SOLID 1: SRP, ISP og DIP

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.

# Bad design

* 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!

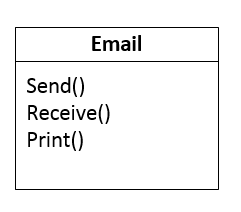
# Single Responsibility Principle

”A class should have only one reason to change” er definitionen på SRP og hvad betyder det så?

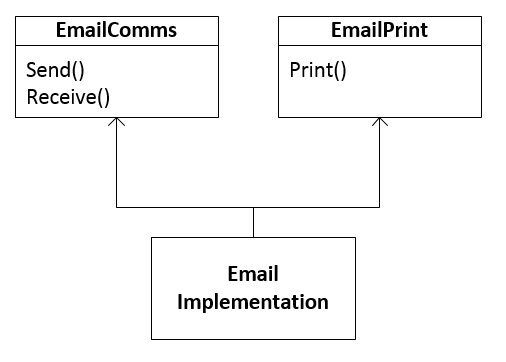
Det betyder at hvis en klasse har ansvar for mange ting kan den have mange grunde til at ændre sig. Det er vi ikke glade for da det giver stivhed i koden.

Eksempel

Her ses E-mail klasse, denne klasse står både for e-mail kommunikations men den kan også printe en e-mail ud, dette er et brud på SRP, da det giver klassen mere end et ansvar.



Hertil vil printfunktionen lægges ud i sin egen klasse som på figuren:



Fordelen ved at separerer opgaven er at vi undgår overfyldte klasse, men også at ændringer i klassen ikke påvirker for meget i systemet.

Pros/cons

**Pros:**

* Mere læselig kode
* Mere maintainable
* Mere robust kode

**Cons:**

* Man nemt komme ud for mange klasser. Det kan give unødvendig kompleksitet, da mange klasser skal til at snakke sammen.
* Man bevæger sig væk fra real-world forståelsen af et objekt. I virkeligheden vil et objekt have mange ansvar.

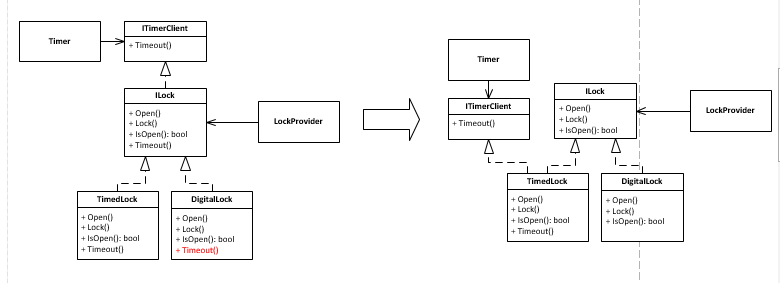
# Interface Segregation Principle

”Clients should not be forced to depend on methods they do not use” er grund sætningen for ISP.

I bund og grund går det ud på I stedet for at udstille et stort interface til alle klienter så er det bedre at stille forskellige interfaces til rådighed for de forskellige klienter.

Eksempel

På dette eksempel ses at ILock implementere ITimerClient, men da det ikke er alle lock der giver bruger en timer er der en der har en unødvendig metode



Dette gøres ved at det kun er det kun er de locks der har brug for timer som implementere dem.

Pros/cons

**Cons:**

* Det give mange klasser og igen medføre unødvendig kompleksitet.

**Pros:**

* Udstiller de funktioner der er behov for, og det kan også give mindre forvirring internt i koden, hvis man kun udstiller de funktioner som man mener andre programmører på sit team bør kalde.

Take the first bullet!

# Dependency Inversion Principle

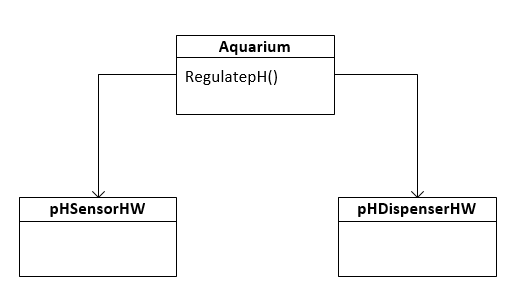
Dependency Inversion Principal (DIP) handler om:

* 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

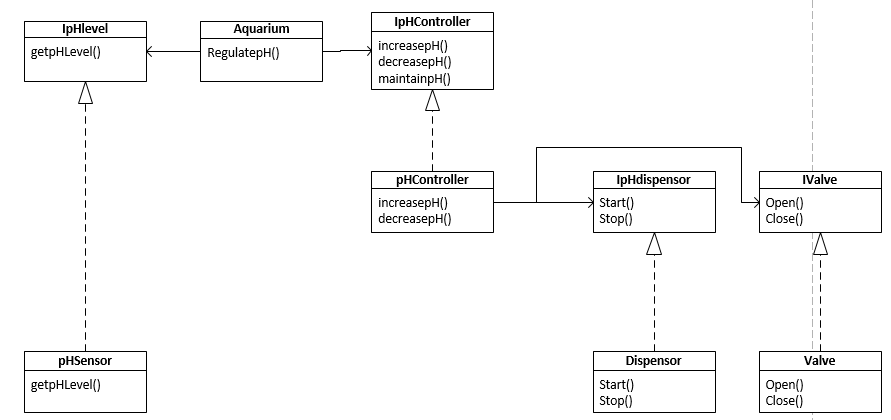
Eksempel

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

**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