2. SOLID 2: OCP, LSP og DIP

Redegør

I software design bruger vi 5 grundprincipper til at designe oo software ud fra de hedder: Single Responsibility Principle, Open/Closed Principle, Liskov’s Substitution Principle, Interface Segregation Principle og Dependency Inversion Principle. De giver alle et godt sted at starte og vil i sidste ende gøre livet lettere for os når vi laver software

Open-Closed Principle

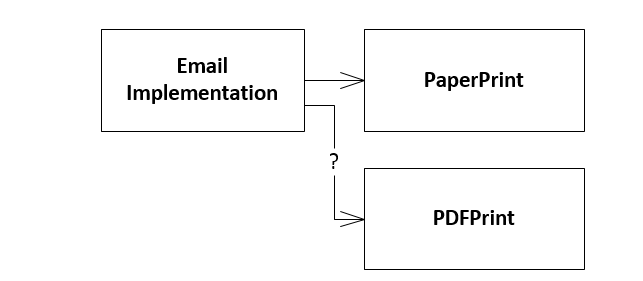
Open-close pricippet siger at software enheder (klasser, funktioner, metoder) skal være åbne for udvidelser, men lukket for modifikationer.

*Åben for udvidelser*: Man skal forvente nogle krav om ændringer, og disse ændringer skal implementeres ved at udvide det eksisterende design.

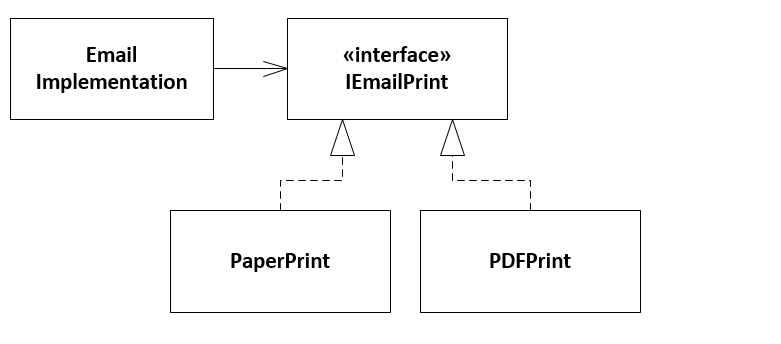
*Lukket for modifikationer*: Når krav bliver ændret, burde det ikke være nødvendigt at ændre den eksisterende implementering a klienterne.

Eksempel

På eksemplet ses et eksempel på et problem der kan komme hvis OCP ikke er overholdt.



Løsningen er at indsætte et interface IEmailPrint, da vi på den måde kan tilføje flere printere uden at røre ved Email Implementation



Pros/cons

**Pros:**

* Fleksibelt (løser problemet med stiv kode)
* Nemt at holde ved lige
* Nemt at udvide

**Cons:**

* Hvis man tilføjer det alle steder bliver det unødvendig kompleksitet

Lisskov substitution Principle

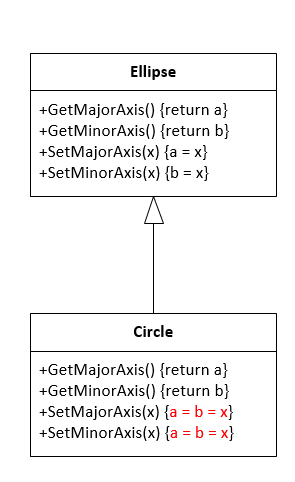
Lisskov substitution princip fortæller om at substituere en superklasse med en af dens underklasser. Dette kræver dog at underklasserne har samme adfærd som superklassen, ellers risikerer vi fejl når der skal anvende dette, da vi forventer en anden adfærd.

Funktioner der bruger en reference til en baseklasse skal også være i stand til at tage en nedarvet klasse uden at vide det.

Eksempel

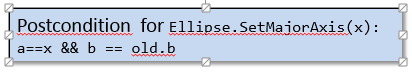
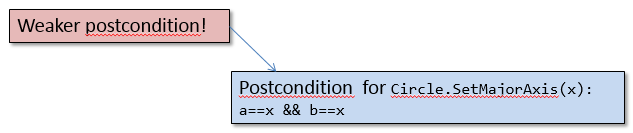
Et eksempel kunne være fugl (layEgg()), så kan alle fugle være med, men hvis den har metoden fly kan den ikke tage en struds med.

Et eksempel kan være Ellipse og cirkel. En cirkel er en udgave af en ellipse og br derfor kunne substistueres. Hvis man i en ellipse ændrer den ene akse, ændres den anden også. Den samme funktionalitet kan ikke findes i en cirkel og derfor kan vi ikke substituere disse.



