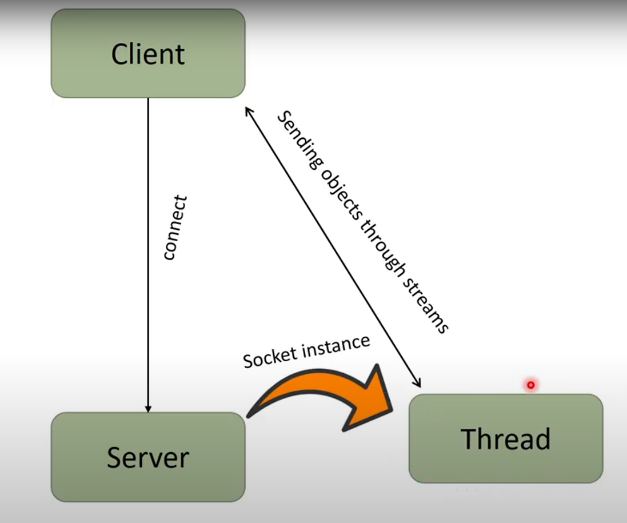
**Question 1A:** Sockets

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

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

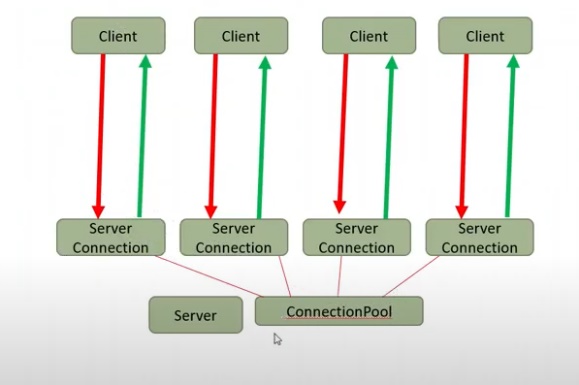
* + Sockets bruges I forbindelse med client-server kommunikation
    - Client-server: flere klienter kan kobles på serveren, og kommunikere med denne, men clients kan ikke kommunikere direkte med hinaden
  + En socket er én endpoint i et to-vejs kommunikationssystem mellem to programmer på samme netværk
  + Socket er bundet op på et ”port-nummer”(det svarer til applikationen)
  + Endpoint er en kombination af ip adresse og port-nummer
  + Koncept:
    - Client sender noget til server, som kan ”ændre” den data, manipulere den, og sende den tilbage i nye formater til client
  + Socket class … ServerSocket class
    - Man laver altid en ServerSocket på server siden(med et port-nummer), så den kan lytte om der er klienter som vil snakke med den
      * ServerSocket serverSocket = new ServerSocket(1234);
      * Socket socket = serverSocket.accept();
    - Linje 2 ovenfor: serverSocket venter på en klient(socket), som har samme port. Når der så er en klient som vil tilkoble samme port, bliver den accepteret
    - Et socket objekt har 2 streams tilknyttet til sig:
      * InputStream = det som sendes fra client, og modtages på serversiden (Set fra serversiden) – eller omvendt er det det server sender til client, og som bliver modtaget på client siden
      * OutputStream = modsat input, så er det det de selv sender afsted til modparten
      * Dem kan man hhv. skrive til eller læse fra
  + Client side: har en Socket, som skal have samme port som ServerSocket, for at koble sig på serveren
    - Her laves også en input- og outputStream
  + Server skal køre én klient færdig, før den kan tage den næste
  + Man bør skrive serverSocket.close() efter while(true) loopet, så man sørge for at få slukket for klienterne
  + PrintWriter og BufferedReader kan bruges ved Streams, men kan typisk kun sende Strings
    - ObjectOutputStream og ObjectInputStream kan sende objekter
  + **Protocol:** system sekvens diagram = kommunikationen i mellem client og server
    - **Det viser hvordan koden på client og serversiden skal passe sammen, hvis man f.eks. glemmer at tage en input, så kan den der venter på svar bare stå og vente – det kan man få overblik med ved brug af system sekvens diagram**

**FLERE KLIENTER SAMTIDIG:**

* Vi skal få klienten til at lytte til serveren i en anden tråd, således klienten kan høre andre ting, og reagere hvis der kommer noget fra serveren
  + Det kræver vi skal have en tråd på server og klient
  + Vi skal have start metode i server og client, og de skal være have en anden klasse, med en mainmetode, som bruger deres .start() metode (ServerMain og ClientMain bør køre start())
* **Serializable**: et interface, man kan sætte på en klasse, så dens state kan konveteres til en byte stream, som så laver en kopi af objektet, som kan sendes mellem server og client
* **Der skal laves en tråd for hver ny klient. Al interaktion med den specifikke klient, sker i gennem den tråd**

