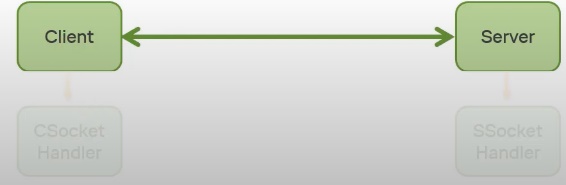
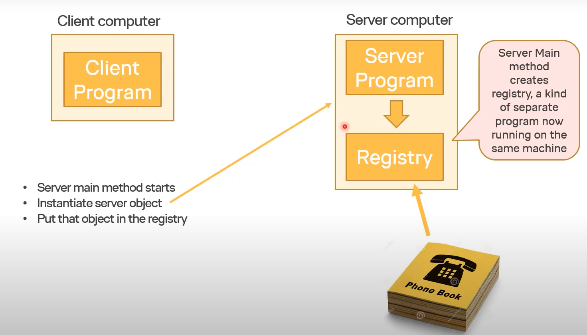
**Question 3A: RMI**

Explain the main parts of a communication between computers using RMI.

* RMI = Remote Method Invocation
  + Bruges til at få 2 programmer til at tale med hinanden
* Purpose: Man vil gerne instantiere objekter på én computer (Server), og så skal man kunne lave metode kald direkte på dette objekt, fra klient siden
* Med RMI kan man – modsat sockets, hvor man skal lave en request – kalder metoderne direkte fra Server objektet, og herved kan vi slippe for ”if-else” statements, for at finde ud af hvilke metoder der skal bruges
* Måden RMI kommunikerer, kan ikke ses fra brugerens side – modsat Sockets, hvor vi bruge input/outputstreams, som var dem der kommunikerede mellem client og server – Det foregår ”behind the scenes”
* RMI er en Java framework, og både server og client skal være skrevet i java, før det kan bruges
* Fordele: Vi behøver ikke at tænke på multi-threading, ligesom ved sockets, hvor hver client skulle have sin egen tråd i sockethandleren– det sørger RMI helt selv for ”behind the scenes”
* Vi skal stadig have synchronization på de metoder, som klienterne ikke må tilgå samtidig
* Figur til højre: sådan ser det ud for os, men i virkeligheden bruger de altså en sockethandler – men det kan vi ikke se, det ordner java for os

Include among other things in your description: Remote interface, Registry and Stub.

* Registry: lidt ligesom en telefonbog
  + Server main method starter, og instantierer Server objektet og opretter registry samtidig, og putter dette objekt i registry
  + Registry er på en måde et separat program, som nu kører på sammme maskine
  + Nu har vi på en måde lavet vores ”server socket”, som venter på en connection fra client
  + Client main starter
    - Den kontakter registry, og herfra vil den ”look-up” det den mangler, som er vores server-program
    - Man kan have mange objekter i registry
* Stub: interface som repræsenterer server objektet, og server objektet skal derfor implementerer dette interface
  + Når client vil ”look-up” vores server objekt i registry, får den en instans af stub
  + Stub bliver til en lokal repræsentation af vores server, og når det er denne der bruges til at lave metode kald fra serveren, da vores stub indeholder metoderne
  + Klienten kan nu kalder metoder på server-objektet (remotely)
* Remote interface:
  + Skal implementers hvis man vil have en server objekt til at være remotely available
  + Alle definerede metoder i dette interface throws RemoteException
  + Serveren skal have en constructor, hvor den eksporterer UnicastRemoteObject (laver en kald, så den ekporterer sig selv til at være tilgængelig for Remote interfacet)

