**Talepapir RMI + adapter**

**RMI**

RMI står for Remote method invocation.

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

Det fungere 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.

**RMI**

* Remote method invocation
* Bruges til at få 2 programmer til at kommunikere
* Fungerer ved instantiere objekter på én computer (server) – direkte metodekald på objekt fra klientsiden
* Modsat sockets = request for at hente data fra server … RMI = kalde metoder direkte fra server objektet
* Måden RMI kommunikerer
  + Kan ikke ses fra brugerens side
  + Sockets = server- og clientSocketHandler = står for at sende data i gennem input- og outputStreams
  + Det kan vi ikke se med RMI, foregår ”behind the scenes”
* RMI = Java framework – begge programmer java
* FORDELE: ved sockets, håndtere serverSocketHandler, at hver klient får en tråd, når de connecter
  + Multi-threading
* Behøves ikke i RMI, Java ordner det for os
  + Stadig synkronisering på metoder, som klienter ikke må bruge samtidig
* Starter server program = main metode instantierer server objekt
  + Samtidig oprettes Registry, hvori objektet puttes
  + På den måde er vores ”ServerSocket” blevet oprettet, som venter på connection fra client
* Når klienten starter, kontakter den registry, og leder efter server objektet den skal bruge
* Når client har fået connection, får den en instans af stub
  + Stub = lokal repræsentation af server
  + Denne instans klienten laver metodekald påå
  + Stub = remote proxy, så ikke det rigtige server objekt, med stedfortræder, som baner vej
  + Klient kan kalde metoder på server-objektet remotely
* Server = implementer remote interface = metoder får remoteException
* Server – constructor = eksporterer sig selv til unicastRemoteObject, så den bliver tilgængelig for remote interfacet
* Data server->klient = returner String = synkront
* Void som returmetode = call-back til klient
  + Metodekald som er tung og tidskrævende = klient skal vente
    - Løsning = call-back
    - Serveren giver resultatet tilbage når det er klar, via call-back
* Call-back til flere klienter = broadcast
* Klienter registrerer sig som lytter, og server holder liste -> broadcaster

**Adapter**

* Pointen er = sætte ting sammen, som er inkompatible
* F.eks. tredjepartprodukt har noget funktionalitet = gerne bruge i eget program
  + Man kan ikke selv gå ind og rette i interfacet, så det tilpasses ens eget
  + Skal man bruge adapter
* F.eks. javas egne klasser – kan man ikke rette i
* Altså = har klasse eller interface, som man ikke kan rette på
  + Vil gerne tilpasse ens eget interface
  + = ”pakker” den ind
* Forskellige parter:
  + Adaptee – den vi vil have tilpasset til eget interface
  + Target = det interface det skal tilpasses til
    - Det klienten bruger
  + Adapter = i mellem de to, som nedarver interface, instans af adaptee
    - Herigennem hente relevant data
* Vi vil implementere adaptee’s metoder, så det passer til target
  + I eksemplet: bruge metode request() – som adapter nedarver
  + Hvori adapter laver metodekald på adaptee instans
  + Eksempel: Adaptee returnerer int, interface skal bruge int, ADAPTER CASTER
* Ift. klient/server setup:
  + ServerModel, som vi vil bruge i vores Model interface på klient side
  + For at bruge ServerModels funktionalitet
    - Evt. tilpasse data = så det passer til vores program
    - ModelManager
    - Adapter/Target/adaptee/client