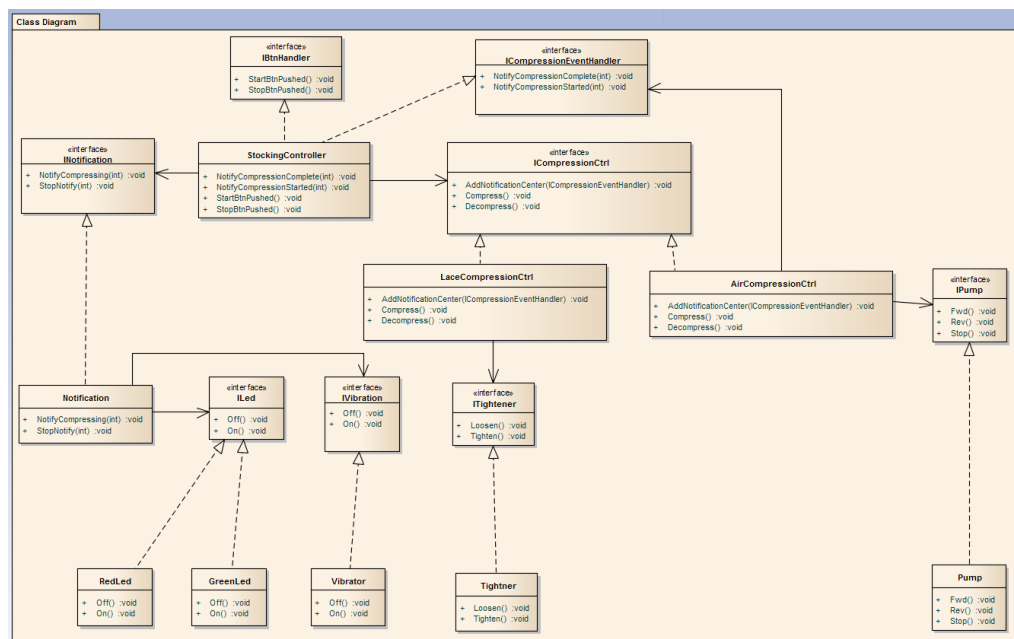


Figure 12: Klassediagram for compressionstocking systemet.

5.4.2 Abstract Factory

Abstract Factory Pattern kan bruges til at "hjælpe" et object i dets constructor med at instantiere dets medlemmer, som det kan ses herunder i listing 3 og en implementering af *CreateHerbivore()* kan også ses i listing 4.

```

1 public class AnimalWorld
2 {
3     Herbivore _herbivore;
4     Carnivore _carnivore;
5
6     public AnimalWorld(ContinentFactory factory)
7     {
8         _herbivore = factory.CreateHerbivore();
9         _carnivore = factory.CreateCarnivore();
10    }
11    // other functions
12 }
  
```

Code listing 3: Abstract Factory brugt i Client Constructor().

```

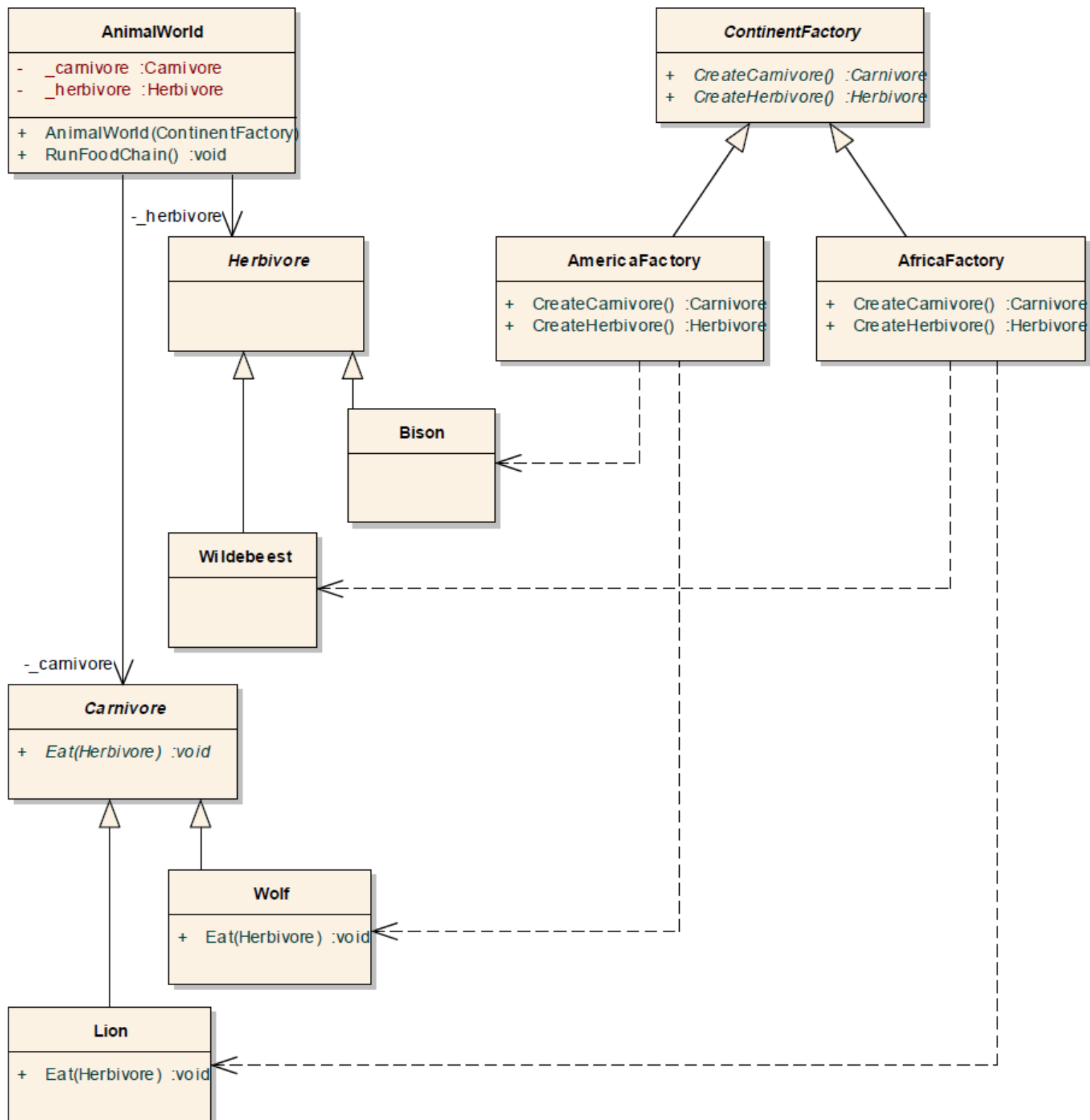
1 public override Herbivore CreateHerbivore()
2 {
3     return new Bison();
4 }
  
```

Code listing 4: Eksempel på implementering af CreateHerbivore metode.

På klassediagrammet på næste side kan opbygningen af abstract factory ses.

5.5 Giv et designeksempel på anvendelsen af GoF Abstract Factory

Brug det store klassediagram og dets klasser som designeksempel. Drop muligvis enten plante eller kødæder klassen.



6 Patterns 4 - State patterns

6.1 Fokuspunkter

- Redegør for, hvad et software design pattern er.
- Redegør for strukturen i GoF State Pattern.
- Sammenlign switch/case-implementering med GoF State.
- Redegør for fordele og ulemper ved anvendelsen af GoF State.
- Redegør for, hvordan et UML (SysML) state machine diagram mapper til GoF State.

6.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](#)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

6.3 Redegør for strukturen i et state pattern

Et state pattern er et behavioral pattern, og er en måde at implementere en state machine på.

State mønstret definerer en måde hvorpå vi kan ændre et objekts opførsel eftersom dens interne state ændres (run-time). State mønstret giver mulighed for at implementere store forskelle i opførslen uden brug af masser af Switches og if sætninger.

1. Context klassen
 - Definerer et interface der bruges af klienter.
 - Indeholder en instans af en ConcreteState subclass, der kan definere en Current State.
 - Indeholder Actions.
2. StateBase klassen
 - Et interface til indkapsling af ConcreteStates adfærd
 - Indeholder virtuelle handles.
3. Concrete State
 - Hver ConcreteState implementerer den adfærd der associeres til i Context klassen (handle til hver action).

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

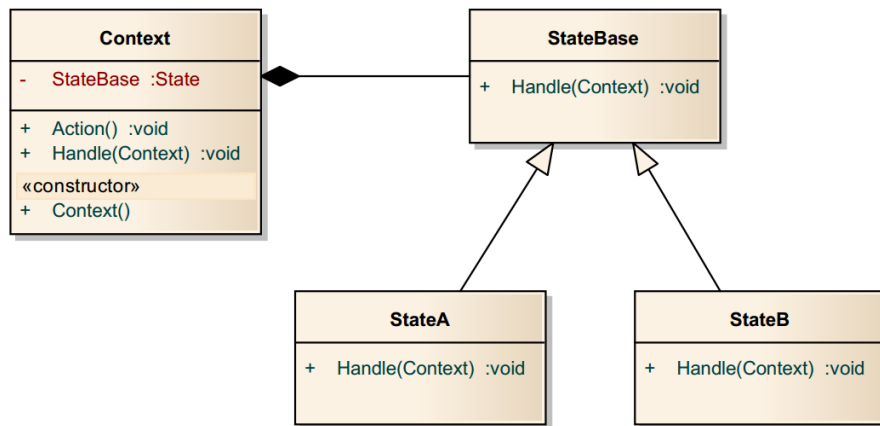


Figure 13: UML for et GoF state pattern

6.3.1 Kodeeksempel

Dette er et kodeeksempel på en state machine der kan toggle dens state.

```

1 abstract class State
2 {
3     public abstract void Handle(Context context);
4 }
  
```

Code listing 5: StateBase klassen

Som det ses på klassen StateBase implementering, har den blot en abstrakt metode der tager mod en Context

```

1 class ConcreteStateA : State
2 {
3     public override void Handle(Context context)
4     {
5         context.State = new ConcreteStateB;
6     }
7 }
  
```

Code listing 6: ConcreteStateA klassen - Switcher til State B

```

1 class ConcreteStateB : State
2 {
3     public override void Handle(Context context)
4     {
5         context.State = new ConcreteStateA;
6     }
7 }
  
```

Code listing 7: StateBase klassen - Switcher til state A

```
1 class Context
2 {
3     private State _state;
4
5     // Constructor
6     public Context(State state)
7     {
8         this.State = state;
9     }
10
11    // Gets or sets the state
12    public State State
13    {
14        get { return _state; }
15        set
16        {
17            _state = value;
18            Console.WriteLine("State: " + _state.GetType().Name);
19        }
20    }
21
22    public void Request()
23    {
24        _state.Handle(this);
25    }
26 }
```

Code listing 8: Context klassen - Bruges i main() til at kalde Request()

I lærebogen erklæres alle states static inde i contexten idet der således ikke skal laves en ny, hver gang der skiftes state... Smart.

State vs. Strategy. Ser man på UML'en for et State Pattern, ligner det jo et strategy pattern. I et State pattern eksisterer der dog et constraint idet hver ConcreteState klasse skal bruge en reference til en Context (den vælger og invoker contextens metoder igennem denne reference). Dette constraint eksisterer ikke i et strategy pattern:

```
1 IStrategy myStrategy = new s1();
2 myStrategy.StrategyFunction();
3
4 myStrategy = new s2();
5 myStrategy.StrategyFunction();
```

Code listing 9: Klient's brug af strategy pattern

Ortogonale og nastede states gennemgås i afsnittet om UML mapping.

6.4 Sammenlign switch/case-implementering med GoF State.

6.4.1 Switch case implementering af STM

- En switch case implementering af en STM er det simple udgave.
- Koden bliver meget hurtig uoverskuelig.
- Den er sværere at teste (Dårligt separeret)
- Sværere at vedligereholde.

