**Question 6A:** Producer-Consumer problem

Explain the Producer-Consumer problem, including the blocking queue (Touch upon monitors, wait/notify, critical sections as needed).

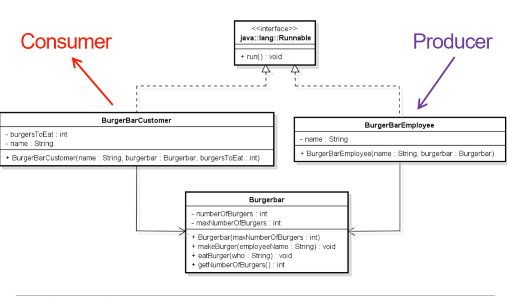
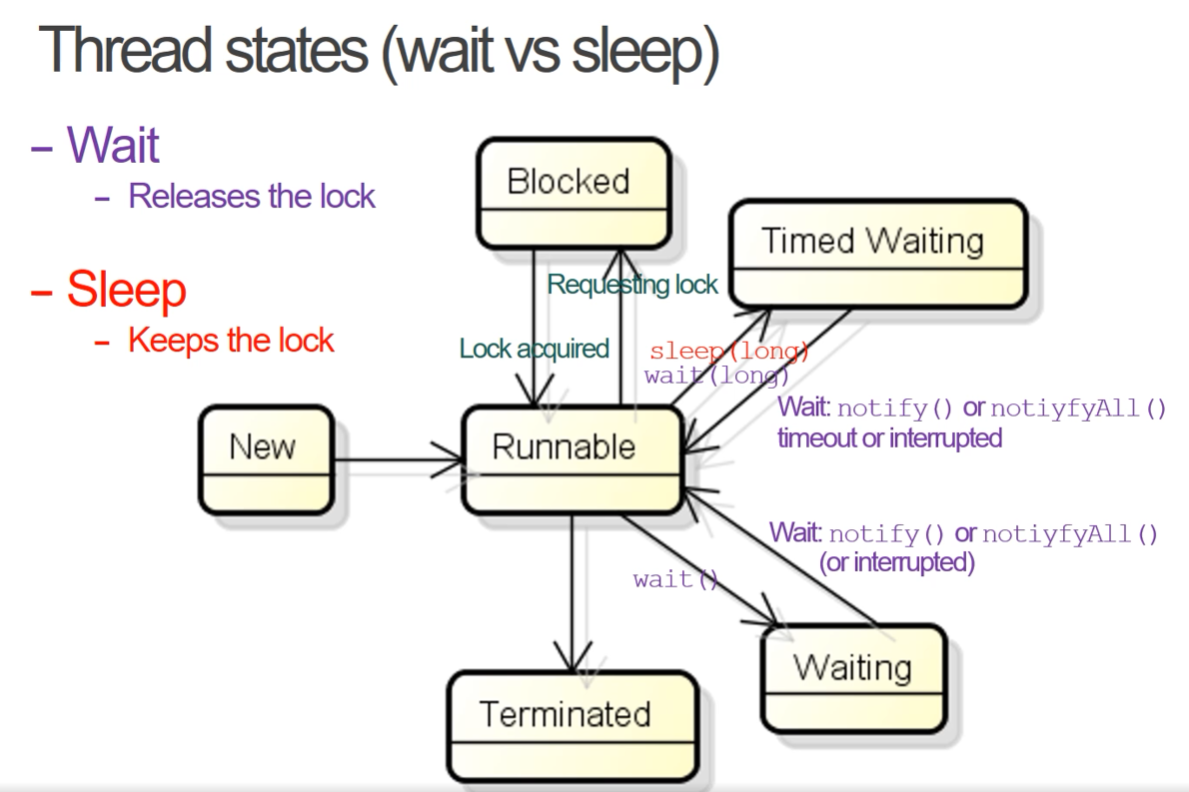
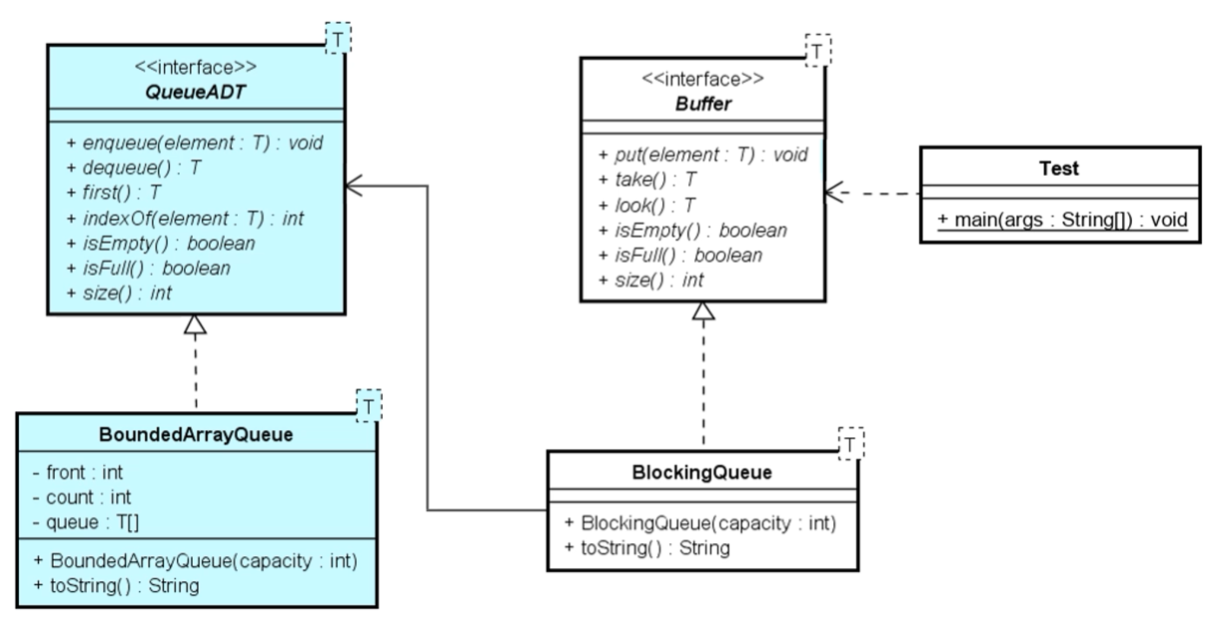
Present your own example from an assignment, SEP2, or exercise