**Mere præcist:**

* Vi laver en ServerMain, som kører startmetoden fra Server.
* I server har vi ServerSocket.Accept(), som kører indtil man er ude af try blokken
  + Den bliver altså ved med at tage imod klienter, så længe den er åben
* Når en ny klient (socket) bliver accepteret, så bliver der oprettet en ny tråd for den enkelte klient, som kører ServerSocketHandler(som implementerer runnable), og heri foregår al interaktion mellem klient og server.
* 2 klienter hopper altså ikke på samme tråd, de får oprettet hver deres  **-** men de bruger de samme metoder, som sker i ServerSocketHandler (se billede ovenfor til højre)
* **Ved at give seperate tråde, blokerer clients altså ikke hinanden, hvilket de ville, hvis vi ikke håndterede dem med tråde**

**Sådan får man klienten til at lytte efter inputs:**

* Serveren skal kunne sende noget ud til alle klienterne – broadcast
* Det gør vi ved at have en klasse, der holder eksisterende listeneres/connectede clients – det er ServerSocketHandlers arbejde
* Klienten skal lytte til uforudsete beskeder fra serveren – den kan blive kontaktet, imens den laver noget andet
* Serveren kan bede om at broadcaste til alle sammen, og så skal de reagere – også selvom de er i gang med noget andet
  + Man opretter en liste af clients, dette kan gøres i en Pool klasse, hvor Server får ansvaret for at tilføjer nye klienter til listen, ServerSocketHandler får ansvar for at holde listen, og bruge broadcast() metoden – så hvis f.eks. der er en der sender en besked, så skal ServerSocketHandler i sin run() metode broadcaste den besked ud til alle, som der er i listen’
  + Broadcast() bruger en eller flere metoder fra ServerSocketHandler. F.eks. står der   
    public void broadcast(String message){  
     for(ServerSocketHandler handler : list)  
     { handler.sendMessage(message);}}
  + Hvor sendMessage fra handler siden lyder:  
    public void sendMessage(String message){out.writeObject(new Message(message));}
    - Den tager altså input fra en bruger, og sender inputtet ud til at andre klienter, ved hjælp af listen der er i Pool klassen – broadcast() fra Pool klassen bliver kaldt i run() i handleren
* Use UML to present an overview for your solution
* Show the relevant diagram for a communication protocol for your solution
* Show the related code parts for socket connections and communication following your protocol.
* Explain how to handle multiple clients
  + Why is multithreading needed on the server side?
    - Når en klient sender en request, bliver en tråd genereret, hvori klienten kan kommunikere med serveren.
    - Hvis man gerne vil have mange klienter som sender mange inputs samtidig, så er man nødt til at have flere tråde, da klienterne ellers kommer til at skulle vente på hinanden (stå i kø) i en single-thread server– de kan ikke sende request før den foran er færdig
    - Der er ingen relation i mellem trådene, så hver gang en ny klient connector, bliver der oprettet en ny tråd
    - Hvis der skulle opstå et problem i én af trådene, bliver de andre IKKE påvirket
    - Hvis man gerne vil broadcaste ud til ALLE klienter, som vil prøve at connecte serveren, så er man også nødt til at have multithreading
  + Why is multithreading needed on the client side?

Talepapir Sockets:

* Sockets bruges, når man gerne vil have en client-server forbindelse. Ved client-server forbindelsen kan flere klienter kobles på serveren, og kommunikere med denne, men clients kan ikke kommunikere direkte med hinanden.

En socket er én endpoint i et to-vejs kommunikationssystem mellem to programmer på samme netværk. Den er bunden op på et port-nummer, og endpointet er port-nummer og IP-adresse.

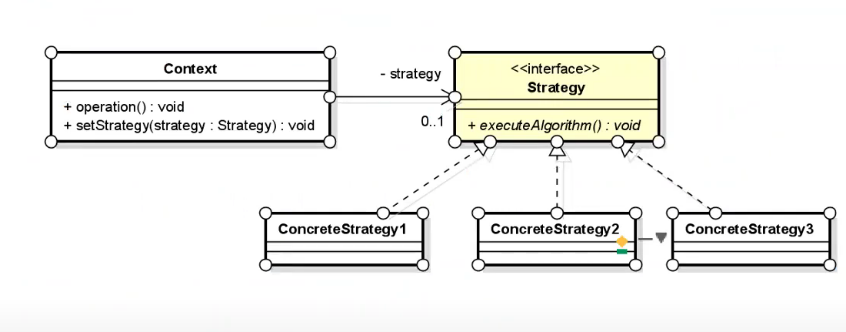
Konceptet med sockets er altså, at en client kan sende noget til serveren, og serveren kan så f.eks. manipulere dataet, eller f.eks. broadcaste beskeden til andre connectede clients. ’

