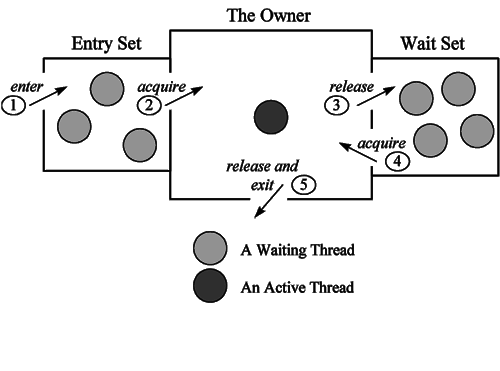
**Question 5A:** Readers-Writers problem

Explain the Readers-Writers problem (Touch upon monitors, wait/notify, critical sections, guarded blocks and the relation to threads as needed)

* Vi kan have en eller writers, som gerne vil modificere et fælles obejekt, og vi flere readers, som gerne vil læse objektet uden at modificere.
* Writers skal helst kun have tilladelse til at modificere på objektet én ad gangen, så de ikke sidder og laver ændringer i samme objekt. Derfor er man nødt til at bruge Synchronized.
* Tilgengæld kan der sagtens være mange readers på ad gangen, og de behøver således ikke synchronization
* Readers kan dog ikke læse, når en writer er i gang med at lave ændringer.
* For at opnå dette, kan man bruge
  + readers preference, hvor Readers begynder at læse, når der ikke er aktive writers – kan resultere i at writers aldrig får adgang, når Readers bliver ved med at komme til
  + ved writers preference, skal readers OGSÅ vente, hvis der er andre writers, som venter på at komme ind – en ventende writer kommer foran i køen end en ventende reader
  + Hvis det skal være helt fair, kan readers og writers få adgang i den rækkefølge, som de først kom ind – her smides de i en kø
* For at implementere dette, skal readers og writers implementere Runnable, således de bliver til tråde, som går ind og læser/skriver data.
* Løsningen er så, at vi skal kontrollere adgangen til den fælles ressource.
* Denne løsning er en monitor klasse, hvor ALLE metoder er synkroniseret, og den kontrollerer hvornår hver part må få adgang, og hvornår nogen må vente. Når reader/writer er færdig, skal den notify andre, så dem som venter ved, hvornår de kan få adgang.
* VIS WAIT/NOTIFY figur til højre: Figuren viser, at der er nogen som gerne vil have adgang til en metode, ved at have en guarded block, som angiver hvornår den må komme ind, så kan den risikere, at den må vente. Den afgiver sin lås i vente positionen, således andre også kan komme ind og få adgang til metoden imens de venter.
* PRÆSENTER UML
* VIS KODE

Talepapir:

* Vi kan have en eller mange writers, som gerne vil modificere et fælles obejekt, hvor flere readers, gerne vil læse objektet uden at modificere.
* Det kan give nogle vanskeligheder, da objektet helst ikke skal modificeres, imens readers er i gang med at læse. Der kan tilgengæld være mange readers i gang på samme tid uden problemer.
* Her kan man bruge lidt forskellige approaches til at løse problemet, alt efter hvad der giver bedst mening for ens program.
  + readers preference, hvor Readers begynder at læse, når der ikke er aktive writers – det kan dog resultere i at writers aldrig får adgang, når Readers bliver ved med at komme til
  + ved writers preference, skal readers OGSÅ vente, hvis der er andre writers, som venter på at komme ind – en ventende writer kommer foran i køen end en ventende reader
  + Hvis det skal være helt fair, kan readers og writers få adgang i den rækkefølge, som de først kom ind – her smides de i en queue, og den forreste er altså den der har ret til adgang som den næste.
* For at implementere dette, skal readers og writers implementere Runnable, således de bliver til tråde, som går ind og læser/skriver data.
* Løsningen er så, at vi skal kontrollere adgangen til den fælles ressource, og dette gøres ved hjælp af en monitor klasse, hvor ALLE metoder er synkroniseret, og den kontrollerer hvornår hver part må få adgang, og hvornår nogen må vente. Når reader/writer er færdig, skal den notify andre, så dem som venter ved, hvornår de kan få adgang.
* FIGUR WAIT: Figuren viser, at der er nogen som gerne vil have adgang til en metode, og ved at have en guarded block, som angiver hvornår den må komme ind, så kan den risikere, at den må vente. Den afgiver sin lås i ventepositionen, således andre også kan komme ind og få adgang til metoden imens de venter.

Når så den selv bliver notified, kan den få adgang.

* UML: her ses en accountant, som gerne vil tælle op i treasureRoom, og en King som gerne vil bruge pengene herfra. Altså er accountant reader, og king er writer.