**præ- og postkonditioner**Der er nogle krave for subklassernes præ- og postkonditioner som skal være overholdt hvis disse skal substitueres med deres superklasser

* **Prækondition**: En prækondition skal være lig eller svagere superklassens prækondition.
* **Postkondition**: En postkondition skal være stærkere eller lig superklassens postkondition

Pros/cons

**Cons:**

* Hvis der ikke er brug for det kan man risikerer unødvendig kompleksitet

**Pros:**

* Test af superklasser vil virker på dens underklasser
* Giver et godt klasse hierarki
* Det er letter at overholde OCD hvis LSP er overholdt

Dependency Inversion Principle (DIP)

Dependency Inversion Principal (DIP) handler om:

* Høj - niveau klasser må ikke være afhængige af lav – niveau klasser, Begge skal være afhængige af abstactions.

# Dependency Inversion Principle

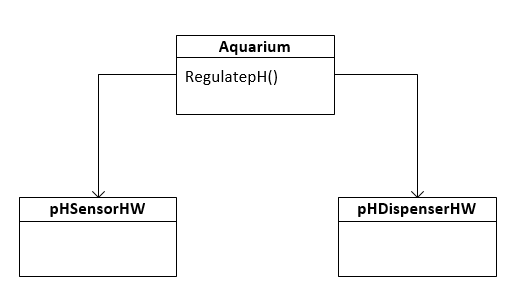
Dependency Inversion Principal (DIP) handler om:

* Høj - niveau klasser må ikke være afhængige af lav – niveau klasser, Begge skal være afhængige af abstactions.
  + Høj - niveau klasser er abstrakte fra detaljer (kommunikation, hardware mm.) og indeholder ting som business modeller.
  + Lav – niveau klasser (drivers), kender til detaljer om HW, og kommunikation, men intet om Høj – niveau.
* Abstraktioner skal ikke være afhængige af detaljer. Detaljer skal være afhængige af abstraktionerne

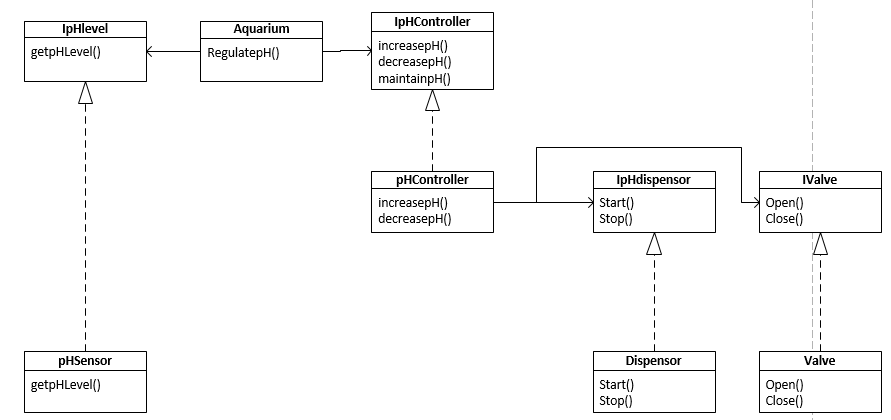
Eksempel

Eksemplet her viser et pH-regulerinssystem som har en pH-sensor og en dispenser,

* ”højniveau klasser må aldrig afhænge af de nedre, men altid en abstrakt klasse”
* ”abstraktioner bør aldrig afhænge af detaljer. Detaljer bør afhænge abstraktioner”



I eksemplet her under løser vi løst vores inversion problem. Der er lagt interfaces ind og Aquarium er ikke længere afhængige af detaljerne fra de andre klasser, så den er afhængig af hvad den skal gøre og ikke kun hvordan den skal gøre det.



Low-level

Mid-level

High-level

Pros/cons

# Bad design

**Cons:**

* Hvis det kunne tænkes at der er dele der aldrig skulle være noget andet, så er det unødvendigt kompleksitet.

**Pros:**

* Nemt at udskifte i da både høj og lav moduler afhænger af abstraktioner og ikke hinanden
* Rigidity (Stivhed)
  + En simpel ændring kan gøre at der skal ændres meget i alt koden.
* Fragility (Skrøbelighed)
  + En ændring skaber defekter andre sted i programmet når disse fikses skabes nye defekter
* Immobility (ubevægelighed)
  + Systemet kan kun bruges i en enhed og dele kan ikke separeres
* Viscosity (Viskositet)
  + Når det er nemmere at gøre tingene på den dumme måde i stedet for at redesigne
* Needless Complexity
  + Når systemet tager højde for fremtidige ændring der måske i sidste ende aldrig bliver behov for.
* Needless Repetitions
  + Copy paste er lort!