* Kommunikationen: Serveren har en ServerSocket, som er bundet op på et specifik portnummer. Serveren venter så på, at der er klienter som vil koble sig på samme port, og hvis alt går som det skal, accepterer serveren klienten, og der oprettes en ny socket, som klienten bruger til at kommunikere med serveren. Serveren får en ny socket bundet på samme port, med endpoint sat til adresse og port-nummer ved client. Serveren skal have en ny socket, så den fortsat kan bruge sin originale socket til at lytte efter nye clients.
* Client og server kan nu kommunikere ved at skrive og læse fra deres sockets.
* Problem: Client og server kan altså kommunikere, og mange clients kan koble sig på samme port og ServerSocket samtidig, men de bliver sat i kø ved porten, og serveren kan først acceptere den næste client, når den førhen er færdig. – og det er træls!
* Løsning: Serveren kan håndtere clients samtidig, ved brug af tråde. Der skal således oprettes én tråd pr. client connection.  
  Det gøres ret simpelt: lav et while(true) loop, hvori man accepterer connection, og herefter tildeler den connection en tråd
* Klienten opretter altså en connection til server, som opretter en Socket instans, som sendes hen i en tråd, hvor client kan sende objekter frem og tilbage til serveren, gennem streams, i gennem denne tråd.
* For at få klienten til at lytte efter inputs, så skal der oprettes en liste med connectede clients. Dette kan f.eks. gemmes i en klasse. Serveren kan så have en instans af denne klasse, hvorved den kan have metoder, som så kan broadcastes til alle klienterne.
* Multithreading fra server siden: Det er nødvendigt, hvis man gerne vil broadcaste til alle klienter samtidig
* Multithreading fra client siden: Nødvendigt hvis klienter skal have mulighed for at sende inputs samtidig, uden at vente på at dem foran i køen er færdige først
* Det er ingen kommunikation i mellem clients, de skal bruge serveren til at kommunikere med hinanden
* Skulle der ske fejl i én af trådene, så bliver de andre ikke påvirket (stor fordel)!

**Question 1B:** Strategy design pattern

Describe the pattern using UML diagrams.

* What is the purpose?
  + Man definerer et interface, hvorfra man definerer en række klasser, som hver indeholder forskellige algoritmer – de varierer uafhængig af de clients der bruger dem
  + Altså: Klienten bruger en algoritme, i form af et objekt. Objektet kan blive erstattet, hvorved algoritmen altså også bliver erstattet
  + PROBLEM: Hvis vi nu 3 kategorier: Langsom, luntende og hurtig, og vi skal bruge en af hastighederne en anden klasse, så i stedet for at køre if-statements, hvori vi definerer dem alle sammen, så kan man lave strategy-pattern – vi vil altså slippe for de 10000 if-statements
  + Vi flytter i stedet varianternne til seperate klasser, hvor vi erstatter if-statements med polymorphism (betyder mange former, og sker når vi har mange klasser, som er relateret til hinanden med inheritance – f.eks. et interface)



* What are the different parts involved?
  + Context, som skal bruge en operation() som er afhængig af hvilken strategi der skal bruges, får en instans af Strategy, og så kalder den ned på den Strategy, som skal bruges.
  + Operation() er altså afhængig af hvilken Strategy der bliver kaldt
  + Man undgår herved også at rette på koden i Context, og kan enten tilføje/fjerne strategier, som man har lyst
  + I context skal man altså have en instans af det interface, som styrer hvilke strategies der er. Man skal ikke have en instans af de specifikke strategies
  + Med det her pattern, opnår vi open/close princippet: Vi kan extende vores eksisterende kode, uden at modificere i det vi har i for vejen

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.

**Talepapir Sockets + Strategy**

Sockets bruges i forbindelse med client server kommunikation.

I et client/server setup, kan flere klienter koble sig på en server, hvorigennem de så indirekte kan kommunikere via serveren.

En socket et én endpoint i et to-vejs kommunikatiossystem mellem 2 programmer på samme netværk. De er bundet op på et port-nummer – som svarer til applikationen.

Enpoint er derfor en kombination af ip-adresse og portnummer.

Konceptet er så, at klienten kan sende data til serveren, som kan manipulere data, og sende ny data tilbage i andre formater.

Måden man implementere det, er ved at lave en serverSocket på serversiden, med et specifikt portnummer, og så kan den lytte, om der er klienter som vil connecte.