For at få adgang til treasureroom, skal de i gennem TreasureRoomGuardian, som er monitorklassen, der bestemmer, hvornår de hver især må få adgang.

Det er en writers approach, så ventende readers kommer altså bag en ventende writer.

* KODE: Begge klasser implementerer Runnable, samt har en instans af TreasureRoomDoor.
* Accountant vil gerne have readAccess, og så ind og se alle gems i TreasureRoom, og herefter releaseReasAccess igen.
* King vil gerne ind og have writeAccess. Så længe at han ikke har råd til at holde en fest for sine penge, henter han gems fra TreasureRoom. Så længe han KAN hente gems fra TreasureRoom, tilføjes disse til hans samling (ArrayList). Hvis IKKE han kan hente nok fra TreasureRoom, ligger han dem alle tilbage hertil igen. HVIS han har nok, kan han holde en fest, og alle gems han havde gemt, vil blive nulstillet.  
  Han releaser herefter sin writerAccess
* GuardsMan bestemmer hvornår de hver især kan få ReaderAccess og WriterAccess, og dette er bestemt med guarded blocks. Her defineres det hvornår man kommer i vente position (FORKLAR UDFRA KODEN)
* Nede i releasemetoderne bliver det bestemt, hvem der bliver notified.

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

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

**Question 5B:** Singleton

Describe the pattern using UML diagrams.

* What is the purpose?
  + Hvis et objekt er meget tungt eller lignende, og det er ressourcekrævende at oprette, så kan det være en fordel først at instantiere det, første gang man bruger det – altså lazy instantiation. Man tjekker således først om objektet er null, og hvis det er, så bliver den oprettet. Det er i stedet for at oprette det i constructoren
  + Singleton bruger lazy instantiation, som således løser, at når man gerne vil have én instans af en klasse, som man kan få global adgang til.
  + Dette kan eksempelvis ske ved logging, hvor man flere forskellige klasser gerne vil logge noget til en fil. Ved at bruge en singleton logger klasse, så sørger man for at der kun bliver oprettet én fil, hvor de alle kan skrive i, ved hjælp af en global metode i Logger klassen.
* What are the different parts involved?
  + IMPLEMENTATIONEN: Den skal have en privat constructor, så udefra kan lave en ny instans af den klasse.
  + Den skal have en privat statisk instans af sig selv, som så bliver den globale instans andre kan tilgå
  + Den skal have en statisk metode, til at returnere denne instans. Metoden er nødt til at være statisk, således man ikke skal bruge et objekt for at kalde den.
  + Et problem som så kan opstå, når man har denne metode, og der muligvis er flere tråde, som gerne vil ind og oprette instansen, så KAN det ske, at 2 eller flere tråde opretter instansen på præcis samme tid, og derved opretter 2 forskellige instanser.
  + Dette kan løses med at gøre metoden Thread safe
  + Det vil sige, at man er nødt til at bruge synchronization. Enten kan man bruge det på hele getInstance metoden, men så blokere man også for tråde, når der er en inde i metoden. Det kan være træls, når instansen er oprettet, og alle nemt burde få tilgang til den
  + Anden løsning er, at der kun bliver låst, aller første gang den bliver kaldt. Her bruger man double checked locking
    - Vi syncronizer efter vi tjekker, om instansen er null. Dog kan flere tråde stadig nå ind i if-statementet, og tro at den er null. Derfor skal vi have endnu en if-statement der tjekker om den er null, efter vi har synkroniseret. På den måde vil de andre tråde tjekke igen, og denne gang se, at den ikke er null
  + KODE:

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.

**Readers Writers Problem**

Vi kan have en eller mange writers, som gerne vil modificere et fælles obejekt, hvor flere readers, gerne vil læse objektet uden at modificere.

Det kan give nogle vanskeligheder, da objektet helst ikke skal modificeres, imens readers er i gang med at læse. Der kan tilgengæld være mange readers i gang på samme tid uden problemer.

PROBLEM: de må ikke alle få adgang til objektet samtidig.

Her kan man bruge lidt forskellige approaches til at løse problemet, alt efter hvad der giver bedst mening for ens program.

* readers preference, hvor Readers begynder at læse, når der ikke er aktive writers – det kan dog resultere i at writers aldrig får adgang, når Readers bliver ved med at komme til
* ved writers preference, skal readers OGSÅ vente, hvis der er andre writers, som venter på at komme ind – en ventende writer kommer foran i køen end en ventende reader
* Hvis det skal være helt fair, kan readers og writers få adgang i den rækkefølge, som de først kom ind – her smides de i en queue, og den forreste er altså den der har ret til adgang som den næste.

For at implementere dette, skal readers og writers implementere Runnable, således de bliver til tråde, som går ind og læser/skriver data.