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

* Use UML to present an overview for your solution
* Show the related code parts for RMI communication for your presented solution.
* Describe e.g. by an example, how to handle a server call-back to a client.
  + I stedet for at bruge en return method – f.eks. når vi vil have en toUpperCase, så returner den en String, og når metoden bliver brugt på client siden, hvor der står server.toUpperCase(), så får vi returneret resultatet med det samme, altså synkront, da vi får en retur værdi
  + Hvis man ikke bruger return method, men har en void f.eks., kan man lave call-back til klienten.
  + Det vil sige: Når et metodekald bliver sendt ned til serveren, kan det være den skal lave en meget tung beregning, eller hente noget fra en database – noget tidskrævende
    - når den er klar med resultatet, vil den kalde en metode på klienten, og sige ”hey, her er dit resultat”
    - Det er altså ikke en retur-værdi, men et kald tilbage til klienten – derfor call-back
* Describe e.g. by an example, how to handle a server broadcast to all clients.
  + Når man gerne vil sende et resultat ud til ALLE lyttende klienter
  + Lyttende klienter skal kunne registrere sig selv, når de gerne vil være i stand til at få updates
  + Serveren får herved en liste af clients, så den kan broadcaste ud til dem alle på én gang

Talepapir RMI:

RMI står for Remote Method Invocation, og det bruges til at få 2 programmer til at tale med hinanden.

Det der så skal ske er, at man opretter ét eller flere Server objekter, som man skal kunne lave metodekald på fra klientsiden.

Vi kan som programmører ikke se hvordan kommunikationen sker i RMI, modsat sockets, hvor vi bruger input/outputstreams. I RMI foregår det hele ”behind the scenes” (vis billede), hvor det for os ligner, at client og server kommunikerer direkte med hinanden, men der ligger altså også en form for sockethandler i baggrunden, som styrer det hele.

Vi behøver derfor ikke at tænke på multithreading, det sørger vores RMI framework for os.

Vi skal dog stadig have synchronization på de metoder, som ikke må bruges af klienterne samtidig.

For at implementerer RMI korrekt, er der tre begreber jeg vil uddybe:

Registry: man kan sammenligne det lidt med en telefonbog. Når det er, at vores server main starter, så opretter den både vores server objekt, og samtidig opretter den også denne her registry, hvori vores objekt bliver puttet i. Når dette er sket, har vi på en måde oprettet vores ServerSocket, som nu venter på klienter skal connecte.

Når client main starter, vil den ”slå op” i vores registry, og lede efter vores server-objekt. Det er derfor registry minder om et opslagsværk.

Når klienten finder server-objektet i registry, får den en instans af server-objektets stub. Stub fungerer som klientens lokale repræsenterede server objekt, og metodekald bliver lavet på stub-objektet, som er ansvarlig for at bringe metodekaldet videre hen på server objektet.

For at alt dette skal kunne lade sig gøre, skal vores server objekt implementere Remote interfacet. En stub for vores server obejekt, implementerer herved også det samme sæt af remote interfaces, som vores server objekt.

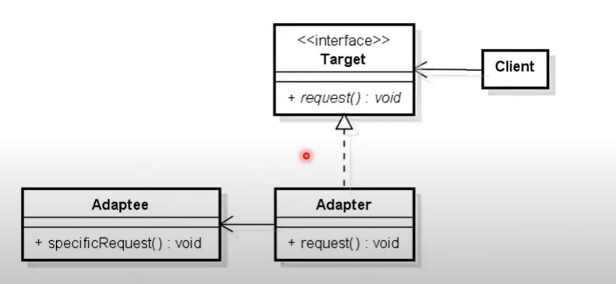
Serveren skal også have en constructor, hvor den eksporterer UnicastRemoteObject – altså skal den lave en kald, så den kan eksportere sig selv til at være tilgængelig for Remote interfacet.