* Pointe med producer-consumer: Man bruger producer/consumer løsningen her, når mange tråde vil arbejde på det samme på en specifik måde.

Der er producer tråde, som eksempelvis producerer opgaver, og consumer tråde, som tager opgaverne, og udfører dem = her opstår en kø, hvor producerne indsætter deres opgaver i køen, og consumer vil tage opgaverne fra køen

Problem: producer tråede producere opgaverne parallelt, og det samme med consumer, som tager opgaverne parallelt – men hvad så, når der bliver produceret flere opgaver, end der må være? Eller hvad hvis consumer trådene tager flere opgaver end der er? Eller kommer til at tage den samme opgave?

Løsning: DE ER NØDT TIL AT VENTE! – MONITOR KLASSE!

* Blocking queue: Det er køen der opstår, når der bliver smidt opgaver ind – men blocking sker, da trådene kun må smide opgaver ind, så længe den ikke er fyldt/trådene kan tage opgaver, så længe der er opgaver, og køen ikke står tom
* Monitor klassen:
  + Den fælles ressource (f.eks er vi en ”optæller” klasse, og en ”nedtæller” klassse, og en ”talt” klasse – ”talt” klassen er den fælles ressource, som de to andre klasse påvirker, og denne er derfor monitor klassen)
  + Privat instans variabler
  + Synkroniserede metoder, da der er tråde som går ind og bruger dens metoder
  + Wait/notify – Det er denne klasse der fortæller producer/consumer klasserne, op der er noget som er tomt/fyldt, og den notifier de andre klasser, hvornår de må begynde på deres jobs igen
* Trådene får altså en instans af monitor klassen, så de kan bruge dens metoder – ingen wait/notify her!
* Forklaring af wait:
*  = wait gør at andre tråde kan komme ind i den synkroniserede metode, da den afgiver sin lås – hvis tråden bare skulle sleep i stedet, så kunne de andre tråde ikke komme ind, før den sovende tråd blev færdig med metoden
* Stadig smart med syncronize, idet at tråden ikke er i vente tilstand, så SKAL de andre vente på den er færdig, så de ikke komme til at arbejde på samme værdi
* Implementering af monitor klassen: BlockingQueue = monitor klassen her, den har synkroniserede metoder, og wait/notify. Den tager en instans af QueueADT som er en eksisterende queue, og så gør den det til en blockingQueue, ved at synkronisere og bruge wait/notify
* I forhold til adapter pattern, så er BoundedArrayQueue og QueueADT Adaptee, BlockingQueue er adapter, Buffer er target, og Test er client   
  

Samlet set: Monitor ER queue, og queue er monitor!

