

1 Design

Design er ikke diagrammer. Disse kan bruges til at forstå og kommunikere det.

Source code er det egentlige design

1.1 Agile Design

- Lad vær at tegne UML diagrammer i ugevis, men tænk over hvad du laver
- Dit projekt udvikler sig
- Arbejd i iterationer
- Lav et design for hver iteration
- Husk, at du bliver klogere undervejs i projektet - ændringerne må gerne indføres

1.2 Dårligt design

- Svært at ændre
- Arkitektur låst til én platform
- Ændringer skal ske for mange steder
- Overdesignet
- For mange gentagelser (underdesignet)
- Kan nemt gå i stykker
- For kompliceret til, at se hvor ændringer skal foretages

Ingen laver bevidst dårlig kode, men det kan forekomme, fordi en ny mand kommer på teamet, og måske ikke forstår filosofien af, hvordan koden skal håndteres. Tiden gør også, at man tænker anderledes.

Fejl introduceres over tid.

1.3 SOLID

Idéen er, at det skal bruges, når koden begynder at stinke. Undlad, hvis der ikke er. Det kan føre til overdesign

- SRP (Single Responsibility Principle)
 - *"A class should have only one reason to change"*
 - Den skal kun køre det, den er designet til
 - Den er også med til at holde lav kobling, da den ikke er afhængig af så mange andre
 - Alle klasser afhængig af klasse A, bliver påvirket, når klasse A ændres
- OCP (Open/Close Principle)
 - Man kan udvide koden, uden at ændre koden.
 - Dette gøres ved at kode op imod et interface
 - Således kan det skiftes ud med andre implementeringer
 - Et modul er dog ikke altid lukket helt
 - * Hvis et interface skal ændres, skal koden også ændre

- Kodens kan dog blive for lukket og kompliceret, hvis man gør for meget med det.
- LSP
 - En nedarbejde klasse skal overholde interfacet. Dvs, at de skal overholde de regler, der er stillet for interfacets funktioner.
- ISP (Interface Segregation Principle)
 - Der er flere funktioner i et interface, end det der bruges
 - Man skal ikke tvinges til dette og unyttige funktioner
 - Et interface skal have SRP til sig.
 - Har man implementering man ønsker at lave mange gange, så lav en adaptor.
- DIP (Dependency Inversion Principle)
 - En klasse er afhængig af en bestemt implementering
 - Dette løses med indirection som en implementation fra et interface

```

1  void doStuff(IStuffToDo obj)
2  {
3      obj.Stuff1();
4  }
5
6  class SomeStuff : IStuffToDo
7  {
8      public Stuff1();
9      ...
10 }
11
12 class OtherStuff : IStuffToDo
13 {
14     public Stuff1();
15     ...
16 }

```

2 Plugins

2.1 Regler

- REP (Reuse/Release Equivalency Principle)
 - Hvis du laver tingene for små, kommer der mange små opdateringer
 - Det kan give en masse administrationsarbejde
- CRP (The Common Reuse Principle)
 - Du du pakker sammen, skal bruges samme
- CCP (Common Closure Principle)
 - Hav god overensstemmelse med virkeligheden, da dette giver bedre implementering.
 - De må dog ikke blive for store (altså ikke database, samt kommunikation i én pakke)
- ADP (The Acyclic Dependencies Principles)

- Du må ikke have dependency løkker
- SDP (The Stable Dependencies Principle)
 - Afhængigheden skal gå i retningen af stabiliteten
 - Ting nederst i et hierarki skal være stabile - ellers skal de have et interface.
- SAP (The Stable Abstraction Principle)
 - Hvis mange er afhængige, skal de være abstrakte