Software Design - Obligatorisk Opgave

Om Mediator Pattern

Bjørn Nørgaard Sørensen
Stud.nr: 201370248

Joachim Dam Andersen
Stud.nr: 201370031

Dennis Tychsen
Stud.nr: 201311503

2. December 2015

Indholdsfortegnelse

1 Problem

Når objekters funktionalitet distribueres ud mellem hinanden, vil der opstå høj kobling, og masser af interkonnektivitet. I et mediator pattern oprettes et separat mediator-objekt, som står for at kontrollere objekters interaktioner med hinanden.

2 Løsning

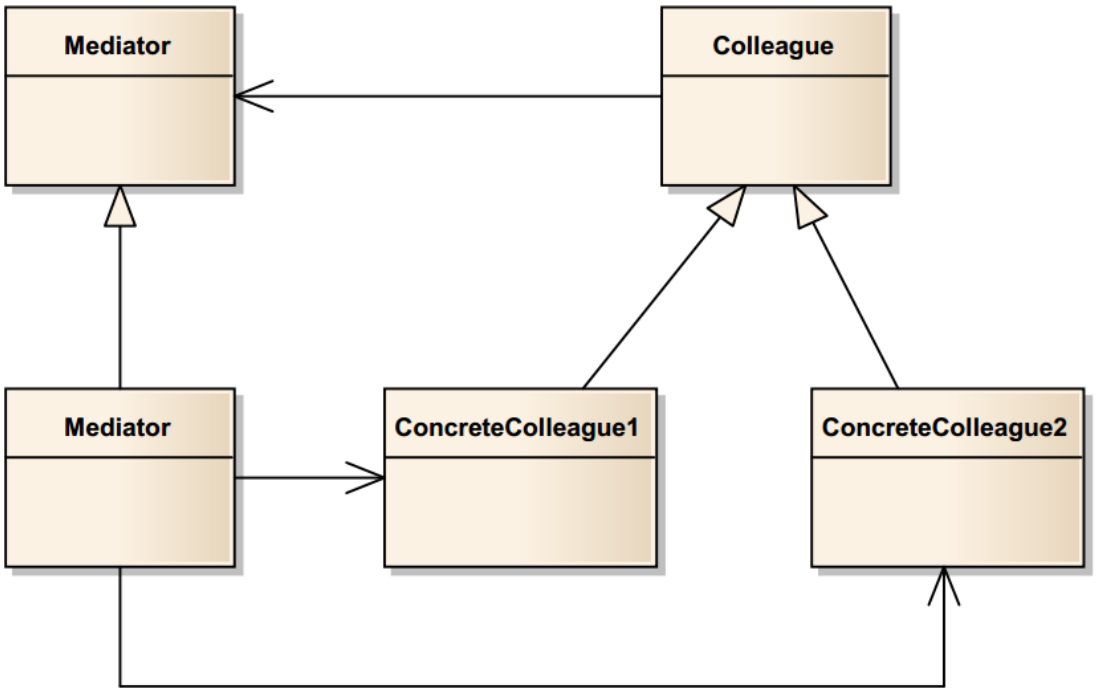

Figure 1: Generelt klassediagram om Mediator pattern.

Definition og identifikation af deltagende klassers type:

- Mediator.
  - Definerer et interface til kommunikation with "Colleague objekter".
- Concrete mediator.
  - Implementerer kommunikationsinterfacet, ved at koordinere Colleague objekter.
- Colleague klasser.
  - Hver colleague klasse kender sit mediator object.
  - Når colleaguen vil snakke med en anden klasse, kommunikeres der udelukkende gennem mediatoren.

Brugen af et mediator pattern begrænser mængden af afledte klasser i et system, i og med mediatoren centraliserer funktionalitet der ellers ville være spredt ud på mange klasser. Ved at pakke objekters interkonnektivitet ind i et mediator pattern, får man samtidig skabt et ekstra abtraktionsniveau der gør funktionalitet mere overskuelig.

3 Eksempel

Her følger et eksempel på brugen af Mediator pattern. Kildekoden kan ses fra github[1] I eksempelet vil klasserne **Kommune** og **Borger** af typen *Participant* kommunikere indbyrdes. Eksempelet er holdt simpelt idet at klasserne via Mediatoren kun kan sende *strings* til hinanden, men selvfølgelig ville dette kunne ændres til hvad end man har brug for.

```
static void Main(string[] args)
{
    Mediator Chatroom = new Mediator();

    Participant Dennis = new Borger("Dennis");
    Participant Joachim = new Borger("Joachim");

    Chatroom.Register(Dennis);
    Chatroom.Register(Joachim);

    Dennis.Send("Joachim", "Herro Jokke!");
    Joachim.Send("Dennis", "Hello Dennis");
}
```

Hele klassediagrammet for programmet kan ses på figur 2 på side 3. En *Participant* har et Mediator medlem og denne bliver sat med *Chatroom.Register("Parcipant");* metoden. Hvorved Mediatoren sætte sig selv som netop denne mediator. Implementering ses her:

```
// you can search for complex class with simple key, i.e. string in this case
protected Dictionary<string, IParticipant> Participants;

public virtual void Register(IParticipant participant)
{
    // checking if participant is already in dictionary
    if (Participants.ContainsValue(participant) == false)
        Participants[participant.Name] = participant;

    // adding this mediator object to participant when registering
    participant.Mediator = this;
}
```

Det er også i Mediatorens *Register* metode at den indeksere det gældende object af typen IParticipant, i dette tilfælde ved *Name* af typen *string*.