Løsningen er så, at vi skal kontrollere adgangen til den fælles ressource, og dette gøres ved hjælp af en monitor klasse, hvor ALLE metoder er synkroniseret, og den kontrollerer hvornår hver part må få adgang, og hvornår nogen må vente. Når reader/writer er færdig, skal den notify andre, så dem som venter ved, hvornår de kan få adgang.

FIGUR WAIT: Figuren viser, at der er nogen som gerne vil have adgang til en metode, og ved at have en guarded block, som angiver hvornår den må komme ind, så kan den risikere, at den må vente.

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 writer tråd som gerne vil have adgang til en metode, og der er en aktiv reader, 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 der så ikke er flere aktive readertråde, alt efter hvilken approach man har selvfølgelig, så skal readers notificere alle ventende writers, så de kan genoptage en lås, og fortsætte deres arbejde.

**Singleton**

Hvis et objekt er meget tungt eller lignende, og det er ressourcekrævende at oprette, så kan det være en fordel først at instantiere det, første gang man bruger det – altså lazy instantiation. Man tjekker således først om objektet er null, og hvis det er, så bliver den oprettet. Det er i stedet for at oprette det i constructoren

Singleton bruger lazy instantiation, som således løser, at når man gerne vil have én instans af en klasse, som man kan få global adgang til.

Dette kan eksempelvis ske ved logging, hvor man flere forskellige klasser gerne vil logge noget til en fil. Ved at bruge en singleton logger klasse, så sørger man for at der kun bliver oprettet én fil, hvor de alle kan skrive i, ved hjælp af en global metode i Logger klassen.

Den skal have en privat constructor, så udefra kommende ikke kan lave en ny instans af den klasse.

Den skal have en privat statisk instans af sig selv, som så bliver den globale instans andre kan tilgå

Den skal have en statisk metode, til at returnere denne instans. Metoden er nødt til at være statisk, således man ikke skal bruge et objekt for at kalde den.

Et problem som så kan opstå, når man har denne metode, og der muligvis er flere tråde, som gerne vil ind og oprette instansen, så KAN det ske, at 2 eller flere tråde opretter instansen på præcis samme tid, og derved opretter 2 forskellige instanser.

Dette kan løses med at gøre metoden Thread safe

Det vil sige, at man er nødt til at bruge synchronization. Enten kan man bruge det på hele getInstance metoden, men så blokere man også for tråde, når der er en inde i metoden. Det kan være træls, når instansen er oprettet, og alle nemt burde få tilgang til den

Anden løsning er, at der kun bliver låst, aller første gang den bliver kaldt. Her bruger man double checked locking

* Vi syncronizer efter vi tjekker, om instansen er null. Dog kan flere tråde stadig nå ind i if-statementet, og tro at den er null. Derfor skal vi have endnu en if-statement der tjekker om den er null, efter vi har synkroniseret. På den måde vil de andre tråde tjekke igen, og denne gang se, at den ikke er null

**Readers writers**

UML: her ses en accountant, som gerne vil tælle op i treasureRoom, og en King som gerne vil bruge pengene herfra. Altså er accountant reader, og king er writer.

* + - For at få adgang til treasureroom, skal de i gennem TreasureRoomGuardian, som er monitorklassen, der bestemmer, hvornår de hver især må få adgang.
    - Det er en writers approach, så ventende readers kommer altså bag en ventende writer.

KODE: Begge klasser implementerer Runnable, samt har en instans af TreasureRoomDoor.

Accountant vil gerne have readAccess, og så ind og se alle gems i TreasureRoom, og herefter releaseReasAccess igen.

King vil gerne ind og have writeAccess. Så længe at han ikke har råd til at holde en fest for sine penge, henter han gems fra TreasureRoom. Så længe han KAN hente gems fra TreasureRoom, tilføjes disse til hans samling (ArrayList). Hvis IKKE han kan hente nok fra TreasureRoom, ligger han dem alle tilbage hertil igen. HVIS han har nok, kan han holde en fest, og alle gems han havde gemt, vil blive nulstillet.  
Han releaser herefter sin writerAccess

GuardsMan bestemmer hvornår de hver især kan få ReaderAccess og WriterAccess, og dette er bestemt med guarded blocks. Her defineres det hvornår man kommer i vente position (FORKLAR UDFRA KODEN)

Nede i releasemetoderne bliver det bestemt, hvem der bliver notified.

**Singleton**

KODE: Som man kunne se i klasserne i Kingdom mappen, så bliver en klasse ved navn Catalogue brugt statisk flere gange, for at logge en String til en tekstfil.

* Denne klasse er en singleton. Den opfylder alle kravene (kig på koden og forklar). Hertil er der en log metode, som kun kan tilgås via den statiske getInstance metode, hvorved man er sikret, at der arbejdes med den samme instans. ’