6.4.2 GoF State implementering af STM

- Grundig implmentering af STM.
- Alle states er inddelt i subclasser af en State baseklasse.
- Derfor let at teste.
- Og lettere at vedligeholde.

6.5 Redegør for fordele og ulemper ved anvendelsen af GoF State.

6.5.1 Fordele

- Let at teste - de mange klasser gør det enkelt at afgrænse tests.
- Let at udvide - Skalerer nemt ved udvidelser af både states og transitions.
- Kan lettere håndtere Nested States.
- Forholdsvis simpelt at implementere Ortogonale states.

6.5.2 Ulemper

- Class Explosion - ved et simpelt STM skal der oprettes mange klasser for meget simpel logik, der vil derfor være en nedre grænse for hvornår man vil implementere dette.
- Memory - Flere klasser betyder mere memory allokering.
- Readability - i mere komplekse programmer vil koden blive svær at overskue, men er dog stadig væsentligt nemmere end alternativerne.

6.6 Redegør for, hvordan et UML (SysML) state machine diagram mapper til GoF State

6.6.1 Simpel UML

Et simpelt STM UML Diagram:

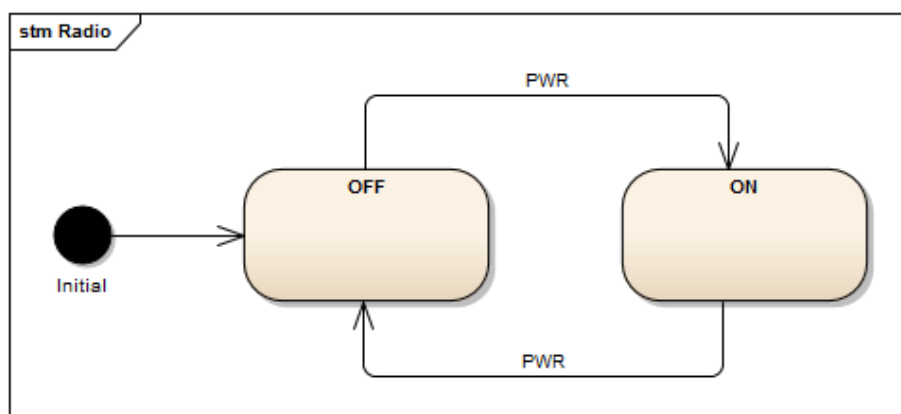


Figure 14: Et UML State Machine Diagram

uløst problem med
OnExit()
Andreas/-
Denned??

Et tilsvarende GoF State Pattern klassediagram for en radioen:

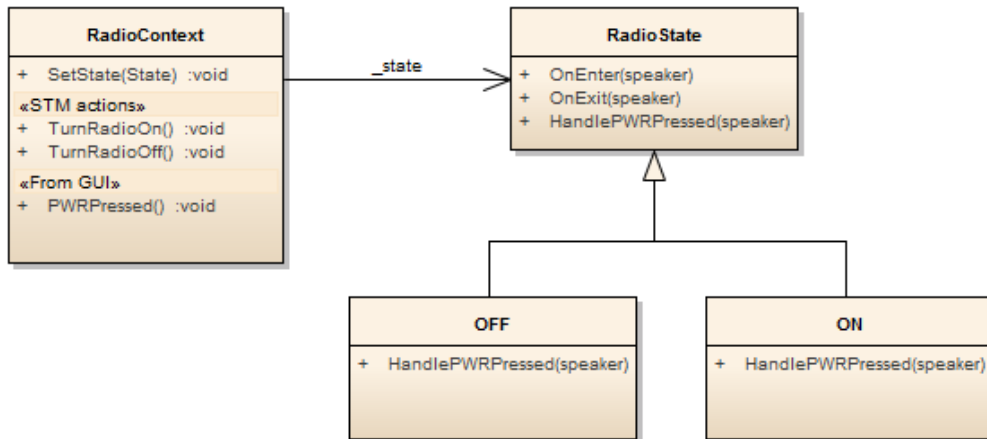


Figure 15: Et UML Klassediagram for STD

6.6.2 Nested States

Et STM UML diagram med nested states:

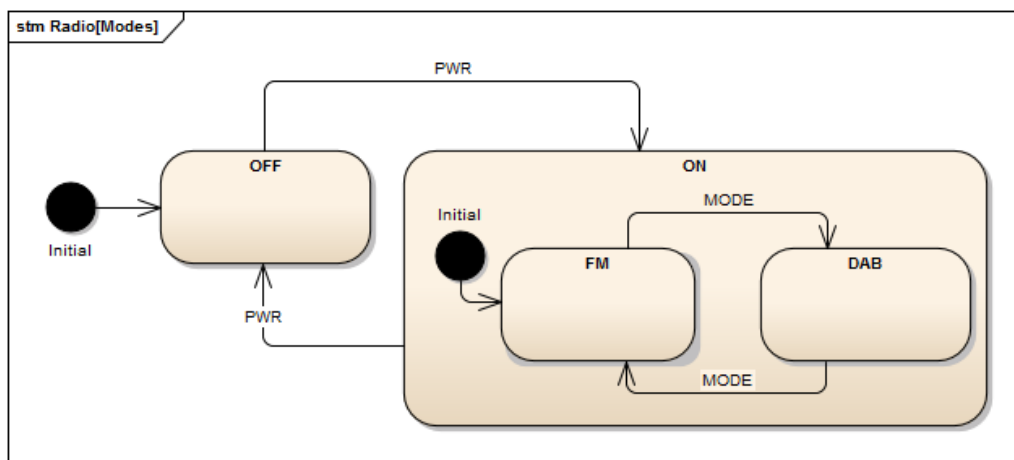


Figure 16: Et UML State Machine Diagram for Nested states

Et tilsvarende GoF State Pattern klassediagram med nested states:

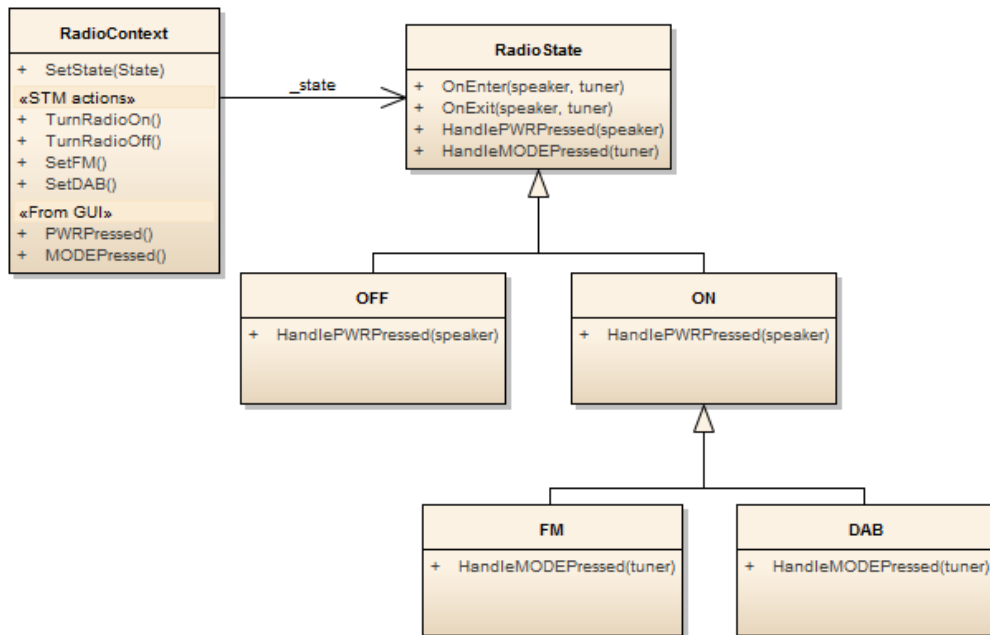


Figure 17: Et UML Klassediagram for Nested states

Vi kan her se i det tilsvarende State pattern, at den blot har to state klasser der hedder “FM” og “DAB” der nedarver fra ON klassen som set tidligere. og dertil mindre ændringer i Context klassens funktioner.

6.6.3 Ortogonale States

Et STM UML diagram med ortogonale states på figur 18:

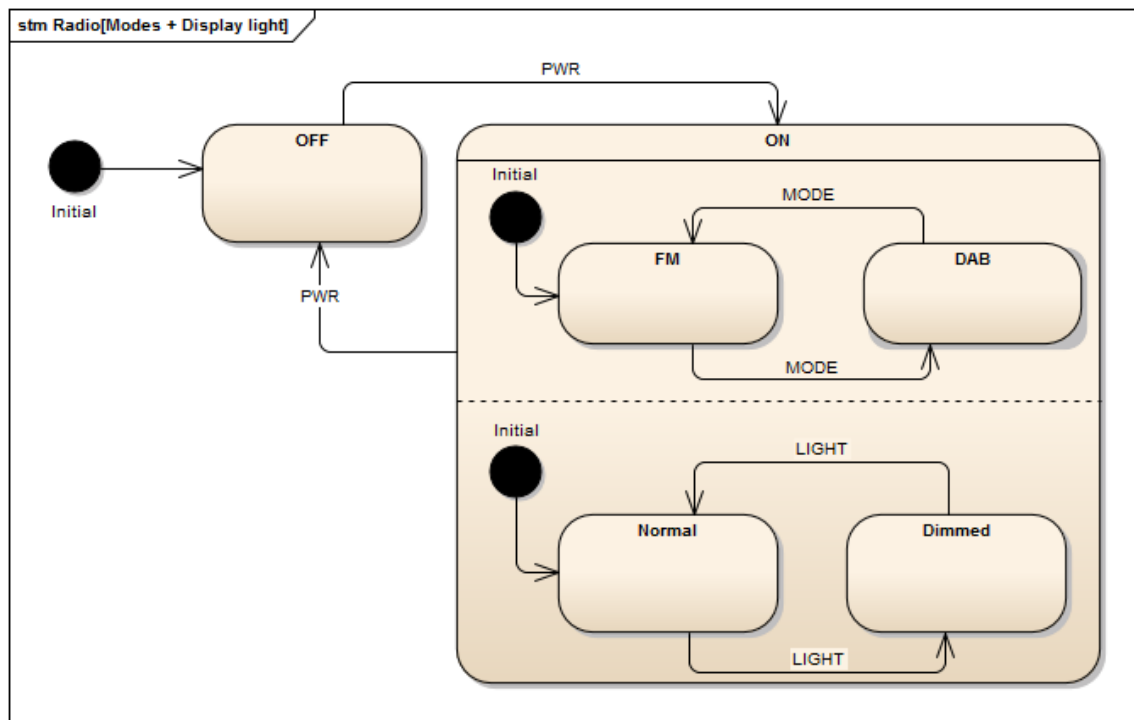


Figure 18: Et UML State Machine Diagram for ortogonale states

Et tilsvarende GoF State Pattern klassediagram med ortogonale states:

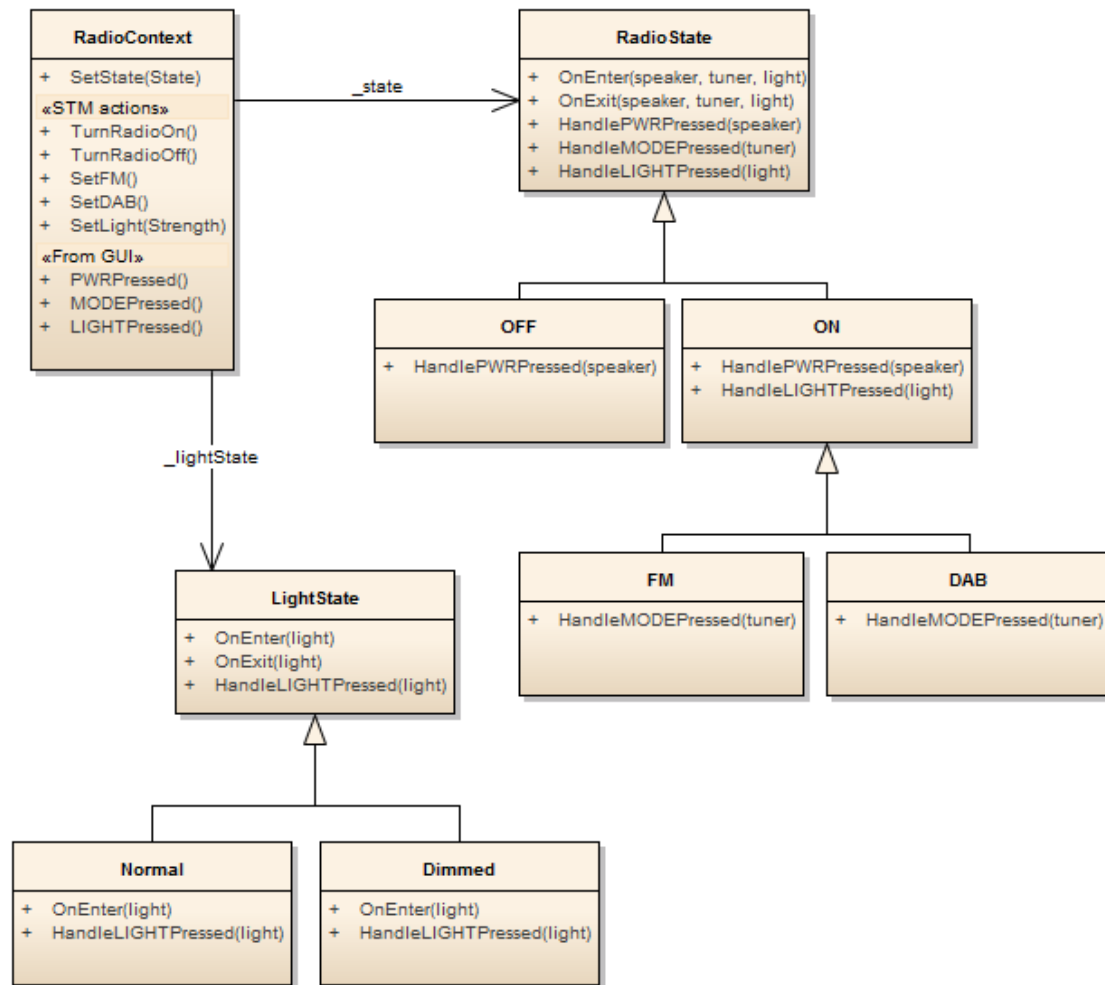


Figure 19: Et UML Klassediagram for Nested states

Læg her mærke til at Ortogonale statemachines, kræver blot en ekstra “superstate” med dertilhørende substates, denne superstate bliver en reference i contexten som dermed gør at de to forskellige superstates kan tilgå hinanden.

7 Patterns 5 - Model View Controller og Model View Presenter

7.1 Fokuspunkter

- Redegør for, hvad et software design pattern er.
- Redegør for Model-View-Control mønstret og dets variationer.
- Redegør for Model-View-Presenter mønstret og dets variationer.

7.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](https://da.wikipedia.org/wiki/Design_pattern)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

7.3 Redegør for Model-View-Control mønstret og dets variationer

MVC er et populært software pattern brugt til GUI applikationer. Ifølge Fowler er dette pattern tit misquoted. Dette skyldes at der er mange elementer i den klassiske MVC der ikke rigtig kan bruges i nutidens GUI applikationer.

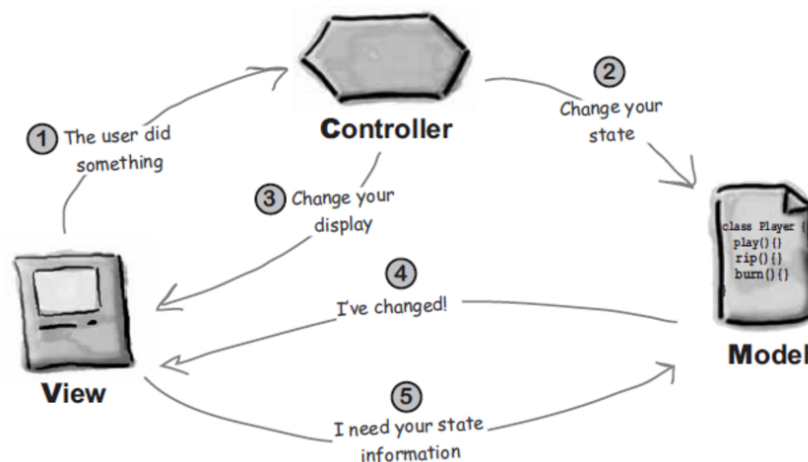


Figure 20: MVC Passiv variation, eksemplificeret med en musik afspiller.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

7.3.1 MVC's opbygning

Model Indeholder logik, data og states for programmet. Model kender intet til UI'et eller controller, og indeholder altså kun funktionalitet for selve business logikken (Business logik er det som aktivt gør - Repræsentation/oprettelse/storing/ændring af data). Model har et interface således at dets **state** kan tilgås eller manipuleres. Kan sende notifikationer om dets state til **observer**. Model notifier viewet, når dets state ændres - Se evt afsnittet om observer.

View Indeholder præsentationslogik (knapper, tekstbokse, etc). Brugeren af applikationen interagerer med view, som fortæller controlleren om *din* "action". View spørger model om at få fat i data fra modellen, når modellen har annonceret en ændring i sit state. Det er muligt at implementere flere views. Der kan eksempelvis routes til forskellige URL's i controlleren.

Controller Bindeleddet mellem View og Model. Controlleren indeholder logik til at finde ud af, hvad der skal ske i modellen, når brugeren trykker på noget. **Nogen gange** kan controlleren bede viewet om at ændre sig når controlleren modtager en "action" fra viewet (Enable/disable knapper/menuer).

Brugeren kan fx trykke på en knap, som registreres af controlleren. Controlleren udfører de nødvendige handlinger på modellen, som har kontakt med evt. harddisk/database. Modellen sender svar tilbage og controlleren kalder det korrekte view, som viser dataen til brugeren.

7.3.2 Passive model

"The passive model is employed when one controller manipulates the model exclusively"

The controller modifies the model and then informs the view that the model has changed and should be refreshed (se figur 21).

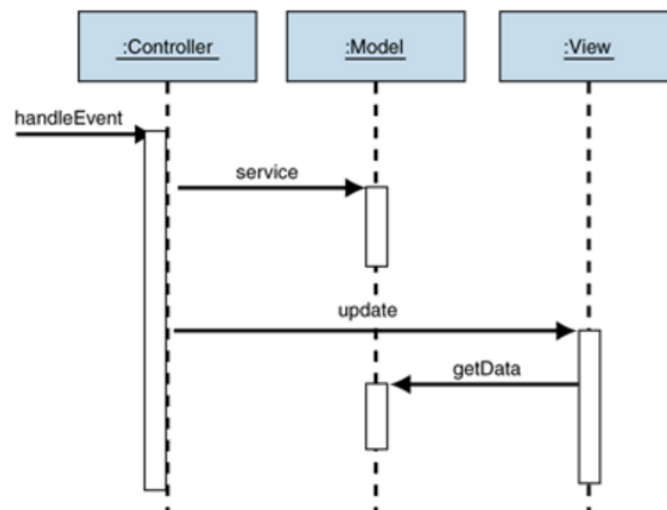


Figure 21: Passiv variation af MVC.

Modellen i dette scenarie er uafhængig af View og Controller. Altså er der ingen måde hvorpå Modellen kan rapportere ændringer i sig selv.

Følgende forklaring er taget fra <https://msdn.microsoft.com/en-us/library/ff649643.aspx>:

"The HTTP protocol is an example of this. There is no simple way in the browser to get asynchronous updates from the server. The browser displays the view and responds to user input, but it does not detect changes in the data on the server. Only when the user explicitly requests a refresh is the server interrogated for changes"

7.3.3 Active model

Den aktive model virker som ved et Observer pattern (illustreret i figur 22) hvorved View subscriber på events fra Model og derpå selv hente opdateringen fra Model.

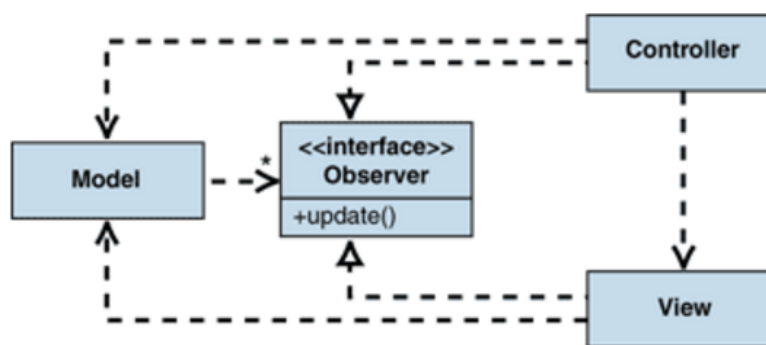


Figure 22: MVC active, observer pattern.

Hvis et givet View kun er interesseret i nogle bestemte oplysninger fra Model (altså IKKE alle mulige tilstande) så vil det være en god ide at lave flere Subjects¹ for så derved at lade hvert View "subscribe" på de events som den selv er interesseret i.

¹jf. emne 4 om Observer Pattern og Subject.

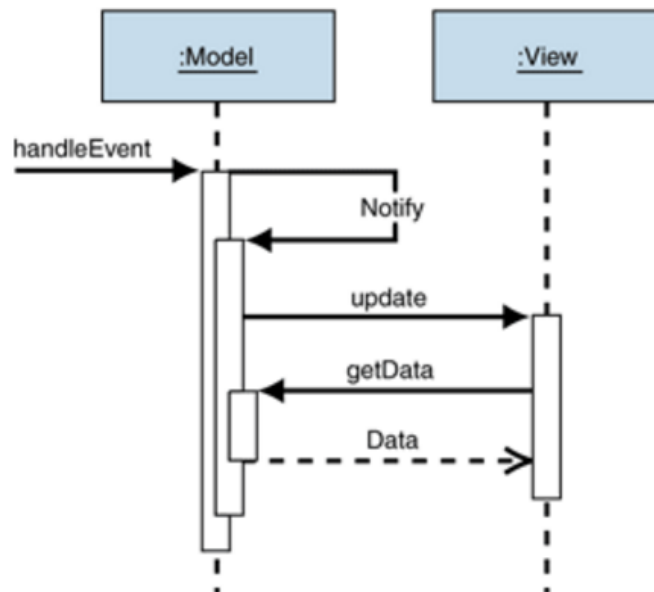


Figure 23: Aktiv variation af MVC.

Følgende forklaring er taget fra <https://msdn.microsoft.com/en-us/library/ff649643.aspx>:

"The active model is used when the model changes state without the controller's involvement. This can happen when other sources are changing the data and the changes must be reflected in the views.

Consider a stock-ticker display. You receive stock data from an external source and want to update the views (for example, a ticker band and an alert window) when the stock data changes. Because only the model detects changes to its internal state when they occur, the model must notify the views to refresh the display"

7.3.4 Fordele ved MVC

- Skaber høj separation mellem præsentationen (view og controller) og domænet (model).
 - View og controller er separeret for at overholde SRP.
 - forskellige Interfaces til kommunikation mellem Model-Controller og Model-View for at overholde ISP.
 - Separation of concerns.
 - Nemmere at teste.
 - Nemmere at vedligeholde.
- Inddeler GUI widgets i en controller som reagerer på brugerinput og view der viser modellens state. View og controller skal som regel ikke kommunikere direkte, men gennem modellen.
- Views (og controllere) kan observere modellen, så det er muligt for **flere widgets at opdatere**, uden nødvendigvis at behøve at kommunikere direkte. (Observer synkronisering).

Eksempel Brugeren af en media player ønsker at skifte til næste sang.

