

# UNIVERSITY OF AARHUS

Faculty of Science

Department of Engineering

## Eksamensdispositioner - Software Design

Bjørn Nørgaard  
IKT  
201370248  
bjornnorgaard@post.au.dk

Joachim Andersen  
IKT  
20137032  
joachimdams@post.au.dk

**Sidste ændring: December 11, 2015**

L<sup>A</sup>T<sub>E</sub>X-koden kan findes [her](#)<sup>1</sup>

---

<sup>1</sup><https://github.com/BjornNorgaard/I4SWD/tree/master/Eksamen/Disposition>

# Indholdsfortegnelse

<b>1</b>	<b>Solid 1 - SRP, ISP og DIP</b>	<b>1</b>
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? . . . . .	3
1.6	Eksempel . . . . .	3
1.7	Redegør for ulemper . . . . .	3
<b>2</b>	<b>Solid 2 - OCP, LSP og DIP</b>	<b>4</b>
2.1	Fokuspunkter . . . . .	4
2.2	Open-Closed Principle (OCP) . . . . .	4
2.3	Liskov's Substitution Principle (LSP) . . . . .	4
2.3.1	Pre -og postconditions . . . . .	4
2.3.2	LSP overholdt . . . . .	4
2.3.3	Brud på LSP . . . . .	5
2.4	Dependency Inversion Principle (DIP) . . . . .	5
2.5	Hvordan fremmes godt SW design? . . . . .	5
2.6	Eksempel . . . . .	5
2.7	Redegør for ulemper . . . . .	5
<b>3</b>	<b>Patterns 1 - GoF Strategy + GoF Template Method</b>	<b>6</b>
3.1	Fokuspunkter . . . . .	6
3.2	Hvad er et software pattern? . . . . .	6
<b>4</b>	<b>Patterns 2 - GoF Observer</b>	<b>7</b>
4.1	Fokuspunkter . . . . .	7
4.2	Hvad er et software pattern? . . . . .	7
<b>5</b>	<b>Patterns 3 - GoF Singleton + Method/Abstract Factory</b>	<b>8</b>
5.1	Fokuspunkter . . . . .	8
5.2	Hvad er et software pattern? . . . . .	8
<b>6</b>	<b>Patterns 4 - State patterns</b>	<b>9</b>
6.1	Fokuspunkter . . . . .	9
6.2	Hvad er et software pattern? . . . . .	9
<b>7</b>	<b>Patterns 5 - Model-View-Controller og Model-View-ViewModel</b>	<b>10</b>
7.1	Fokuspunkter . . . . .	10
7.2	Hvad er et software pattern? . . . . .	10
<b>8</b>	<b>Patterns 6 - Redegør for følgende concurrency mønstre</b>	<b>11</b>
8.1	Fokuspunkter . . . . .	11
<b>9</b>	<b>Domænemodeller og Domain Driven Design</b>	<b>12</b>
9.1	Fokuspunkter . . . . .	12
<b>10</b>	<b>Software arkitektur</b>	<b>13</b>
10.1	Fokuspunkter . . . . .	13

# 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 bogen<sup>1</sup>. Skal vi dele modem klassen op? Det kommer an på hvordan applikationen ændrer sig. Hvis connection-delen ændres skal resten af klassen 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.

---

<sup>1</sup>Bogen til kurset: Agile Principles, Patterns, and Practices in C#

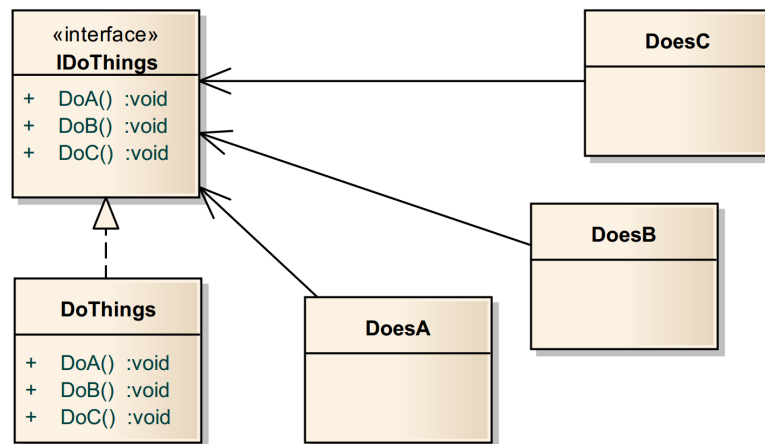


Figure 1: Flere klienter afhængige af samme "store" interface.

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.