Når en socket fra klient siden, gerne vil koble sig på serveren, så får den tildelt en ny socket, som serverSocket accepterer.

En socket har 2 streams tilknyttet til sig:

* InputStream som parten modtager fra den anden
* OutputStream som er det parten sender til modparten

Det er dem som man bruge til at læse fra eller skrive til.

Klient skal have en socket på sin egen side, som har samme portnummer som ServerSocket, altså et endpoint. Hertil laves der input- og outputStreams, så klienten og serveren kan kommunikere.

Man kan også bruge PrintWriter og BufferedReader til streams, men de bruges typisk kun, hvis man sender Strings frem og tilbge – modsat input- otputStreams som sender objekter

Man kan lave en protokol, som viser kommunikationen imllem client og server. Dette kan vises i et system sekvens diagram. Det giver et godt overblik over, hvordan klienten og serveren skal kommunikere sammen. Hvis man f.eks. glemmer et lave et input på den ene side, så vil begge parter bare stå og vente, uden der sker noget yderligere. Derfor er SSD smart, så man kan se om det går op.

Hvis man gerne vil have flere klienter samtidig, skal vi have en tråd på både server og client, og hertil skal vi have en start metode til dem, samt en main klasse, som starter hhv. server og client.

Når vi opretter en socket til klienten på serverSiden, så sørger vi for at den får sin egen seperate tråd. På den måde laves der en ny tråd, for hver klient som connecter, og al kommunikation sker i gennem denne tråd.

Ved at give seperate tråde, så blokerer clients ikke hinanden, hvilket de ellers ville, hvis vi ikke håndterede dem med tråde.

Så har vi noget der hedder serializable, som er et interface, som fælles klasser, der bliver brugt fra både client/server, skal implementere, således deres state kan konverteres til en byte stream, som laver en kopi af objektet, og så kan det sendes i mellem client og server.

Hvis server gerne vil broadcaste noget ud til alle klienter, så kan man lave en klasse, der holder eksisterende listeners/connectede clients.

Denne liste med clients kan laves i en Pool klasse, hvor Server får ansvaret for at tilføje nye klienter til listen, og en ServerSocketHandler får ansvar for at holde listen, og bruge broadcast() metoden, som oprettes i Pool klassen. Det er så ServerSocketHandler der holder de metoder, som eventuelt skal broadcastes, og Broadcast() i Pool klassen laver derfor metodekald på Handler objektet.

Klienten kan på den måde få uforudsete beskeder fra server, også selvom de er i gang med at lave noget andet.

Multithreading er derfor vigtig på klientsiden, så de kan komme med inputs til serveren samtidig, uden at skulle vente på hinanden. Der er ingen relation i mellem trådene, da en hver klient får tildelt en ny tråd. Det gør at de altså IKKE kan påvirke hinanden.

Det er derfor også vigtigt med multithreading på Server siden, og specielt hvis man gerne vil kunne broadcaste til alle lyttende tråde, så er multithreading en nødvendighed.

**Strategy**

Strategy design pattern er et adfærdsmæssigt design pattern, som gør at man kan definere en række algoritmer, og hertil give dem hver deres separate klasse, hvortil de bliver gjort udskiftelige.

Det kan man gøre ved at definere et interface, hvorfra hver klasse skal implementere interfacet. Algoritmerne varierer uafhængig af de klienter, som bruger dem

Klienten bruger altså en algoritme, i form af et objekt. Objektet kan blive erstattet, hvorved algoritmen så også bliver erstattet.

Hvis man nu antog at man ikke bruge dette setup, så kan vi have 3 forskellige strategier; langom, luntende og løbende, og hvis en klasse skal bygge en rute for disse, på baggrund af deres hastighed, så skal vi til at have en masse if-statements, hvor vi først tjekker hvilken rute vi skal tegne for, og så skal vi bruge den tilhørende algoritme.

Det kan hurtig blive noget bøvl, og fylde rigtig meget i et program.

Derfor flytter vi i stedet varianterne i deres seperate klasser, hvor vi erstatter if-statements med polymorphism, som betyder mange former, hvilket sker, når vi har mange klasser, som er relateret til hinanden med inharitance.

De forksellige ting som indgår, er context, som i eksemplet her skal bruge en operation() metode, som er afhængig af hvilken strategi der skal bruges. Den får derfor en instans af Strategy, hvorigennem den kan vælge en af de nedarvede strategier.

Man undgår herved også at rette på koden i context, og kan enten tilføje/fjerne strategier, som man har lyst.

Med det her pattern, så opnår vi også open/close princippet, således der er mulighed for at udvide programmet, uden at ændre i allerede eksisterende kode.