**Talepapir Producer-consumer + proxy**

Producer-consumer problemet opstår, når mange tråde gerne vil arbejde med det samme på en specifik måde.

I vores producer-consumer setup, har vi en række producer tråde, som eksepelvis producerer opgaver, som consumer tråde således kan tag, og så udføre dem. Her vil der således opstå en kø, hvor producerne sætter deres opgaver ind i køen, og consumer vil tage dem fra køen.

Problemet opstår så, hvis alle trådene arbejder parallelt med hinanden. Hvis producerne producerer opgaverne parallelt, så risikerer man, at de overskrider en grænse for antallet at opgaver, som maksimalt må produceres. Omvendt kan consumer trådene komme til at arbejde med flere opgaver, end der egentlig overhovedet er produceret, eller de kan komme til at arbejde med samme opgave.

For at undgå dette, kan man lave en monitor klasse og en blocking queue.

Blocking queue fungerer som den kø, som jeg snakkede om før, som de forskellige parter arbejder med. Den opstår når der bliver smidt opgaver nid, men så tilføjer vi blocking, således at trådene ikke kan smide opgaver ind, så længe den er fyldt. Og omvendt, så kan trådene ikke tage opgaver, hvis køen er tom. De bliver på den måde ”blokeret”, indtil de må arbejde igen.

Monitor klassen er den fælles ressource, som producer og consumer arbejder med. Den ”overvåger” adgangen mellem de to klasser, og sørger for synkronisering af metoder, så flere tråde kan arbejde heri på en sikker måde. Trådene får en instans af monitor klassen, så de kan bruge dens metoder.

Det er altså den klasse der skal fortælle, hvornår køen er tom/fuld, og den skal notificere de andre klasser, hvornår de må begynde at arbejde igen.

Her bruger man wait/notify.

Når en tråd træder en i en synkroniseret metode, så får den en lås, som gør at andre ikke kan komme ind og arbejde, før den er færdig. Hvis det nu er en consumer tråd, og køen den er tom, så skal man kalde wait() metoden, så tråden afgiver sin lås, men i stedet kommer i en ventetilstand. På den måde kan andre tråde komme ind og arbejde med metoden.

Når en producertråd så fylder i køen igen, så skal den notificere alle ventende consumers, så de kan genoptage en lås, og fortsætte deres arbejde.

**Proxy**

Proxy har mange formål, da den kan indtræde i forskellige situationer. Et af formålene er at give en stedfortræder til et andet objekt, så man kan kontrollere adgangen til den. Det kan man kalde for en slags ”wrapper”, som pakker realSubjekt ind i en anden klasse, som får lov til at deligere arbejdet videre til realSubjekt. F.eks. hvis en klasse ikke er thread-safe, så kan vi pakke den ind i en anden klasse, som er.

Pointen med procy er hovedsageligt, at man kan have objekter, som er meget tunge at oprette, og så kan vi have en proxy, som så styrer, hvornår vi rent faktisk opretter objektet. Der er ingen grund til at oprette et objekt, før det bliver efterspurgt af klienten.

De forskellige typer proxy vi kan støde på, er

* Virtual Proxy: Sørger for først at oprette objektet, når klienten kalder på det
* Remote Proxy: Laver en lokal repræsentation af et objekt, som ellers har lokaseret på en anden adresse. Det er det vi typisk bruger i client/server RMI, det er det ”stub” koden sørger for.
* Protection Proxy: validering – den tjekker om client kan få adgang til RealSubject – her kan man indsætte adgangskrav (DEN JEG HAR I AFLEVERING)
* Cache Proxy: Hvis man har en klasse, der har data på tabel, som sjældent ændrer sig, så kan man lave en proxy, som tjekker om data allerede er hentet, og så gemme det i proxyen, og hvis ikke det allerede er søgt på før, og data derfor ikke er gemt i proxy, så kan man give videre adgang til RealSubject – DET GÅR MEGET HURTIGERE!
* Smart Proxy: Man tilføjer ekstra funktionalitet, f.eks. hvis man har et log interface, som har to klasser som implementerer den – den ene logger til consollen, den anden logger til en fil – her kan man smide proxy foran, som har en metode, der vil have et argument for, om den skal tilføjes det ene eller andet sted – f.eks LogProxy(toFile:boolean), altså bestemmes det i constructoren

Det er altså alle sammen en form for ”wrapper” klasser, som indpakker realSubekt.