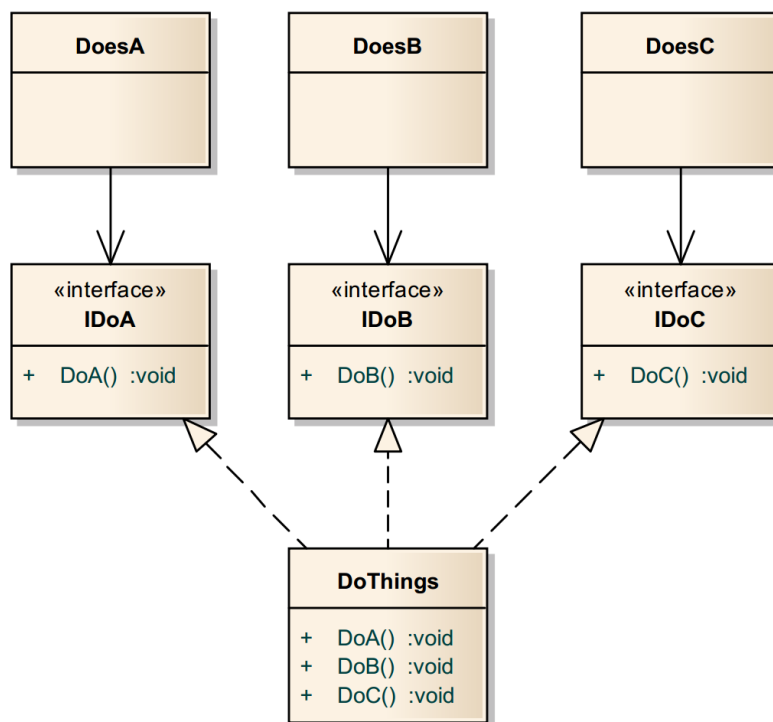


Figure 2: ISP anvendt på eksempel fra figur 1.

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 der menes med abstraktioner her, er interfaces. Et eksempel på DIP er vores øvelse med Compressions Stocking, hvor vi fx havde nogen knapper (high-level) der kaldte noget low-level funktionalitet (blinkende LED'er osv). I stedet for at gøre knappen afhængig af low-level funktionalitetet, lader vi den afhænge af et interface. Klassediagrammet for øvelsen kan ses på dette

## 1.5 Hvordan fremmes godt SW design?

### 1.6 Eksempel

### 1.7 Redegør for ulemper

## 2 Solid 2 - OCP, LSP og DIP

### 2.1 Fokuspunkter

- Redegør for:
  - Open-Closed Principle (OCP).
  - Liskov'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".*

### 2.3 Liskov's Substitution Principle (LSP)

*"Subtypes must be substitutable for their base types".*

#### 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 }
```

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)

## 2.5 Hvordan fremmes godt SW design?

## 2.6 Eksempel

## 2.7 Redegør for ulemper

## 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](https://da.wikipedia.org/wiki/Design_pattern)<sup>1</sup>: "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."

---

<sup>1</sup>Dansk wiki om design pattern: [https://da.wikipedia.org/wiki/Design\\_pattern](https://da.wikipedia.org/wiki/Design_pattern)



## 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)<sup>1</sup>: "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."

---

<sup>1</sup>Dansk wiki om design pattern: [https://da.wikipedia.org/wiki/Design\\_pattern](https://da.wikipedia.org/wiki/Design_pattern)

## 5 Patterns 3 - GoF Singleton + 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.
- Redegør for opbygningen af GoF Singleton.
- Redegør for fordele og ulemper ved anvendelsen af GoF Singleton

### 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)<sup>1</sup>: "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."

---

<sup>1</sup>Dansk wiki om design pattern: [https://da.wikipedia.org/wiki/Design\\_pattern](https://da.wikipedia.org/wiki/Design_pattern)

## 6 Patterns 4 - State patterns

### 6.1 Fokuspunkter

- Redegør for, hvad et software design pattern er.
- Redegør for de forskellige måder at implementere en state machine på.
- Redegør for opbygning af 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](https://da.wikipedia.org/wiki/Design_pattern)<sup>1</sup>: "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."

---

<sup>1</sup>Dansk wiki om design pattern: [https://da.wikipedia.org/wiki/Design\\_pattern](https://da.wikipedia.org/wiki/Design_pattern)

## 7 Patterns 5 - Model-View-Controller og Model-View-ViewModel

### 7.1 Fokuspunkter

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

### 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)<sup>1</sup>: "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."

---

<sup>1</sup>Dansk wiki om design pattern: [https://da.wikipedia.org/wiki/Design\\_pattern](https://da.wikipedia.org/wiki/Design_pattern)

## 8 Patterns 6 - Redegør for følgende concurrency mønstre

### 8.1 Fokuspunkter

- Parallel Loops
- Passing data
- Producer/consumer
- Mapreduce
- Shared state

## 9 Domænemodeller og Domain Driven Design

### 9.1 Fokuspunkter

- Hvad er en domændemodel?
- Hvordan dokumenteres den?
- Hvad bruges domænemodellen til?
- Hvilke metoder kan man bruge til at finde de konceptuelle klasser?
- Redegør for begrebet Domain Driven Design.

## 10 Software arkitektur

### 10.1 Fokuspunkter

- Redegøre for begrebet softwarearkitektur.
- Hvordan er den typiske software arkitektur?
- Hvordan udarbejdes en software arkitektur?
- Hvordan dokumenteres en software arkitektur?
- Hvorledes udarbejdes og dokumentes en concurrency model?