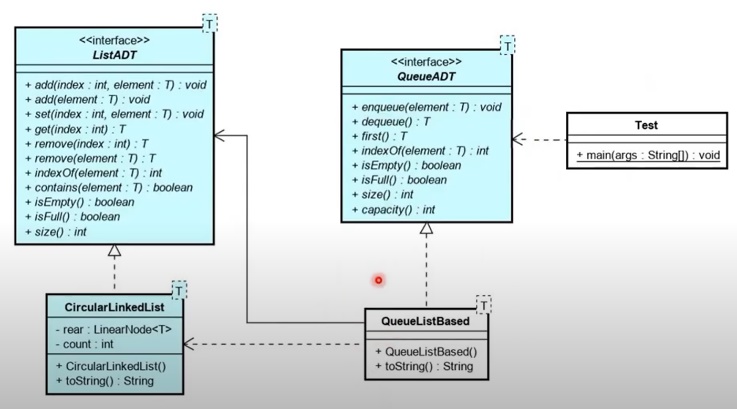
Hvis man nu ikke skal have et synkront resultat på klientens side, f.eks. hvis man kalder en metode ned på server siden, og den ikke giver en retur værdi (som ellers vil give en resultat med det samme-derfor synkront), men er void i stedet for, og serveren stadig skal sende et resultat tilbage (f.eks hvis det er meget tidskrævende), så vil den kalde en metode på klientens side – altså laver den et call-back. Den skal derfor kende til klienten.

Hvis resultatet skal sendes ud til flere eller alle lyttende klienter, så skal klienterne skulle kunne registrere sig selv, så de bliver i stand til at få updates. Server skal holde en liste med klienterne, så den kan broadcaste ud til alle.

**Question 3B:** Adapter design pattern

Describe the pattern using UML diagrams.

* What is the purpose?
* ADAPTER: Implementer et interface, og deligere til en klasse/system som IKKE implementerer det samme interface
* At sætte ting sammen, som er inkompatible – de skal på en måde sættes sammen, så de bliver kompatible
* Problem: hvis et tredjeparts produkt har noget funktionalitet, som man gerne selv vil genbruge i ens program, men man ikke selv kan gå ind at rette i interfacet (f.eks. javas egne klasser, dem kan man ikke rette) ->
* Man har en klasse man ikke kan rette på, og man vil gerne have at den bruger ens eget interface – men det kan man ikke. Hvad gør man så?
  + Man ”pakker den ind” (se billede)
  + Adaptee = den vi gerne vil bruge, men ikke kan rette i
  + Vi vil gerne have implementeret Adaptee’s metode, så den passer i vores interface target, da det er det klienten bruger – Klienten kender kun til vores Target interface, ikke alt det andet
  + Løsning: Vi sætter en klasse ind, som implementere det rigtige interface, og som bruger en instans af Adaptee
    - På den måde, når metoden request() bliver kaldt i Target, så går den ned og henter specificRequest() fra adaptee, ved hjælp af Adapter
    - Hvis specificRequest() returnerede end int, men Target skulle bruge en String, så kan man tilføje ekstra i Adapter, så den konverterer, den skal bare sørge for at tilpasse!

QueueListBased, har en variabel af listADT, og implementerer QueueADT.

QueueListBased sørger for, at de metoder der ”minder” om hinanden i QueueADT og ListADT bliver ens.

Så når metoderne bliver implementeret i QueueListBased fra QueueADT, så skriver man inde i metoder, f.eks.   
enqueue(String name){listADT.add(”name”);}

* What are the different parts involved?

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.

**RMI**

RMI står for Remote method invocation.

Det bruges til at få 2 programmer til at kommunikere med hinanden.

Det fungerer således, at man gerne vil instantiere objekter på én computer, som f.eks. kan være en server, og så skal man kunne lave metodekald direkte på objektet fra klientens side.

Modsat sockets, hvor man skal lave en request for at hente data fra serveren, så kan man med RMI kalde metoder direkte fra Server objektet.

Måden RMI kommunikerer, kan ikke ses fra brugerens side. Når vi bruger sockets, så har vi server- og clientSocketHandlers, som står for at sende data igennem input- og output streams. Dette kan vi ikke se ske, når vi bruger RMI, men det foregår nu stadig ”behind the scenes”.