1. Bruger-input til view.
2. Controller behandler bruger input og beder model ændre sit state herefter (kalder fx play() funktionen. Hvis det er nødvendigt kan controlleren få viewet til at ændre sig (grey out button etc.).
3. Model annocerer at den har ændret sig.
4. View beder om data på ændringen (fx en string i form af navnet på sangen).
5. View opdateres.

7.4 Redegør for Model-View-Presenter mønstret og dets variationer

MVP er en videreudviklet version af MVC, der forsøger at forene Forms and Controls med MVC.

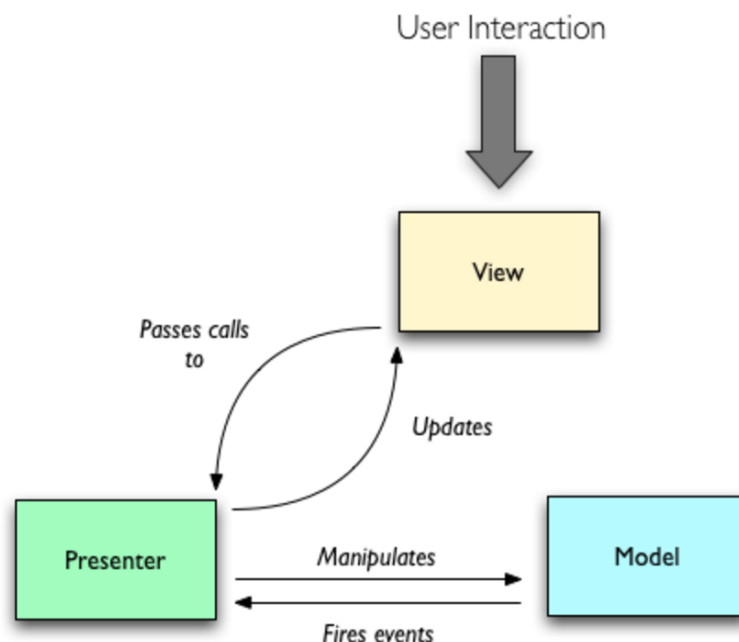


Figure 24: Programmet's flow i en MVP applikation

7.4.1 MVP's opbygning

Model Indeholder logic til at interface med programmets data.

View Modtager som i MVC UI actions og kalder derved presenters funktionalitet. View indeholder typisk eventhandler og logic til at kalde presenters funktionalitet.

Presenter Binder model til view (de-coupling). Har ansvaret for at opdatere view når ny data fra model. Al business logic der bruges til at behandle user inputs skrives i presenter. Presenter skal indhente data fra model, og behandle så det er klar til brug i view uden overhead.

7.4.2 Fordele ved MVP

- I og med vi har frtaget View alt dens “myndighed” og gjort den passiv. Kan vi skifte views (ændre hvordan vi ønsker at repræsentere modellen).
- Stærkere **separation of concerns** - view står KUN for rendering.
- Lettere at teste.

7.4.3 Variationer af MVP

Der findes to primære variationer af MVP, begge illustreret i figur 25.

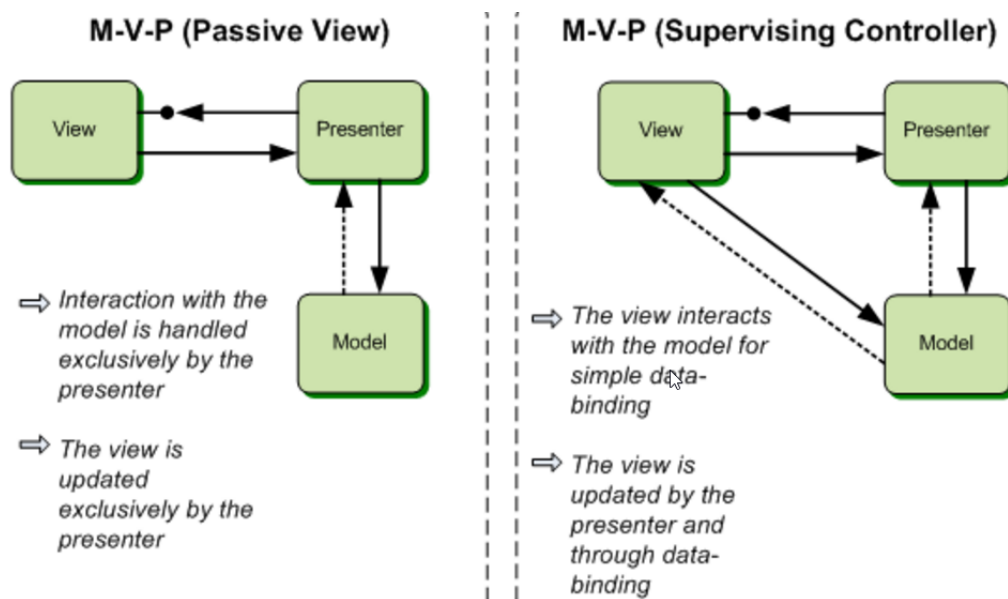


Figure 25: Variationer af MVP.

Passive View Se MVP's opbygning i figur 24 på side 35.

Supervising Controller En variation af MVP hvor viewet er mere intelligent. I denne variation har View mulighed for at kommunikere **direkte** med model og requeste data.

Presenter står her for at informere Modellen om aktiviteter i view (der måske ændrer state). Presenter passer en reference til model så at View kan interagere med Model når dens state ændres. På denne måde minder MVP SC meget om MVC Aktiv idet den totale de-coupling mellem view og model nu er blevet vag.

Hvorfor bruge MVP SC? SC kræver mindre kode end Passive View idet Presenteren ikke laver simple view update. Dette er dog på bekostning af testability - Dårligere separation.

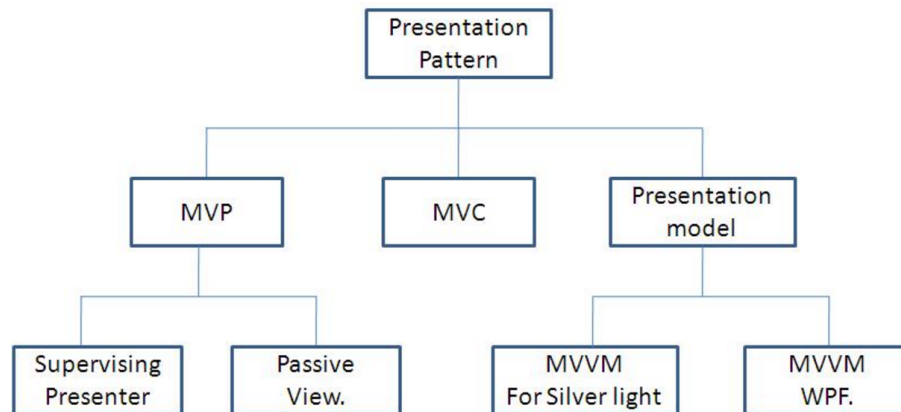


Figure 26: De forskellige GUI patterns og deres variationer

8 Patterns 6 - Model-View-ViewModel (MVVM)

8.1 Fokuspunkter

- Redegør for, hvad et software pattern er.
- Redegør for Model-View-ViewModel mønstret og dets variationer.

8.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](https://da.wikipedia.org/wiki/Design_pattern)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

Dette design pattern, er et GUI architecture pattern, som er en specialisering af Model-View-Presenter mønstret.

8.3 Redegør for Model-View-ViewModel mønstret og dets variationer

8.3.1 View

1. User Interface.
2. Definerer struktur, layout og appearance.
3. Ideelt er det udelukkende defineret i XAML, med begrænset "code behind" som *ikke* indeholder business logic.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

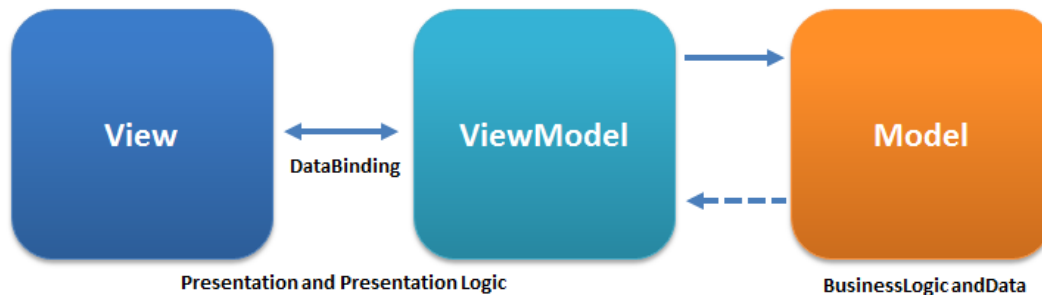


Figure 27: Sammenhæng mellem Model-View-ViewModel

4. Kan være en subcomponent af et andet view.
5. Kan have sin egen ViewModel eller arve dens parents.
6. Viewet får data fra ViewModel gennem ***databinding*** eller metodekald på ViewModel
7. View ændres ***run-time*** når UI controls, agerer efter ændringer i ViewModels properties.
8. Hvad sker der der laves User Input på View?

GUIPERSON:
what is
dis?

- Hvis control er en ***Command Source*** kan control's Command property være data bound til en ICommand property på view model.
- View er connected til ViewModel gennem databinding og sender commands to View-Model

8.3.2 ViewModel

1. **En model af Viewet.**
 - En abstraktion af viewet der leverer ***data binding*** mellem View og Model
 - En specialiseret udgave af MVP's presenter (agerer også ***converter*** der omdanner Model data til View data), og samtidig overfører commands fra View til Model.
2. ViewModel har ***public*** properties og commands.
3. Interagerer med Model gennem Databinding, properties og metodekald.
4. Modtager events fra Model

8.3.3 Model

1. Refererer som i MVC til enten:
 - Object-model, der repræsenterer state content.
 - Data Access layer der repræsenterer indhold.
2. **Kender ikke til ViewModel**
3. Sender evnets til ViewModel

GUIPERSON:
hvordan
sender den
events?

8.3.4 Commands

Data binding af View til Properties i ViewModel fungerer ganske fint og kan sagtens lade sig gøre. Data binding til funktioner er dog ikke muligt, selvom det ofte er til nytte. Data binding til funktionskald er f.eks. brugbart til en "Gem"-knap, som skal gemme en række data, når brugeren trykker på knappen. For at kunne data binde til en funktion kan der bruges "RelayCommand". RelayCommand er udviklet af Josh Smith. Et alternativ kan være "DelegateCommand" fra Prism frameworket. RelayCommand er brugt i GUI-kurset. RelayCommand fungerer således, at der laves en Command property, som View kan data binde til. Commanden "relayer" til en metode, og når en knap der er data bindet til en command bliver trykket på, så vil den pågældende metode blive kaldt. På denne måde kan der data bindes til metoder.

er databinding ikke ét ord?

hvordan kan noget en ordforklaring til dette afsnit ville være nice til os uden GUI.

"" antyder indforstået betydning, kender vi andre ikke noget til.

