**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.

**Sockets**

* Sockets bruges i forbindelse med client/server kommunikation
* Client/server setup kan flere klienter koble sig på server
  + Kan kommunikere med server
  + Kan ikke kommunikere direkte med hinanden
* Socket er én endpoint i et to-vejs kommunikationssystem mellem 2 programmer på samme netværk
* De er bundet op på portnummer = svarer til applikationen
* Endpoint derfor = portnummer + ip-adresse
* Konceptet er så = klient kan sende data til server
* Server manipulerer data
* Sender data tilbage i nye formater
* Måden at implementere:
  + Lave serverSocket på serversiden, med specifikt portnummer
    - Den kan lytte efter, om clients vil connecte
    - Når socket fra klientsiden gerne vil koble sig på serverSocket, får den tildelt en ny socket, hvori serverSocket accepterer denne.
* Socket har 2 streams tilknyttet:
  + InputStream = modtager
  + OutputStream = sender
* Klient holder ENDPOINT socket, som har ip-adresse og portnummer.
* Hertil laves input/output streams
  + Så de kan kommunikere
  + Kan også bruge printWriter og BufferedReader til Strings
* Det derfor vigtigt, at sættte streams korrekt op
* Bruge protocol til at tjekke = System Sekvens Diagram
  + Viser kommunikation i mellem client/server
  + Giver overblik
    - Man kan se om man misser en inputStream/outputStream
    - Ville resultere i at begge bare vil vente, uden der sker noget
  + Derfor SSD smart
* Hvis flere klienter samtidig
  + Tråd på server og client side
    - Hertil skal de have start metoder, og en main klasse, som starter hhv. server og client
* Når vi tildeler en socket til klienten, når de connector til ServerSocket
  + Tildelt separate tråde
  + Ny tråd for hver client som connector
  + Al kommunikation sker via tråden
* Hvis client og server vil sende et fælles objekt i mellem sig
  + SERIALIZABLE
    - Interface som fælles klassen skal implementere
    - Således deres state bliver konverteret til byte stream
      * Laver kopi af objekt, og sender i mellem client/server
* Hvis server vil broadcaste til alle klienter
  + Opret klasse, som indeholder liste med listeners
  + Server har ansvaret for, at tilføje lyttende clients til listen
  + Lav serverSocketHandler, som holder listen
* Multithreading vigtigt fordi:
  + Klienter kan komme med inputs samtidig uden at vente på hinanden
  + Ingen relation i mellem trådene
  + Serveren kan broadcaste ud til alle lyttende tråde

**Strategy**

* Er et adfærdsmæssigt design pattern
  + Gør at man kan definere en række algoritmer
  + Inddele separate klasser
    - Bliver gjort udskiftelige
* Det gør man ved at definere interface
  + Hver klasse skal implementere
  + Algoritmer varierer uafhængig af de klienter, som bruger dem
* Klienten bruger altså en algoritme, i form af et objekt
  + Objektet erstattes = algoritme erstattes
* Hvis man antog ikke bruge dette setup
  + 3 forskellige algoritmer;
    - Langsom
    - Luntende
    - Løbende
  + Tegne rute, på baggrund af hastighed
    - Mange if-statements
    - Først tjek hvilken rute der tegnes for -> algoritme
  + BØVL og fylder
* Derfor flytter vi varianterne ind i separate klasser
  + Hvor vi erstatter if-statements med POLYMOPHISM
    - Betyder mange former
    - Sker når vi har mange klasser, som er relateret til hinanden med inharitance
* Forskellige dele som indgår:
  + Context, som eksempel operation() metode
    - Afhængig af strategi
    - Får instans af Strategy
    - Hvorigennem den kan vælge en af de nedarvede strategier
* Man undgår herved at rette på context, og kan enten tilføje/fjerne, som man har lyst
* Opnår OPEN/CLOSE