Komponentbaseret Program Udvikling

Rasmus Bækgaard, 10893, IKT

0.1 Lektion 1

- WIN RT stammer fra både COM og .NET teknologien

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.

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

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"

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:

DLL'er og Memory Management

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

Klasser i DLL

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

0.2 Design

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

Source code er det egentlige design

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

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.

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  }
```

0.3 Plugins

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

0.4 COM

Distribuering af .dll-filer

- Førhen brugte man .obj-filer. Disse fylder dog meget, og man kan ikke ændre i dem (kun udviklerne kan). Disse kan dog linkes til en .exe-fil.
- I stedet kan man tage en .dll-fil, som kun tager det nødvendige fra en .obj-fil.
- En .dll bliver kun oprettet én gang og alle applikationer der bruger den, bliver opdateret.
- C++ er dog kun standardiseret på Source-niveau og ikke på Binary-niveau - dette kan medføre problemer.
- Laver man en opdatering på en .dll-fil, skal de programmer der bruger dem recompileres

Måden man omgår dette på:

- Man laver en handler, der blot indeholder en pointer til det konkrete objekts funktioner. På den måde udvider .dll-filen sig ikke.
- Man laver en abstrakt base-class (pure virtual). Clienten har kun brug for interfacet. Her skal man dog have et .dll, der har en 'create new concrete object' samt en 'delte concrete object'.

Resource Management

- Hold styr på, hvor mange der bruger et object. Når der ikke er flere, kan det nedlægges.

Goals of COM

- Binary encapsulation: Clienter skal ikke recompilere kode
- Binary compatibility: Filer kan bruges påtvær at platforme

Priciples

- Alt info går igennem et interface
- Ret aldrig i et udgivet interface

Arkitektur

Bruges lokalt, men er så godt som helt dødt på remote.

- Client spørger COM Runtime (CR) efter et objekt.
- CR spørger server efter pointer, der returneres
- Client bruger pointer til server.

COM bruger Registreringsdatabasen. Her ligger alle informationerne om COM-elementerne.

0.5 Memory leak

GC algoritme

- Har et tom blok, hvor en pointer viser, hvor der er plads. Her ligges nye objekter.
- Der kan ikke laves en hurtigere *new* i C#

Finalization

- Det hedder ikke en *destructor* længere
- *Finalize* laver hvad C++'s *destructor* laver - blot kaldt fra GC
- GC holder kun styr på Managed Handles
- Er der noget Unmanaged, skal Finalizes kaldes

```
1 class Foo
2 {
3   ~Foo()
4   {
5     Console.WriteLine("In destructor");
6   }
7 }
```

0.6 MEF - Extentebility Framework

- Ligger i standard frameworket
- Skulle være universelt anvendeligt
- Skal kunne udvide ved blot at uploade ny dll-fil
- *Unity* gør konfigurerbar - ikke udvidelse
- *PRISM* er rettet mod brugergrænseflader
- *System.Addin* vil også lave plugins, men sikkerheden er lidt for god. Det er også meget kompliceret at bruge.
- Kan stille interfaces til rådighed
- Der er ikke en centraliseret konfiguration for MEF
-

En *Hello World*-opbygning:

```
1  using System.ComponentModel.Composition;
2  ...
3  public class Program
4  {
5    public static void Main(string[] args)
6    {
7      Program p = new Program();
8      p.Run();
9    }
10
11   public void Run()
12   {
13     Compose();
14   }
15
16   private void Compose()
17   {
18     var container = new CompositionContainer();
19     container.ComposeParts(this);
20   }
21 }
```

- Contract
  - Normalt et interface (ofte)
  - Normalt også en datatype (men kan være en string)
  - Skriv *[Export]* over klassen der skal eksporteres
  - Skriv *[Export(typeof(...))]* for at specificere en klasse fra et interface
  - Skriv *[Export("funktionsNavn")]* for at specificere en properties
  - Der kan bruges *Shared* og *Non-Shared*
- Import
  - *[Import]* for klasse
  - *[ImportingConstructor]* for en hel klasse

  - Hvis der skal importeres flere af samme type, skriv *[ImportMany(AllowRecomposition=true)]*
  - Kan gøres *Lazy*, hvilket vil sige, at den først initieret, når den skal bruges.

- Catalog
  - Opslag over kendte kontrakter, dog ikke initieret endnu
    * TypeCatalog: Tilføj selv typer
    * AssemblyCatalog: Finder assemblies med bestemet type
    * DirectoryCatalog: Finder et bestemt katalog og dennes indhold
    * AggregatinCatalog: En samling af de tre overstående

- Composition Container
  - Denne laver arbejdet
  - Skal have et catalog og initierer funktioner og klasser.

0.7 PInvoke

- Data er ikke ens. Derfor vil man gerne lave dem om, så de kan bruges sammen. Det kunne være *char** til *string*
- Ejere er ikke de samme. Et et program ejer unmanaged data og GC ejer managed. GC har ikke mulighed for, at fjerne dette

Løsningen:

- PInvode - giver adgang til static entries for unmanaged DLL'er
  - Minder om *Loadlibrary() + GetProcAdress()*

Kapitel 1

DLR - Dynamic Language Runtime

- En motor, hvor man kan bygge dynamiske sprog og statiske sprog
- Disse kører på .NET platformen
-

Terminology:

- Binding - Når man kalder *Foo()* skal compileren vide hvad man kalder
  - Static binding - binder på compile-time
  - Dynamic binding - binder på run-time