8.3.5 Oprettelse af Views og ViewModels

To måder:

1. **View First** - Viewet laves først og en ViewModel kobles efterfølgende på, ved at denne oprettes fra viewet. Dette kan gøres på forskellige måder – eksempler følger.
2. **ViewModel First** - ViewModel laves først, og det view der hører til ViewModel findes efterfølgende og vises til brugeren. Denne metode er lidt mere kompliceret. Eksempel følger.

```

1 public MainWindow()
2 {
3     InitializeComponent();
4     DataContext = new ViewModel(new Model)
5 }

```

Code listing 10: ViewModel oprettes i views codeBehind

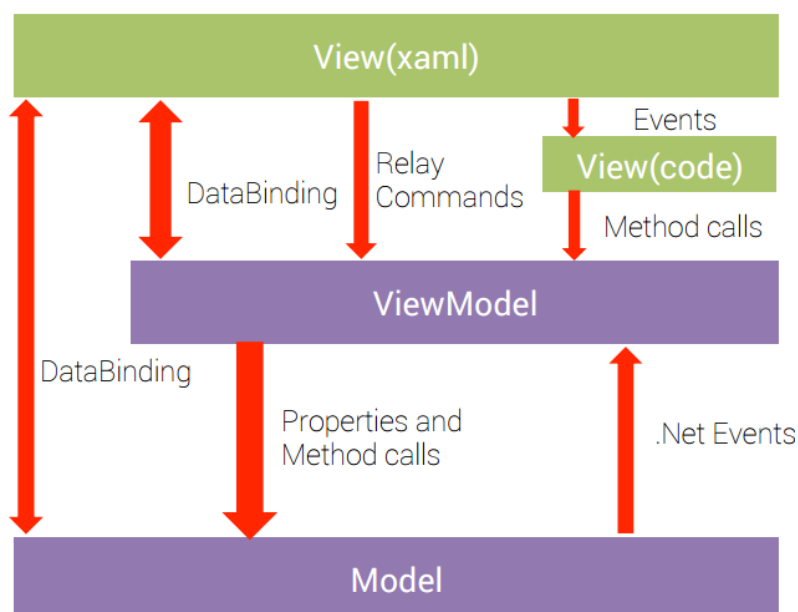


Figure 28: Intern adfærd i MVVM pattern

8.3.6 Hvorfor er det smart?

- **Test uden UI** - UI indeholder ikke business logic.
- Separation of Concerns.
- Vi kan uddelegere viewets implementering til en designer, hvorimod en programmør kan sørge for at implementere ViewModel

9 Patterns 7 - Redegør for følgende concurrency mønstre

9.1 Fokuspunkter

- Parallel Loops.
- Parallel Tasks.

9.2 Task Parallel Library

- Bibliotek i .NET.
- Man behøver ikke vide hvor mange kerner programmet skal køre på.
 - På general-purpose platforme kan programøren ikke forudsige hvor mange kerne/hvilken hardware der er til rådighed.
- TPL sørger for Dynamic Load Balancing.
- Der er ikke garanti for hvilken rækkefølge tasks afvikles i.

9.3 Parallel loops

Parallel loops pattern minder meget om et almindeligt loop. Der udføres samme operation på hvert element for et givet antal iterationer. Forskellen er dog at et almindeligt loop sker **sekventielt**, hvorimod et parallelt loop ofte udfører steps **parallelt**. Når man bruger parallelle loops er det derfor vigtigt at sikre sig at iterationerne ikke er afhængige af hinanden.

9.3.1 Hvornår giver det mening med Parallel Loops?

Det er rigtig dyrt at oprette og nedlægge tråde! **1MB Stack, 100.000 til 200.000 cycles** for construction/teardown(!).

- Som sagt kan det kun bruges hvor iterationerne ikke er afhængige af hinanden!
- Brug når iterationer ikke tilgår samme memory/filer.
- Brugbart hvor der laves blokerende kald (skriv/læs til fil).
- Brugbart når beregninger kan spredes ud på flere kerner.

Trådåndtering giver overhead på et parallelt loop, og derfor kan meget små iterationer blive kvalt heri. TPL håndterer partitionering, thread scheduling af tråde i threadPool'en og generelt alle low-level details.

```
1 // Dont use Parallel for this!  
2 Parallel.For(0, n, i => {  
3     a[i] = a[i-1] + a[i-2]; // Oh the pain!  
4 });
```

Code listing 11: Hvad Parallel.For ikke kan bruges til.

9.3.2 Oversubscription

Oversubscription er når vi opretter flere tråde end der er fysiske tråde tilgængelig. Dette leder til at OS bruger lang tid på at Context-Switche (takes time, kill caches).

9.3.3 Partitinerer

Partitionering er fordelingen af Tasks ud på trådene.

Statisk Trådene får et bestemt antal Tasks tildelt. Trådene kan ikke hjælpe hinanden. Hvis hver iteration har forskellig størrelse vil opgaverne blive ulige fordelt og vi kan risikere at en tråd bliver færdig lang tid før en anden, men ikke er i stand til at ”hjælpe” den anden tråd. Kræver ikke synkronisering.

Hvis vi skal balancere belastningen skal vi kende til opgavernes størrelse før ”execution”.

Dynamic Ved dynamisk load balancing kan en ”færdig” tråd hjælpe en stadig arbejdende tråd ved at tage nogle af dens opgaver. Dette **kræver dog synkronisering**.

Her behøver vi dog intet kendskab til eksekverings-tid.

9.3.4 Kodeeksempler

```
1 for (int i = 0; i < n; i++)
2 {
3     //Do stuff...
4 }
```

Code listing 12: Normal for loop

Parallelt Loop Eksempel på parallelt for loop: lambda expression

```
1 Parallel.For (0, EndValue, iterator =>
2 {
3     //Do stuff with 'i' ...
4 });
```

Code listing 13: Parallel for loop

Parallel ForEach-Loop For Each løkken kan også køres parallelt: lambda expression.

```
1 string[] navne = { "Torben", "Birger", "Niels" };
2
3 Parallel.ForEach(navne, navn =>
4 {
5     Console.WriteLine(navn);
6 });
```

Code listing 14: Parallelt for each loop

Det er Task Parallel Library der står for at håndtere threading, og man skal derfor ikke selv sørge for at fordele opgaven til CPU'en.

9.4 Parallel tasks

En task er ikke en tråd! En task er en opgave som udføres **sekventielt**.

Findes i *System.Threading.Tasks*.

TPL sørger automatisk for at scale graden af parallerisering så det er mest effektivt. Hjælper også til med scheduling af Thread Pool.

Vi kan bruge **Task Parallel Library** biblioteket til at køre disse tasks parallelt. TPL kan selv dynamisk scale graden af parallelisme sådan at det mest effektivt udnytter de kerner den har til rådighed.

9.4.1 Kodeeksempler

I kode udsnit 15 udføres en opgave på gammel sekventiel vis, mens listing 16 viser bruges af *Parallel.Invoke()*.

```
1 // Old sequential way of doing things
2 public void DoAll() {
3     DoLeft();
4     DoRight();
5 }
```

Code listing 15: Almindelig udførsel af opgave.

Invoke I eksemplet eksisterer to funktioner: *DoLeft()* og *DoRight()*. De to funktioner bliver afviklet asynkront vha. TPL. Dette er den mest simple måde at starte tasks på.

Invoke-metoden returnerer når alle tasks er afviklet.

```
1 // Simplest start af threads, vha invoke
2 public void DoAll() {
3     Parallel.Invoke(DoLeft, DoRight); // blocks till finished
4 }
```

Code listing 16: Parallel udførsel af opgave/task.

StartNew, Wait og WaitAll Ligeledes kan funktionerne *DoRight()* og *DoLeft()* udføres med *TaskFactory* som i listing 18.

```
1 public void DoAll() {
2     Task t1 = Task.Factory.StartNew(DoLeft);
3     Task t2 = Task.Factory.StartNew(DoRight);
4
5     Task t3 = Task.Run(DoStuff); // Simplere udgave af ovenstående.
6
7     // Enten det her...
8     Task.WaitAll(t1, t2);
9
10    // Eller det her
11    t1.Wait();
12    t2.Wait();
13 }
```

Code listing 17: Brug af *TaskFactory* - gør det samme som listing 16

WaitAny *WaitAny()* afventer, at mindst én task er blevet afviklet. Det kan benyttes til at udføre arbejde, mens der stadig ventes på, at andre tasks er blevet afviklet færdigt.

```
1 Task[] tasks = {
2     Task.Factory.StartNew(DoLeft),
3     Task.Factory.StartNew(DoCenter),
4     Task.Factory.StartNew(DoRight),
5 };
6
7 while(tasks.Length > 0) {
8     var taskIndex = Task.WaitAny(tasks);
9 }
```

forstår
ikke helt
linje 14
i det ne-
denstående.

```

10 // Do meaningful work with task[taskIndex] here.
11
12 // Update the set of tasks upon which we wait.
13 // Filter out the completed task (remove it from tasks-array).
14 tasks = tasks.Where((t) => t != tasks[taskIndex]).ToArray();
15 }

```

Code listing 18: Brug af WaitAny.

Giv en variable med til en *task* I eksemplet startes 4 tasks, som hver især udskriver deres index (0-3).

i kan ikke benyttes direkte i lambda-udtrykket, da den er global mellem alle fire tasks. Inden den første task går i gang med at afvikle kan *i* have værdien 3.

Løsning: Lav en midlertidig variabel, sæt den til at pege på *i*, og benyt den i stedet.

```

1 for (int i = 0; i < 4; i++)
2 {
3     var tmp = i;
4     Task.Factory.StartNew(() => Console.WriteLine(tmp));
5 }

```

Code listing 19: Giv en tråd en variable med.

Bemærk: Hvis der IKKE laves en midlertidig variabel, vil *i* være delt mellem alle tasks. Det kan i nogle tilfælde være nødvendigt.

Kør en *task* på et objekt Dette er en anden måde at give data med til en task.

Der afvikles en metode på et objekt. Metoden har adgang til alle attributter på objektet.

```

1 class Work
2 {
3     public int Data1;
4     public string Data2;
5     public void Run()
6     {
7         Console.WriteLine(Data1 + ": " + Data2);
8     }
9 }
10
11 public static void Main()
12 {
13     Work w = new Work();
14     w.Data1 = 42;
15     w.Data2 = "The Answer to the Ultimate Question of ...";
16     Task.Factory.StartNew(w.Run);
17 }

```

Code listing 20: Giv en tråd en variable med vha klasser.

9.5 Thread Pool

Måde at "opbevare" flere tasks, sådan at en færdig tråd altid kan hente nye opgaver.

9.5.1 Den simpel og dårlige løsning

Den simple og dårlige(!) løsning, som ikke benyttes af TPL.

- Naiv og langsom.

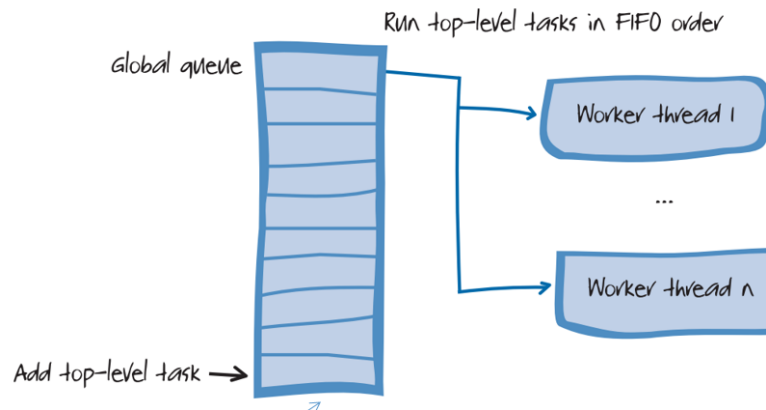


Figure 29: Simple implementering af threadpool

- Én global kø, som alle tråde henter jobs fra.
- Betyder at kun én tråd kan hente et job af gangen.
 - Køen bliver flaskehalsen.
 - Mange små tasks forværre problemet.
 - Køen bliver tilgået ofte, giver stort overhead.

9.5.2 Default Task Scheduler

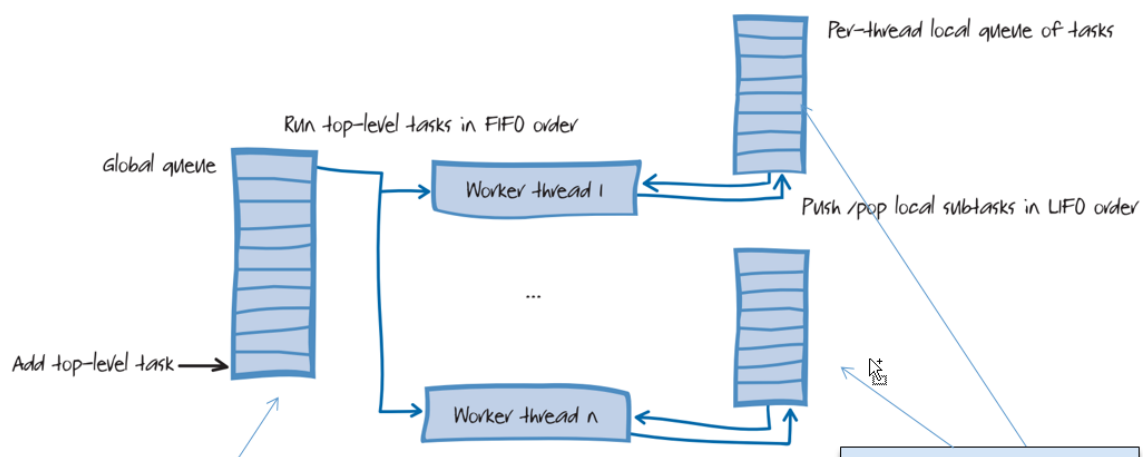


Figure 30: Faktisk implementering af threadpool, i .NET.

