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: December 12, 2015

LaTeX-koden kan findes her[1]

[1] https://github.com/BjornNorgaard/I4SWD/tree/master/Eksamen/Disposition

Todo list

Indholdsfortegnelse

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 redeploy* 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[1]. Skal vi dele modem klassen op? Det kommer an på hvordan applikationen ændrer sig. Hvis connection-delen ændres skal resten af klassen også rekompilere.

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.

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

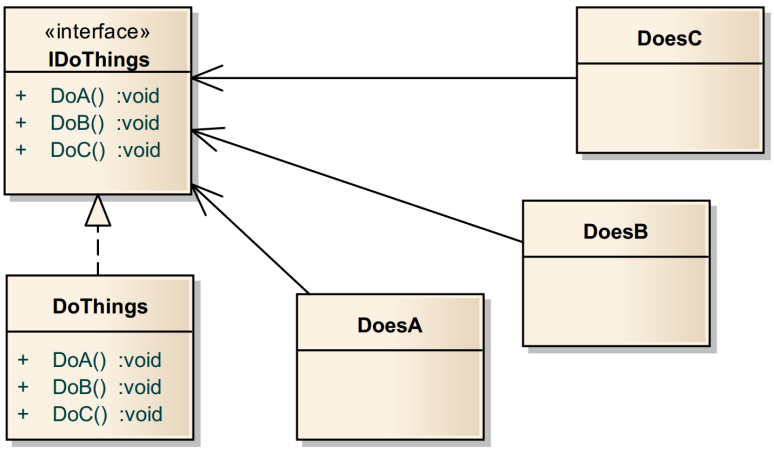

Figure 1: Flere klienter afhængige af samme "store" interface.

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.

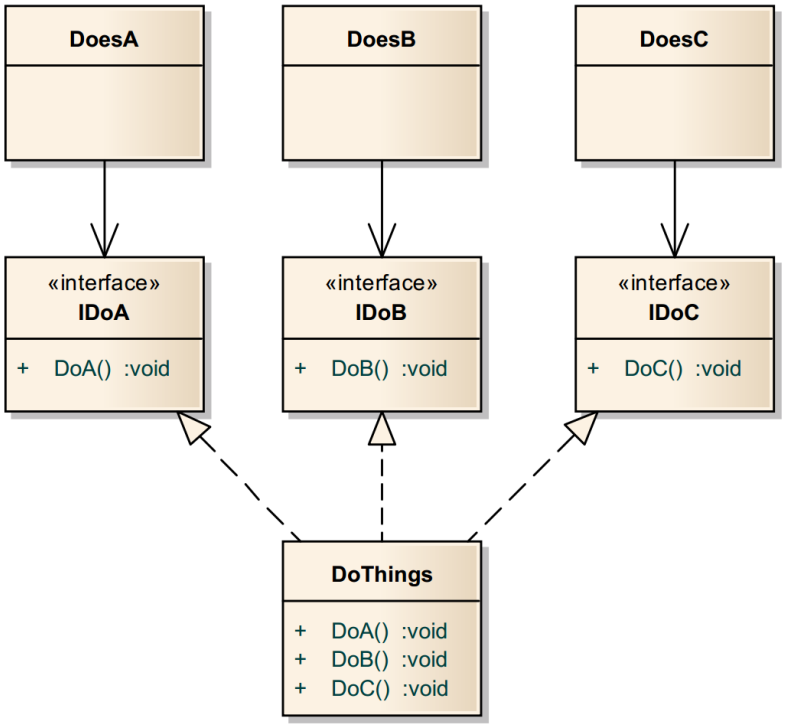

Figure 2: ISP anvendt på eksempel fra figur 1.

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 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 insersion fordi...

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 kan ikke være på én side så derfor kan billedet for øvelsen kan ses på dette billede fra github repo'et.

hør fil 1441956 389456 og skriv hvor navnet kommer fra

1.5 Hvordan fremmes godt SW design?

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

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 genkompileres, gentestes 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.

Et personligt eksempel er i faget ISU, hvor vi distribuerede et parkeringssystems funktionalitet ud på enormt mange små klasser, med enkelte metoder/attributter. Dette medførte en meget uoverskuelig og kompleks kode, og det positive ved SRP opvejede ikke længere unødig kompleksitet.

skal med? måske for tidskrævende?

1.5.2 ISP

1.5.3 DIP

1.6 Eksempel

1.7 Redegør for ulemper

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 problemer[1] i resten af programmet.

Når OCP er "well applied" betyder det at vi kan tilføje ny kode uden at behøve ændre i det gamle der i forvejen 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 ikka at røre ved exekverbare filer, DLL'er og library filer.

2.2.1 HOW TO OCP

- Brug abstrakte base klasser. — hvorfor base?
- Eksempel - Client der bruger Server.
- To konkrete klasser er lort $\longrightarrow$ brug et interface (client). — why it says client?

2.2.2 Perspektiver

Strategy pattern Herp derp something something... — her skal stå mere...

[1] rekompilering, retest, redeploy.

2.3 Lisskov'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 **throwExeption**) vil bryde OCP! Et //Do Nothing er sikkert fint men ikke særligt sikkert. Altså er en Tesla ikke en bil i Barbara Liskovs optik.

hvorfor er dette i parentes?

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 }
```

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 føglende 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 Templete 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."

[1] Dansk wiki om 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 undersøtter.

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[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."

[1] Dansk wiki om 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[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."

[1] Dansk wiki om 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[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."

[1] Dansk wiki om 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[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."

[1] Dansk wiki om 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?