Designing Classes (Responsibility-driven Design, Cohesion, Coupling and Refactoring Kapitel 3)

Joakim Iversen

January 10, 2025

Disposition:

1. Motivation
2. Copupling
3. Cohesion
4. Responsibility-Driven Design
5. Refactoring

Noter:

KODE EKSEMPEL: WORLD-OF-ZUUL

Main concepts

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

Noter

Motivationen bag at lave god klase design er ikke nødvendigvis at lave et program der virker, da dette godt kan være tilfældet i et dårligt struktureret program. Men derimod at gøre det nemmer at veligeholde koden.

Coupling og Cohesion

Betydning:

Coupling:

- Den interne sammenhængen mellem klasser
- Vi ønske at have lav coupling / loose coupling

Et program med tæt coupling vil være sværer at lave ændringer i da en ændring i en klasse ofte vil kræve at man ændre andre klasser.

Hvis der er loose coupling, vil man derimod godt kunne lave ændringer i en klasse uden at skulle til at ændre de andre klasser. *Vi vil komme ind på nogle eksempler senere*

Cohesion:

- Antallet og forskelligheden af opgaver en enkelt "enhed" i applikation står for

Coheision er relevant at snakke om for enkelte klasser, men også individuelle metoder. Det skal forståes osm at en metode skal være ansvarlig for en cohesive opgav, altså en opgave der også vil kunne ses som logisk enhed.

Tanken bag coheisen er at øge muligheden for at genanvende metoder. Hvis en metode/klasse kun gør en ting - er der større chance for at denne metode/klasse vil kunne bruges igen senere. På den måde slipper vi for at lave for meget kode duplikation.

Kode duplikation

Kode duplikation er produktet af dårlig coheision. Det bunder i, at flere metoder har gentagelser af den samme kode, men med små ændringer der gør, at vi ikke kan andvende dem til at hjælpe hinanden.

I *world-of-zuulu* eksemplet starter vi ud med at have metoderne ***printWelcome*** og ***goRoom*** der begge har kode til at printe det nuværende rum man er i, og de mulige rum man kan gå ind i. Men grundet at hver af de to metoder gør lidt noget forskellige kan vi ikke andvende den ene til at printe mulighederne ved den anden. Dette er dårlig coheision, og det at fikse det er meget simpelt.

Vi vil samle koden der gentager sig i sin egen metode. Vi kan herefter bare kalde denne metode, hver gang vi vil have printet denne information, i stedet for at skrive den samme kode hele tiden.

Responsibility-Driven Design

Resopsibily-Driven Design refferere til tanken om, at en klasse selv skal stå for at behandle sin egen data. Når vi tilføjer en ny funktion til et program, og derved stiller spørgsmålet om hvor denne funktion skal implementeres, er den gode måde at tænke - Den klasse der opbevarer dataen vi skal bruge, skal have metoden til at modificere og manipulere den data.

Dette medfører også, at desto bedre RDD er implementeret, desto lavere vil coupling også være.

Så vi hiver lige et eksempel frem hvor vi nu vil tilføje muligheden for at gå ned imellem kælleren og kontoret. Efter vi har fikset game og room klassen til at håndtere deres egen data, kan vi nemt tilføje denne funktion ved at tilføje følgende linjer til game klassen:

```
private void createRomms()
{
 Room outside, theater, pub, lab, office, cellar;
 ...
 celler = new Room("in the cellar");
 ...
 office.setExit("down", cellar);
 cellar.setExit("up", office);
}
```

På grund af ændringerne i room klassen, vil vi kunne ipmlementere dette uden problemer. Dette bekræftigere at vores applikation design bliver bedre.

Vi kan også forbdre vores getExitString metode:

```
public String getExitString() {
 String returnString = "Exits:";
 Set<String> keys = exits.keySet();
 for(String exit : keys){
  returnString += " " + exit;
 }
 return returnString;
}
```

Her har vi gået fra at tjekke hver mulig ugang, til at udnytte vi har et map med alle udgangene til hvert rum.

Refactoring

Refactoring er når vi har haft et program der er blevet tilføjet flere metoder og linjer af kode til gennem en længere periode. Dette kan medfører at vores program har fået en mere tight cohesion og en dårligere coupling. På dette tidspunkt kan det være nødvendigt at refactor sin application.

Det er når vi gentænker vores applikations design og splitter - eller mindre normalt samler - vores nuværende metoder og klasse for at give en mere løs cohesion og bedre coupling i vores applikation.

Det er vigtigt når man skal refakturere sit program at man ikke tilføjer nye metoder til det, før man har refektureret det og tjekket at det virkede som før. Det kan være fristende at tilføje nye metoder og variabler undervejs, hvis man kommer i tanke om noget nyt, men dette gør ofte hele processen mere uoverskuelig.

Hvis du tilføjere nye features under en refakturering kan det også blive sværer at finde ud af, hvor og hvad der er meført denne fejl i programmet.