- En global kø.
 - Opbevare tasks som endnu ikke er delt ud på en tråd.
- Hver tråd har sin egen kø.
 - Tråde skal ikke vente på synkronisering med den globale kø.

- Er LIFO i private ende og FIFO i offentlig ende.
- Har en privat og offentlig ende.
- Når køen er tom, hentes tasks fra den globale kø.
- Work stealing.
 - Tråde kan ”stjæle” hinandens tasks.
 - Det sker når egen og global kø er tomme.
 - Tråde tilgår kun den offentlige ende af andres køer.
- Inline execution
 - hvis...
 - * *task1* venter på *task2*.
 - * *task2* er ikke startet når *task1* venter.
 - * Begge tasks er placeret i samme kø.
 - ... kan scheduleren.
 - * Give *task2* højere prioritet og afvikle den med samme.
 - * Så *task1* bliver hurtigere færdig.

10 Patterns 8 - Redegør for følgende concurrency mønstre

10.1 Fokuspunkter

- Parallel Aggregation.
- MapReduce.

10.2 Parallel Aggregation

I forlængelse af parallelle loop kan det sige at ikke alle PL's iterationer eksekveres uafhængigt. Fx. et loop der udregner en sum, bruger hver iterations resultat til at akkumulere en enkelt variabel der indeholder den udregnede sum op til det iterationsniveau vi er på. Denne akkumulerede værdi er en aggregation.

Man skulle dermed tro at at aggregation operation ikke kan forenes med *parallel loops*. Dog findes der en vej! Parallell Aggregation Pattern!

Dette pattern benytter *unshared* lokale variabler som merges til slut for at give det endelige resultat. PA kaldes også Parallel Reduction Pattern, idet den kombinerer flere inputs til et enkelt output.

10.2.1 Concurrent Collection in .NET 4.0

Vi får problemer hvis vi forsøger at tilgå en standard *List < Derp >* uden at bruge en locking-mechanisme. Se eksemplerne herunder:

```
1 // NO THREAD SAFE!
2 var output = new List<TOutput>();
3 Parallel.ForEach(input, item =>
4 {
5     var result = Compute(item);
6     lock(output) output.Add(result);
7 });
```

Code listing 21: Brug af ikke-trådsikker List

```
1 // ThreadSafe, Yay!
2 var output = new ConcurrentBag<TOutput>();
3 Parallel.ForEach(input, item =>
4 {
5     var result = Compute(item);
6     output.Add(result);
7 });
```

Code listing 22: Brug af trådsikker List

10.2.2 Eksempel med summering

Tag eksempelvis summeringen af nogle tal vist i ligning 1:

$$sum = a + b + c + d \quad (1)$$

Istedet for at løse dette sekventielt, som det i ligning 2 bliver demonstreret.

$$= ((a + b) + c) + d \quad (2)$$

Så kan vi ved at bruge Parallel Aggregation gøre som ligning 3 giver udtryk for.

$$= (a + b) + (c + d) \quad (3)$$

Den sidste metode vist i ligning 3 kan vi bruge parallelt og dette er meningen med Parallel Aggregation!

En illustrering af dette har vi i figur 32, hvordan man igen kan se hvordan paralleliseringen af løsningen kan være simpel og effektiv på samme tid.

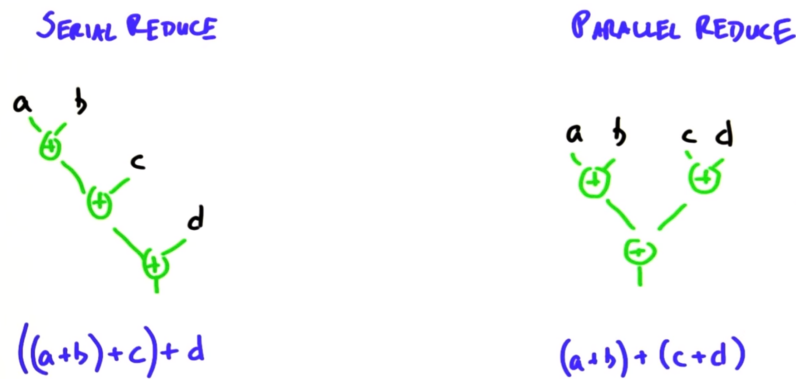


Figure 31: Forskellen på seriel og parallel reducering.

10.3 MapReduce

MapReduce er en måde at arbejde på meget store datasæt. Der arbejdes med MapReduce parallelt på dataen.

10.3.1 4 steps

Vi kan opdele MapReduce modellen i 4 steps som for vores eksempel bruges til at finde ud af hvor mange af hver kulør der findes i vores stak med spillekort.

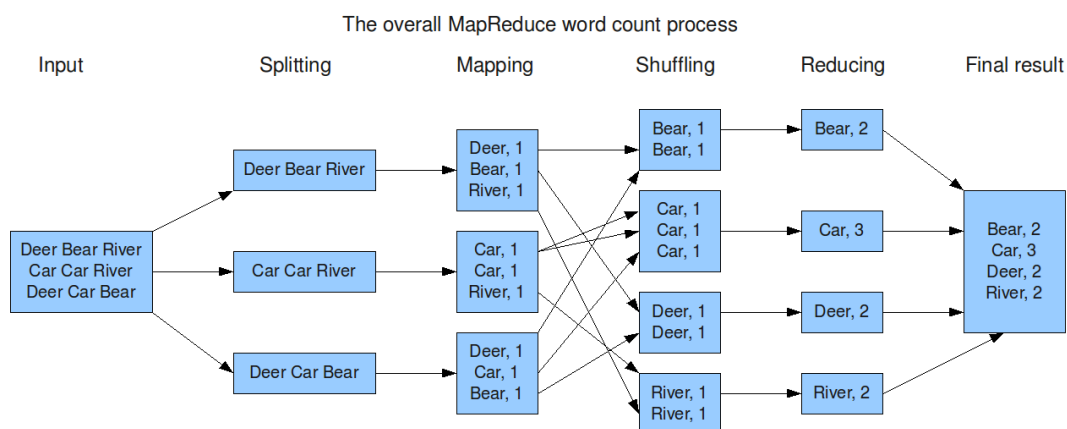


Figure 32: MapReduce illustreret.

1. **Distribuer source data til forskellige nodes.**

Fordel stakken af kort ud på alle noder.

2. **Map data - Repræsenter data i key-value par.**

Hver node tæller hvor mange af hver kulør den har fået tildelt. Dette repræsenteres med key-value par. eg. Kulør (key) og kortværdi (value).

3. **Gruppér data.**

Grouperen har til ansvar at gruppere mappernes key-value par. Her tages alle klør par og sættes i samme gruppe, alle ruder par sættes sammen osv.

4. **Reducer eller merge data fra grouperen.**

Her tælles sammen hvor mange af hver kulør der er i de grupperede data.

11 Patterns 9 - Redegør for følgende concurrency mønstre

11.1 Fokuspunkter

- Futures.
- Pipelines.

11.2 Future

Vi starter ud med lidt om **dependencies**:

Afhængigheder er et problem for parallelisering. Vi sorterer det der kan paralleliseres og gør det!

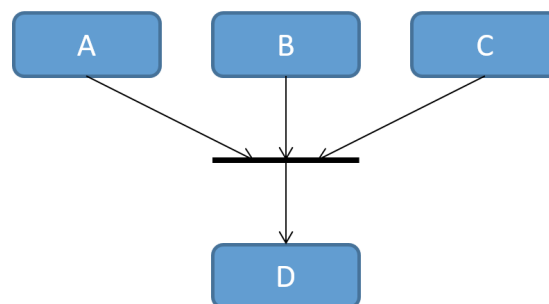


Figure 33: Node D skal køres før A, B og C kan køres parallelt

En future kan betegnes som en stand-in for en værdi der som udgangspunkt er utilgængelig, men som bliver tilgængelig på et senere tidspunkt. Altså bruges futures når vi ønsker at parallelisere kode der har data dependencies.

```
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);
}
```

Figure 34: Data dependencies

Vi laver en dependency til en future! Altså et løfte om at den kommer!


```

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);
}

```

Figure 35: Brug af Futures

Forskellen mellem futures og parallel tasks er at **Futures** er asynkrone funktioner der **returnerer** en værdi, hvorimod Parallel tasks **ikke returnerer** en værdi.

11.2.1 Fordele og ulemper

Man undgår at en funktion får et forkert input fordi en anden funktion ikke er færdig.

På den anden side er det den langsomste funktion der afgør hastigheden på den samlede udførselstid.

11.3 Pipelines

- Pipeline mønstret paralleliserer processeringen af en input sekvens.
- Pipeline mønstret består af en serie af producer/consumere.
- Opdeler processering i parallelliserbare stadier.
 - Output af stage i er indput i stage i + 1:
 - Andre stadier er uafhængige.

Som det kan ses på figur 36 har vi pipeline processeringen af C_1, C_2, C_3 og C_4

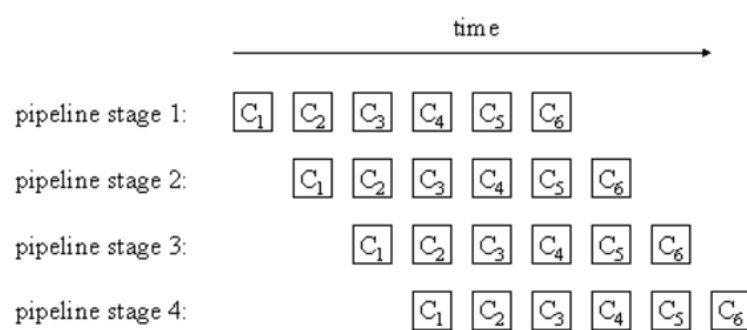


Figure 36: Brug af pipelining

Man kunne forestille sig at et enkelt stage vill tage længere tid end de andre (uneven stage duration).

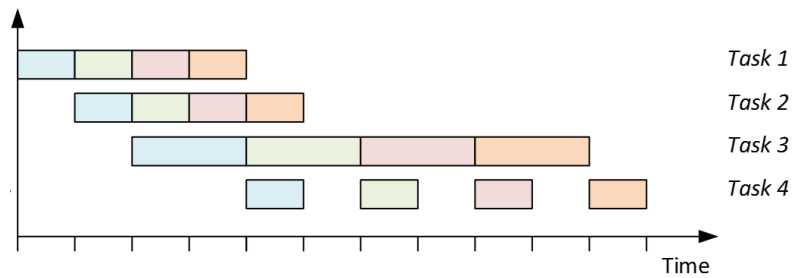


Figure 37: Standard pipelining med uneven stages

Løsningen på dette problem er at tilføje en ekstra stage 3

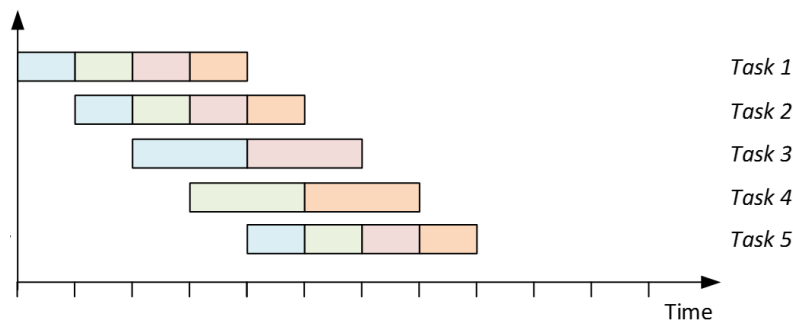


Figure 38: Pipelining med uneven stages + Ekstra stage

11.3.1 Fordele og ulemper

Pipelining kan fremskynde processing der har flere stages.