De forskellige parter involveret er vores subject interface, som er det klienten arbejder med. Så har vi realSubject, som nedarver fra interfaces. Hvis vi vil tilføje ekstra funktionalitet til realSubject, og så har vi en proxy, som også nedarver fra interfacet, og fungerer som stedfortræder.

De skal altså begge nedarve fra samme interface, for at proxy’en kan fungere som stedfortræder.

**Producer consumer**

* Producer/consumer opstår = mange tråde, arbejde på det samme, specifik måde
* Producer-consumer setup = Række producer tråde = producerer opgaver
* Consumer tråde tager opgaverne, og udfører dem
* KØ, hvor producer indsætter, consumer tager
* PROBLEM: hvis alle tråde arbejder parallelt
* Producer parallelt = overskrider grænse
* Consumer parallelt = tager mere end de må ELLER arbejder på samme opgave
* LØSNING: Blocking queue og monitor klasse
* Blocking queue: kø fra før
  + Opstår, når opgaver bliver smidt ind
  + Man tilføjer blocking = grænser
    - Kan ikke fylde på når fyldt
    - Kan ikke tømmes når tom
  + DE BLIVER BLOKERET, indtil de må arbejde igen
* Monitor klasse: den fælles ressource, begge parter arbejder med
* Monitor klassen ”overvåger” adgangen mellem de trådene
  + GENERELT KLASSE MED FLERE TRÅDE
* Sørger for synkronisering af metoder
  + = flere tråde kan arbejde sikkert
* Trådene får en instans af monitor klassen = brug dens ”sikrede” metoder
* Monitor klassen skal fortælle, hvor køen er tom/fuld
* Notificer andre, hvornår de må arbejde igen
* WAIT/NOTIFY
* Tråd træder ind i sykroniseret metode = lås
* Ingen adgang for andre
* ARBEJER IKKE PÅ DET SAMME
* Wait
* Consumer tråd og tom kø
* Wait()
* Afgiver sin lås, og kommer i ventetilstand
  + Så kan andre tråde stadig komme ind i metoden
* Når producer indsætter = notificer consumer, så de kan genoptage lås, og fortsætte arbejde

**Proxy**

* Kan have mange formål, alt efter situation
* Hovedformål: lave en stedfortræder af et objekt
* F.eks. hvis man via stedfortræderen kontrollerer adgang til oprindelige objekt
* Slags ”wrapper”, som indpakker realSubjekt i anden klasse
* Deligerer arbejdet videre til realSubjekt, vha. INSTANS
* F.eks. ikke thread proof klasse -> thread-proof
* POINTE: tunge objekter at oprette
* Proxy styrer hvornår vi opretter
* Forskellige proxyer:
  + Virtuel: først opret objektet, når klient skal bruge
  + Remote: lokal repræsentation af objekt- anden adresse – client/server det STUB koden gør
  + Protection: validering = tjekker om klient må få adgang til realSubjekt = ADGANGSKRAV
  + Cache: datatabel sjældent ændrer sig – gem allerede hentet data i proxy
  + Smart: ekstra funktionalitet = logger til konsol og txt-fil, proxy tager argument for hvilken en af dem
* ”Wrapper” klasser = indpakker realSubject
* Forskellige parter:
  + Subject interface: det klienten arbejder med
  + RealSubject = nedarver subject
  + Hvis vi vil tilføje ekstra funktionalitet =
    - Proxy = nedarver subject = instans af realSubject = stedfortræder
  + DE SKAL NEDARVE SAMME INTERFACE