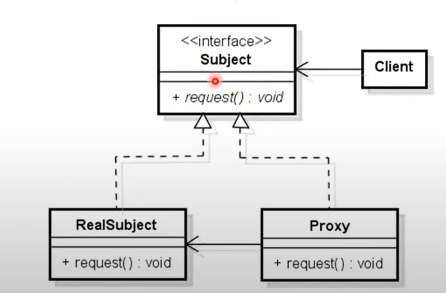
Man kan bruge Javas standard BlockingQueue klasse, det er nemt – men man kan ikke kontrollere ”limits”, regler for hvornår man skal vente

* Use UML to present an overview for your solution to a Producer-Consumer problem
* Show the related code parts for your presented solution.

**Question 6B:** Proxy design pattern

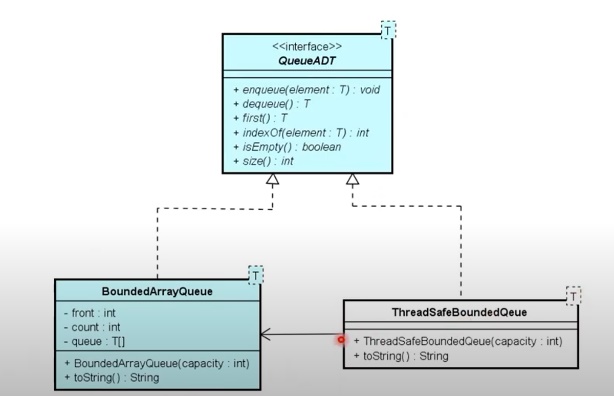
Describe the pattern using UML diagrams.

* What is the purpose?
  + Formål: Kan have mange formål – et af dem er: At give en stedfortræder til et andet objekt, så man kan kontrollere adgang til den
    - En ”wrapper” til at deligere arbejde til realSubject
    - F.eks. hvis vi har en klasse som ikke er thread safe, kan vi pakke den ind i en anden klasse, som er thread safe
  + Problem: Man kan have objekter som er tunge at oprette, og så kan vi lave en proxy, som så styrer, hvornår vi rent faktisk opretter objektet – ingen grund til at oprette det, før det rent faktisk bliver efterspurgt af client



* What are the different parts involved?
  + Vi har et Subject interface
  + Vi har et realSubject, som implementerer Subject
  + Vi vil gerne tilføje funktionalitet, og det gør vi ved at sætte en proxy foran
  + Proxy = stedfortræder
  + Subject tror den har fat i det rigtige object, men det er det ikke helt, Proxy har bare en instans af RealSubject, så vi kan give mere funktionalitet i mellem dem

Proxy: A thread safe class:



Vi har en BoundedArrayQueue, som implementerer QueueADT – så vi har altså en Queue

MEN INGEN AF METODERNE ER SYNCRONIZED!  
Vi tilføjer derfor en proxy, altså ThreadSafeBoundedQueue, som implementerer samme interface, tager en instans af BoundedArrayQueue, men tilføjer den ekstra funktionalitet, at metoderne skal være Thread Proof!

Forskellige typer proxy:

* Virtual Proxy:
* Remote Proxy: Når ClientModelManager gerne vil have data fra ServerModelManager
  + ClientModelManager er proxy, den har en instans af client, som kommunikerer til ClientHandler, som har en instans af ServerModelManager, som er RealSubject
    - ClientModelManager og ServerModelManager implementerer begge Model interfacet (remote exception ting ting)
* 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
* Wrapper Proxy:

Present your own example from an assignment, SEP2, or exercise

* Use UML and describe how your solution follows the pattern
* Show Java code – how the pattern is implemented.
* Show Java code for how the pattern can be used.

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.