Det er dog værd at bemærke at pipelining ikke fremskynder den enkelte process, men derimod sikrer et større throughput. Hvis pipelining ikke er nædbendigt, kræver det blot flere ressourcer.

12 Software arkitektur

12.1 Fokuspunkter

- Redegøre for begrebet softwarearkitektur.
- Giv et eksempel på en typisk softwarearkitektur og dens anvendelse?
- Hvordan udarbejdes en software arkitektur?
- Hvordan dokumenteres en software arkitektur?

12.2 Redegør for begrebet softwarearkitektur

"The highest-level breakdown of a system into its parts; the decisions that are hard to change; ... and, in the end, architecture boils down to whatever the important stuff is."

12.2.1 Hvad er en software arkitektur?

Software arkitektur er processen hvor der defineres en struktureret løsning der imødekommer tekniske og operationelle krav, og samtidig optimerer kvalitet som performance, security, manageability osv.

Man kan sige at en SW arkitektur er den højeste abstraktion i et system.

- Fokuserer på de større elementer og komponenter.
 - Hvordan er de organiserede.
 - Hvordan interagerer de?
 - Hvad er vores "Areas of concern".
- Software Design er en realisering af arkitekturen bag.
- Arkitekturen viser de elementer som er svære at ændre.
- Viser også de elementer der er de vigtigste.

12.2.2 Hvorfor er det vigtigt?

Med komplekse systemer er det vigtigt at have et solidt fundament.

- Der er følgende risici uden en arkitektur:
 - Hovedscenarier kan overses.
 - Almindelige problemer bliver overset.
 - At værdsætte fremtidige konsekvenser på baggrund af afgørende beslutninger.
 - Uden en arkitektur, står man let med et monolithic program (big ball of mud), dvs. uden egentlig struktur og under-elementer.

12.2.3 Generelle overvejselser

- **Applikationstype** - Mobile app, web app osv.
- **Deployment strategi** - Distribueret/ikke distribueret, runtime kørsel, mnætværksinfrastruktur.
- **Teknologier** - Protokoller, Sprog, hvilken server, etc - Hvilke skal bruges?
- **Quality attributes** - Kvalitet af systemet.
 - Conceptual integrity.
 - Maintainability.
 - Reusability.
 - Performance.
 - Security.
 - Scalability.
 - Reliability.
- **Design kvalitet**
 - Supportability.
 - Testability.

12.2.4 Architectural Styles

Ligesom design patterns, findes der også patterns af arkitekturer, eller styles. En arkitektur stil definerer en familie af systemer med ens struktur, navne af komponenter og connectors og med restriktioner omkring hvordan disse kan forbindes. De blandes ofte (på sammen måde som design patterns) - der bruges næsten aldrig kun én style.

Typer af Arkitekturnal styles (almindelige)

- **Client/Server**
Serveren kører et eller flere programmer, som deler deres ressourcer med klienter.
- **Service oriented**
Det samme som Client/server blot med flere servers.
- **N-tier/3-Tier**
En Client/server arkitektur hvor presentation, applikation og processering er separeret. Tier (lag) opdeling. Den mest brugt er 3-tier.
- **Layered**
Den mest udbredte. Denne gives der eksempel på.
- Component based.
- Domain Driven Design.
- Message bus.
- Object Oriented.

12.3 Giv et eksempel på en typisk softwarearkitektur og dens anvendelse

12.3.1 Layer

- Et layer er en grovkornet gruppering af: *klasser, packages, og subsystemer*.
- Giver en forståelse af koblingsgraden i et system.
- Organisering af lag:
 - Højere lag kalder funktioner i lavere lag.
 - Lavere lag må ikke kalde på funktioner i højere lag. Lavere lag må dog godt notificere højere lag omkring hændelser med f.eks. observer pattern eller call-back funktions pointere (events).
- Layer vs Tier - Layer er en **logisk separering**, hvorimod Tier er en repræsentation af den **fysiske seaparering**.

Fordele ved Layer

- Separation af ansvar (SRP).
 - Separation af applikationsspecifik logik fra generel logik.
 - Separation af høj niveau fra lav-niveau logik.
- Reducerer kobling og afhængigheder.
- Større samhørighed.
- Højt potentiale for genbrug.
- Forholdsvist let at forstå/gennemskue.
- **Fremmer parallel udvikning** da systemet bliver delt op i logiske segmenter.
- Et lag kan udskrives hvis der er brugt interface baseret programmering.

Eksempel på layers Figur 39 viser hvordan en layeropdelt arkitektur kunne se ud for et system som skal have en GUI og Database med et Business Logik niveau imellem.

Ligeledes kan man på figur 40 se den samme lagdeling i netværks modellen OSI¹, som viser hvordan lagene imellem bruger hinanden.

12.4 Hvordan udarbejdes en software arkitektur

Der findes flere værktøjer:

- ROPES: **R**apid **O**bject-oriented **P**rocess for **E**mbedded **S**ystems.
Som navnet antyder er det primært til embedded software.
 - Har 3 faser:
 - * Architectural.
 - * Mechanistic.
 - * Detailed Design (implementation).

¹OSI: Open Systems Interconnection.

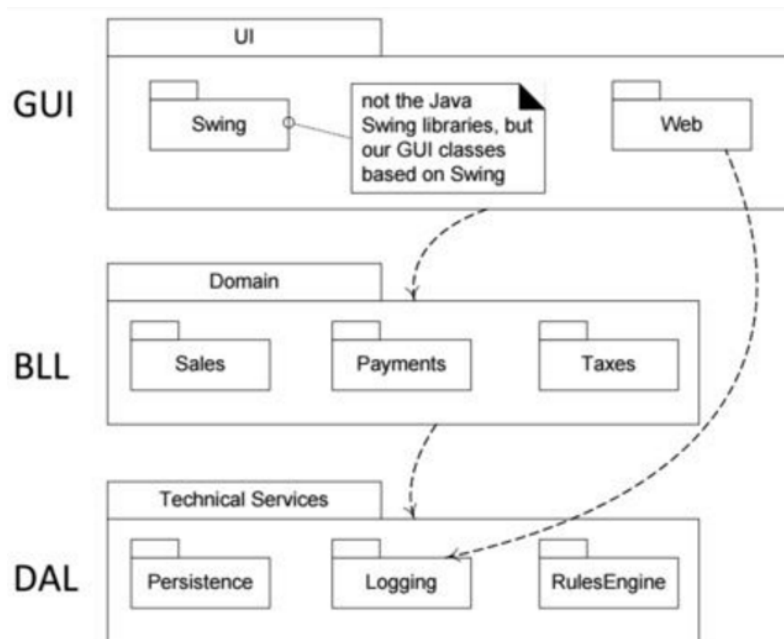


Figure 39: Arkitektur for et system med GUI, BLL og DAB logik.

- Iterativ (den vi mest bruger)
 - 5 punkter, hvor de sidste 4 gentages iterativt, illustreret i figur 41 og videre beskrevet i afsnit 12.4.1.
 1. Identify architecture objectives.
 2. Identify key scenarios.
 3. Create application overview.
 4. Identify key issues.
 5. Define candidate solutions.

12.4.1 Brug af Iterativ process

Følgende trin anvendes i udvikling af produkt med iterativ projekt.

Identify Architecture Objectives

- Klare mål hjælper med at fokusere på arkitekturen og løse de konkrete problemer i designet.
- Identifier arkitektur mål fra start.
- Identifier hvem der skal bruge arkitekturen, andre arkitekter eller udviklere og testere?
- Identifier dine begrænsninger.

Identify Key Scenarios

- Brug hovedscenarier til at fokusere dit design på hvad der betyder mest, og evaluer arkitektur kandidater herpå når de er klar.

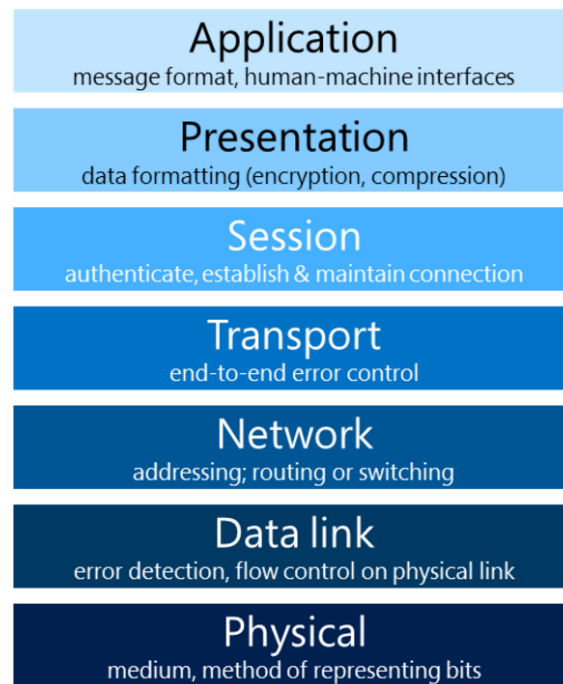


Figure 40: Netværksarkitektur, Open Systems Interconnection Model (OSI model).

- Hovedscenarier kan defineres som et scenarie der overholder en eller flere af følgende kriterier:
 - Repræsenterer et betydeligt ukendt område eller et område af betydelig risiko - bruges en ubekendt teknologi eller store "fare" forbundet med fejl på området.
 - Referere til en arkitektonisk vigtig use cases - de vigtigste.

Create Application Overview

- Identifier:
 - Applikations type - app, web, etc.
 - Deployment arkitektur - firma procedure og hvordan vi deployer systemet.
 - Arkitektur style (type), findes i afsnit [12.2.4](#).
 - Teknologier - valg af framework etc.

Identify Key Issues Identifier de største problemer baseret på kvalitets attributter og cross-cutting concerns. Begge nævn herunder.

- **Kvalitets attributter**
repræsenterer områder af bekymring der kan have potentielle følger på tværs af lag.
 - System kvalitet.
Den generelle kvalitet for systemet som hvor godt/nemt er det at supporte/teste.
 - Run-time kvalitet.
Kvaliteten af systemet der gives til udtryk run-time, som performance og scalability.

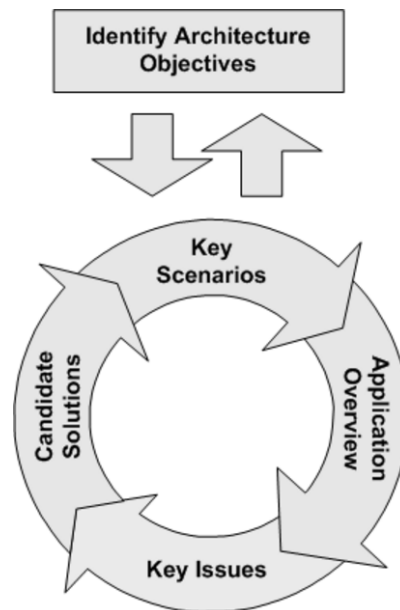


Figure 41: Process for iterativ arkitekturudvikling.

- Design kvalitet.
Kvaliteten der reflekteres i designet af systemet, som flexibilitet og genbrugelighed.
- Bruger kvalitet.
Hvor godt/nemt systemet er at bruge.

- **Crosscutting concerns**

- Grænseflader.
- Valg af metoder til Authentication, Caching, Kommunikation etc.

Define Candidate Solutions Skab en prototype der udvikler og forbedre løsningen og evaluer denne mod hovedscenarierne, problemerne og deployment begrænsningerne, før næste iteration af arkitekturen begynder.