RMI er et java framework, og både server og client SKAL være java programmer.

Fordele: Når vi bruger sockets, så håndterer serverSocketHandler, at være klient som hopper på serveren, får sin egen tråd, så der sker multithreading. Det behøver vi ikke at tænke på med RMI, det klarer frameworket automatisk for os.

* Dog skal vi stadig have synkronisering på de metoder, som klienter ikke må bruge samtidig.

Figur: Så det er sådan her det ser ud for brugeren, men vi har altså stadig de her socketHandlers, som java styrer for os i baggrunden.

Når det er vi starter vores server program, så har vi først en main metode, som instantierer Server objektet, og samtidig oprettes der også et registry, hvori server objektet puttes i.

Når det så er sket, har vi på en måde lavet vores ”server socket”, som nu bare venter på en connection fra client.

Når client programmet starter, kontakter den så registry, og så vil den lede efter det den skal bruge, altså vores serverprogram.

Når client så har fundet serveren, og lavet sin connection, så får den en instans af stub. Stub bliver en lokalpræsentation af vores server, og det er så denne instans klienten laver metodekald på. Stub er altså en remote proxy, så det er ikke vores rigtige server Objekt, men en slags stedfortræder, som baner vejen dertil.

Klienten kan nu kalde metoder på server-objektet remotely

For at det også skal kunne lade sig gøre, skal server-objektet implemetere Remote interfacet, hvortil alle metoder skal have remoteException.

Serveren skal således også have en constructor, hvor den eksporterer unicastRemoteObject – den laver et kald, så den ekporterer sig selv til at være tilgængelig for Remote interfacet

Hvis man gerne vil have data fra server til client, kan man synkront få det, hvis serveren eksempelvis returnerer en String. Hvis serveren tilgengæld har void som returtype, så skal man lave et call-back til klienten. Det er også en fordel hvis man laver et metodekald som er meget tungt eller tidskrævende, så i stedet for at klienten venter, så kan serveren give resultatet, når den er færdig med at hente data, ved at lave et kald – altså call-back.

Hvis man så gerne vil sende resultatet ud til alle lyttende klienter, som selv sørger for at registrere sig selv som lyttere, så kan serveren få en liste med lyttere, og så broadcaste ud til alle på listen.

**Adapter**

Pointen med adapter er at sætte ting sammen, som er inkompatible.

F.eks. hvis et tredjepartsprodukt har noget funktionalitet, som man gerne selv vil bruge i ens program, men man ikke selv kan gå ind og rette i interfacet, så det bliver tilpasset til ens program, så skal man bruge en adapter. Det kan f.eks. være at man vil bruge en af javas egne klasser, og dem kan man ikke rette i.

Altså har man en klasse, som man ikke kan rette på, men man vil gerne tilpasse den til ens eget interface.

Det man så gør, er at man ”pakker” denne klasse ind i en adapter.

De forskellige parter, som indgår, er derfor vores adaptee, altså den klasse vi gerne vil tilpasse vores eget interface, som er vores target – det er også det klienten bruger – og herimellem har vi vores adapter, som nedarver fra vores interface, men får en instans af adaptee, så den herigennem kan hente relevant data.

Vi vil gerne have implementeret adaptee’s metode, så den passer til target. Så i eksemplet her, så vil vi gerne bruge metoden request, som adapter nedarver, hvori adapter laver metodekald på adaptee, så vi egentlig kommer til at bruge adaptee’s metode.

Hvis f.eks. specificRequest() returnerede en int, og Target skal returnere en String, så kunne man bruge adapteren til at caste int’en til en String.

I forhold til et client/server setup, så kan vi have en ServerModel, som vi gerne vil bruge i vores model på klient siden. Men for at kunne bruge ServerModel’s funktionalitet, og eventuelt tilpasse den, så det passer til vores program, så skal vi bruge den her ModelManager, som fungerer som adapter mellem adaptee, ServerModel, og Target, Model.