Komponentbaseret Program Udvikling

Rasmus Bækgaard, 10893, IKT

1 Lektion 1

- WIN RT stammer fra både COM og .NET teknologien

1.1 Komponenter

- Komponenter er udskiftelige.
- Kan interagere med hinanden.
- Kommer af "komponere"altså sætte sammen med andre komponenter.
- Forskellige producenter kan lave en enhed, sålænge de har det samme interface.
- På denne måde er det nemmere at lave nye ting.

De var oprindeligt tænkt som hardwarekomponenter, da man havde lavet hardware der kunne yde meget. Man manglede desværre software, der kunne udnytte hardwaren.
Man lavede derfor nogle små dele (komponenter), der kunne bruges til noget af hardwaren.

Eftersom komponenter kan genbruges, kan de sælges til andre. Dette kan også reducere udviklingstiden.
En grov regel siger, at den skal kunne genbruges 3 gange, før det er tidsbesparende - men dette er måske lidt outdatet.

Bruges gerne på store projekter, da det er nemmere for andre folk ikke at se på source kode, men på et interface og magi. Det kræver derfor, at man får lavet nogle ordenlige interfaces.

1.2 Load-time Dynamic Linking

Udviklingen

- Man skriver et program (gerne i flere sourcefiler)
- En compiler laver objekt-filer
- En linker laver binære .exe-filer og .dll-filer
- Ved at køre .exe-filen siger den "jeg skal bruge en .dll-fil"
- Den leder så efter en .dll med et bestemt navn - her kan man indsætte sin egen.

Fordele:

- Dynamic linking sparrer plads
- Skal ikke rekompileres

1.3 Run-time Dynamic Linking

- Her fortælles hvilken dll, der ønskes loaded.
- Når loaded har man en klub kode
- Så kaldes "GetProcAdrdress", hvilket giver os en funktions pointer"

1.4 DLL typer

Traditionelle C-Style Win32 DLL'er:

- Standard som er en del af Windows.
- Bruger lavet

COM DLL'er:

- Indeholder kun 4 funktioner

.NET DLL:

1.5 DLL'er og Memory Management

En dll kan bruges fra flere programmer og loades i hver sit program, med hver sin ram.

1.6 Klasser i DLL

Det skal helst undgås, da det KUN er C++, der kan forstå disse klasser

2 Design

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

Source code er det egentlige design

2.1 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

2.2 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.

2.3 SOLID

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

- SRP (Single Responsebility 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

  - Koden kan dog blive for lukket og kompliceret, hvis man gør for meget med det.

- LSP
  - En nedarbede 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  }
```

3 Plugins

3.1 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 overenstemmelse 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 Princples)

  - 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 Abstrction Prinicple)
  - Hvis mange er afhægige, skal de være abstrakte

4 Design

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

Source code er det egentlige design

4.1 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

4.2 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.

4.3 SOLID

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

- SRP (Single Responsebility 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

  - Koden kan dog blive for lukket og kompliceret, hvis man gør for meget med det.

- LSP
  - En nedarbede 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  }
```

5 Plugins

5.1 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 overenstemmelse 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 Princples)

  - 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 Abstrction Prinicple)
  - Hvis mange er afhægige, skal de være abstrakte