- Få de største risici af vejen først.
- Lav iterative arkitekturelle tests der beviser at hver iteration er bedre end den forrige.

12.5 Hvordan dokumenteres en software arkitektur

Ved dokumentering af software arkitektur taler vi om to ting:

- UML.
- n+1.

Vi bruger disse værktøjer til at analysere vores system, samt reducere risiko.

12.5.1 UML og N+1

Ved design af arkitektur kan der anvendes N+1 og UML

- Task diagrammer, som udarbejdes vha. UML klassediagrammer.

Da der findes utallige måder at illustrere et system på, er n+1 (også kaldet 4+1), en bredspektret og mere struktureret måde at uddele forskellig funktionalitet i forskellige views.

Man behøver dog ikke bruge/vise alle views!

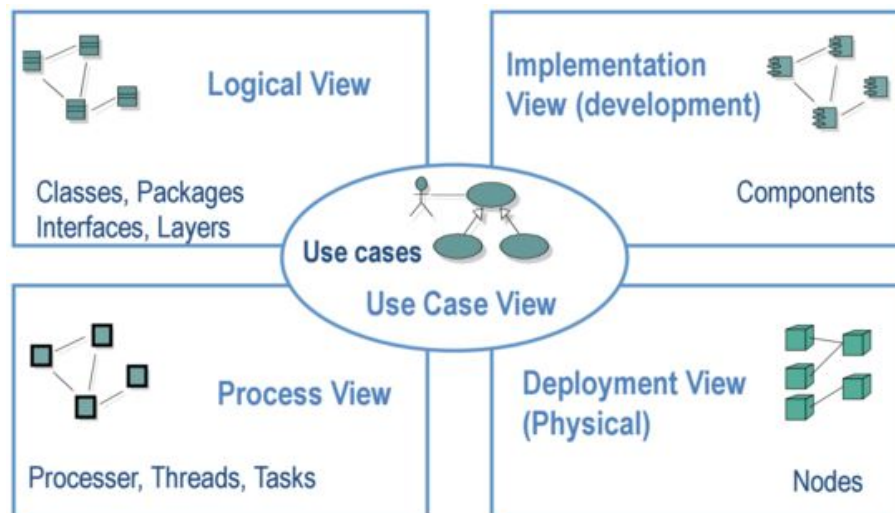


Figure 42: De forskellige N+1 views.

I forhold til hvilke diagrammer der bruges til de forskellige views, så viser figur 43 hvilke der kan bruges hvor.

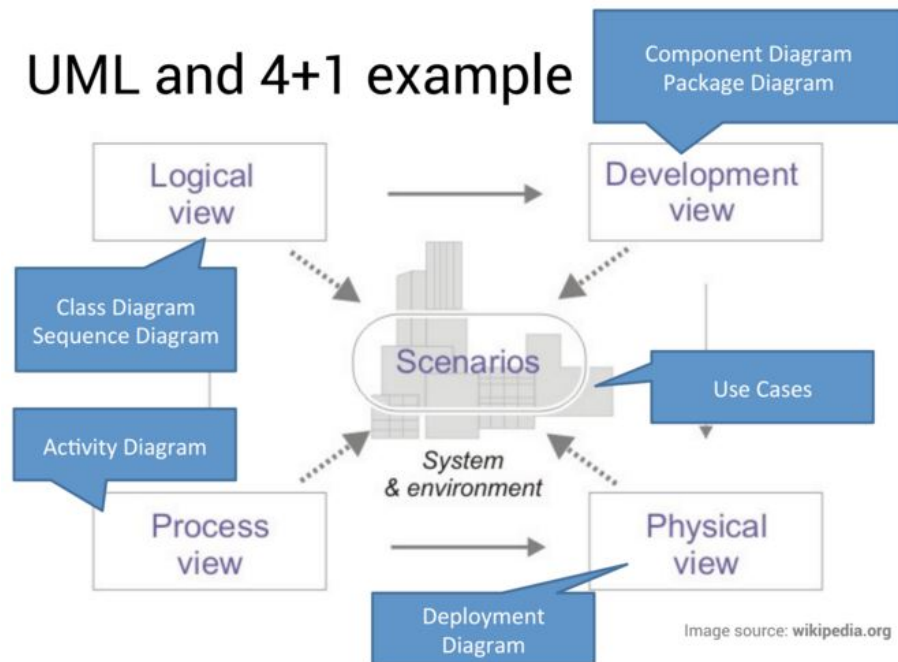


Figure 43: De forskellige N+1 views.

De fire views designs ud fra forskellige scenarier der udgør de 5. view "Scenarier". Disse er som regel systemets use cases.

De fire views som er vist i figur 42 er så:

- **Logical view** Systemets logik - hvordan kommunikere de forskellige klasser og komponenter.
 - Klassediagrammer.
 - Sekvensdiagrammer.
- **Process view** Systemets performance - forholder sig til concurrency problemer og viser hvor der kan opstå problemer med parallelitet.
 - Activity Diagrammer.
- **Development view** Systemets afhængigheder - fortæller hvordan de forskellige pakker/komponenter i systemet bliver ramt af ændring fra hinanden.
 - Komponent diagrammer (Systemets hovedkomponenter, Ordre, Account, Product, Customer osv).
 - Package Diagrammer.
- **Physical view** System lokalisering - hvor er de forskellige delsystemet og hvor skal der implementeres kommunikation.
 - Deployment diagrammer (Databaser, Servere osv. - nodes).

13 Obligatorisk Opgave (Mediator Pattern)

13.1 Valgte fokuspunkter

- Redegør for, hvad et Software Design er.
- Redegør for opbygningen af Mediator Pattern.
- Sammenlign de forskellige variationer af GoF Mediator - hvilke bør anvendes hvornår?
- Sammenlign de to Design Patterns GoF Observer og GoF Mediator - hvornår vil du anvende hvilket og hvorfor?
- Redegør for fordele og ulemper ved anvendelsen af GoF Mediator.
- Redegør for hvilke(t) SOLID-princip(per) du mener anvendelsen af GoF Mediator understøtter.

13.2 Hvad er et software pattern?

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design."

Det tilføjes at: *"A design pattern is not a finished design that can be transformed directly into source or machine code"*.

Og taget fra den danske wiki om [design pattern](https://da.wikipedia.org/wiki/Design_pattern)¹: "Design Pattern eller designmønster er en generel løsning på en problemtype, der ofte opstår i softwareudvikling. Et design pattern er ikke et endeligt design, der kan programmeres direkte; det er en beskrivelse eller skabelon for, hvordan man løser et problem i mange forskellige situationer.

En algoritme betragtes ikke som et design pattern, eftersom den løser et beregningsproblem og ikke et designproblem."

13.3 Redegør for opbygningen af Mediator Pattern

Når objekters funktionalitet distribueres ud mellem hinanden, vil der opstå høj kobling, og masser af interkonnektivitet. I et mediator pattern oprettes et separat mediator-objekt, som står for at kontrollere objekters interaktioner med hinanden.

¹Dansk wiki om design pattern:

https://da.wikipedia.org/wiki/Design_pattern

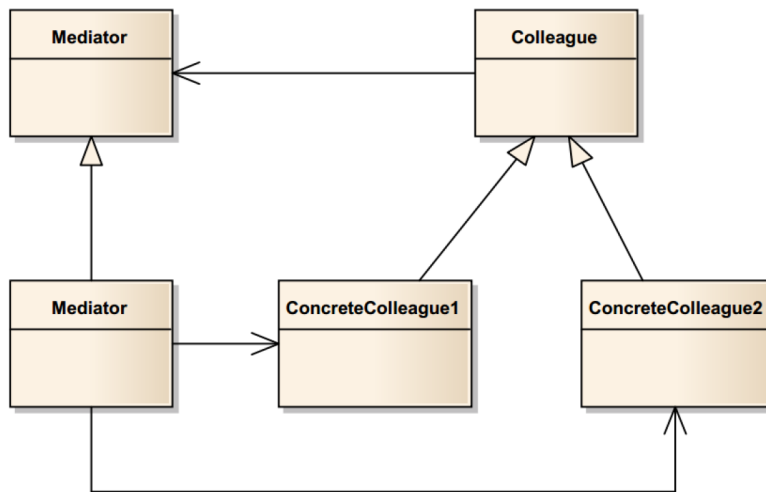


Figure 44: Et mediator pattern

Definition af mediators elementer

- BaseMediator
Definerer et interface til kommunikation med "Colleague objekterne".
- ConcreteMediator
Implementerer kommunikationsinterfacet, ved at koordinere Colleague objekter.
- Colleague Klasser.
 - Hver colleague klasse kender sit mediator object.
 - Når colleaguen vil snakke med en anden klasse, kommunikeres der udelukkende gennem mediatoren.

Brugen af et mediator pattern begrænser mængden af afedte klasser i et system, i og med mediatoren centraliserer funktionalitet der ellers ville være spredt ud på mange klasser. Ved at pakke objekters interkonnektivitet ind i et mediator pattern, får man samtidig skabt et ekstra abstraktionsniveau der gør funktionalitet mere overskuelig.

13.4 Sammenlign de to Design Patterns GoF Observer og GoF Mediator - hvornår vil du anvende hvilket og hvorfor?

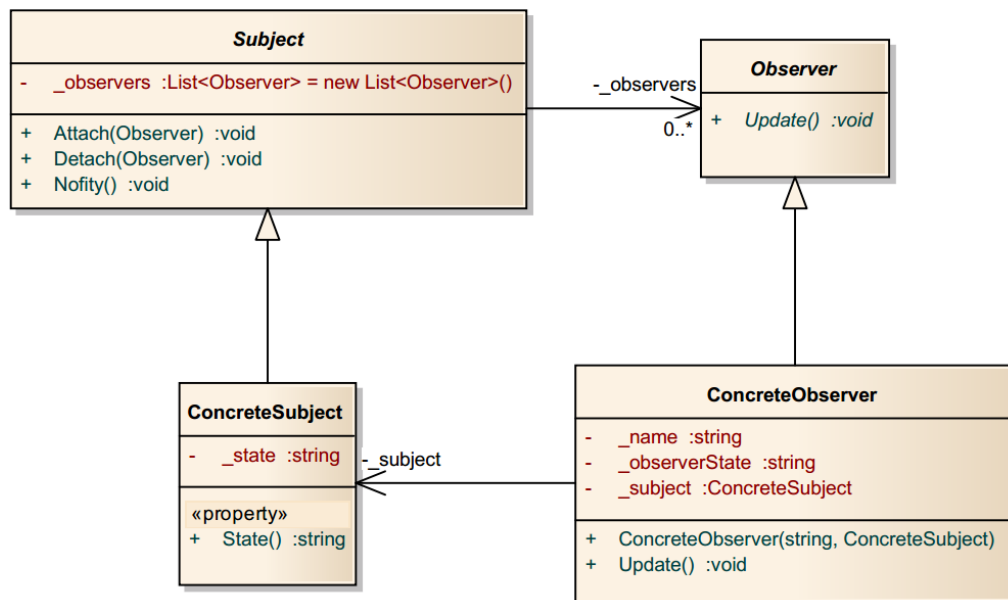


Figure 45: Klassediagram som viser GoF Observers opbygning.

- **Observer:** Flere observers der gerne vil subscribe på et subject.
- **Mediator:** Når klasser gerne vil kommunikere med hinanden.

13.5 Redegør for fordele og ulemper ved anvendelsen af GoF Mediator

13.5.1 Fordele

- Mindre kobling mellem participants.
- Simplificerede protokoller - simpelt at kommunikere med mediator.

13.5.2 Ulemper

- Mediatoren kan hurtigt blive kompleks. Det er derfor vigtigt at holde sig til mediatoren som besked interface.

13.6 Redegør for hvilke(t) SOLID-princip(er) du mener anvendelsen af GoF Mediator understøtter

- **OCP** - Let at tilføje participants og lave andre mediators.