Når en Participant vil sende en besked skal den bare kende modtagerens navn og kalde sin mediators *Send* metode, som det kan ses herunder:

```
public virtual void Send(string to, string message)
{
    // Name being the "return-address"/Senders name
    Mediator.Send(Name, to, message);
}
```

Når Mediatoren så skal videreformidle denne besked skal den blot kalde modtagerens *Receive* funktion.

```
public virtual void Send(string from, string to, string message)
{
    // using dict's key as receiver, no need for actual value/object
    IParticipant participant = Participants[to];

    if (participant != null)
        participant.Receive(from, message);
}
```

Herefter kan modtageren så gøre med besked hvad den nu skal. I dette eksempel vil den blot lave en udskrift:

[1] https://github.com/BjornNorgaard/I4SWD/tree/master/MediatorPattern

```
1 public virtual void Receive(string from, string message)
2 {
3       Console.WriteLine(from + " to " + Name + ": " + message);
4 }
```

Igen så er koden at finde på `https://github.com/BjornNorgaard/I4SWD/tree/master/MediatorPattern`. Her følger klassediagrammet for implementering af eksemplet:

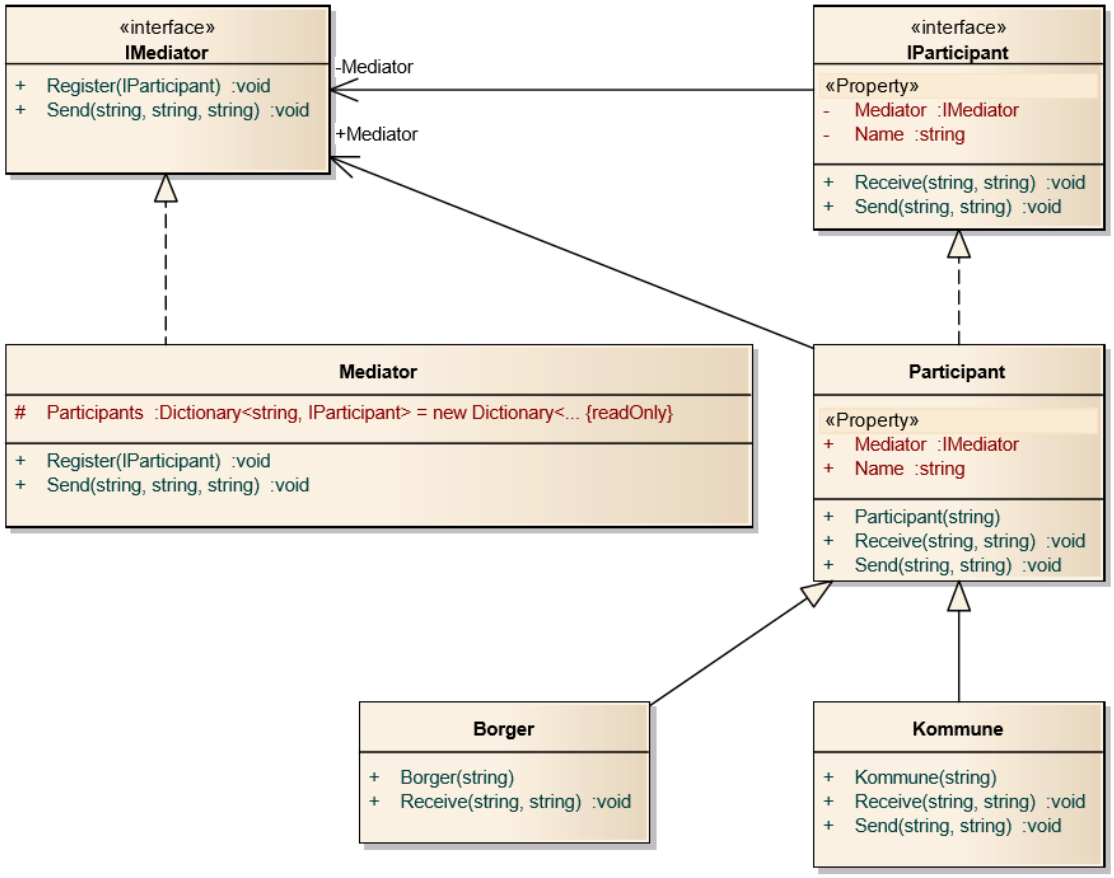

Figure 2: Klassediagram for vores eksempel på Mediator pattern.

4 Sammenligning

4.1 Publisher-subscriber

Mediator patternet kan sammenlignes med publisher-subscriber. Begge står for at håndtere kommunikationen mellem to eller flere klasser. Hvor publisher-subscriber blot broadcaster til alle subscribers, så har mediatoren i højere grad funktionalitet til at finde ud af, hvilke "colleague" eller "subscribers" der skal udføres noget på. Wiki har et glimrende eksempel på siden om Mediator[2], som viser, at alt efter hvilken kommando der kaldes, så udfører mediatoren noget forskelligt, og kalder metoderne hos de registrerede objekter med forskellige parametre.

[2] `https://en.wikipedia.org/wiki/Mediator_pattern#Java` - Mediator eksempel i Java

4.2 Konsekvenser

Mediatoren samler en masse funktionalitet på ét sted, hvilket har den fordel at interaktionen mellem objekter bliver nemmere, men mediatoren får derved større ansvar og bliver mere kompleks. Mediatoren kan derfor hurtigt blive en monolit, som er svær at vedligeholde.

5 Konklusion

Ud fra journalen kan det konkluderes, at Mediator pattern er godt at bruge, i tilfælde, hvor der opstår mange forbindelser mellem mange forskellige klasser. Ved hjælp af mediatoren centraliseres referencerne til de forskellige objekter ét sted, og samler kald til mediatoren, i stedet for enkelte klasser. Dog skal der passes på at mediatoren ikke bliver til en monolit, hvis for meget funktionalitet bliver pakket ind i den.