Stian Hvatum – MTDT
(hvatum)

TDT4186 Operativsystemer

Øving 2

I denne øvingen har jeg implementert Barber, Doorman og CustomerQueue, i tillegg har jeg lagt inn «tilfeldige variable» i Globals, samt et GUI til å justere disse.

Følgende er gjort i CustomerQueue:

Konstuktøren tar vare på GUI, slik at vi kan kalle metoder på denne videre inne fra vårt objekt. Konstuktøren lager også et *array* av *Customer*, som blir vår indre representasjon av venterommet/den sirkulære bufferen. Vi lager også to heltalsvariable, *queueStart* og *nextPos*, disse sier hvem som er den første i køen, og hvor nestemann kan sette seg dersom det kommer en ny kunde.

Metododen *addCustomer* legger til en ny kunde i venterommet, dersom det er plass, hvis ikke vil den kaste en *IndexOutOfBoundsException*. Når det legges til en ny kunde, oppdateres GUI tilsvarende, og *nextPos* økes med 1. Får å sikre at vi får et sirkulært buffer, kan man enten modulere *nextPos* og *queueStart* ved innsetting, eller ved referering til *array*. Jeg valgte siste måten, da dette gjør av i kan holde telling på hvor mange kunder vi har hatt totalt, dersom man noen gang skulle ønske det. Elementer i arrayet refereres altså med *array[nextPos++ % array.length]*.

Metoden *removeCustomer* fungere på samme måte, men fjerner et element fra *array*et, inkrementerer *queueStart*, og returnerer elementet. Elementer i *array*et refereres her med *array[queueStart++ % array.length]*. Dersom det ikke finnes flere kunder,

Metodene *hasCustomers*, *hasMoreSpace* og *getNextCustomerPos* gjør akkurat det de heter, altså svarer på om det finnes kunder i køen, om det er ledig plass i køen, og hvilken posisjon har føstemann (dette er altså *nextPos % array.length*).

Følgende er gjort i Doorman:

Konstruktøren tar vare på køen og GUI, slik at disse kan jobbes på senere. Vi oppretter også en variabel *boolean stop*, slik at vi kan stoppe tråden når vi får beskjed om det. Denne starter som *true*, og settes til *false* ved *startThread()*, og til *true* igjen ved *stopThread()*.

Jeg lot *Doorman* implementere *Runnable*, og implementerte metoden *run()*. Her følger Doorman historien fra oppgaveteksten, vi starter med å hente inn en kunde, deretter vekker vi en barberer, og går og sover litt. Dette gjør vi helt til enten *stop* blir *true* (programmet avsluttes), eller til køen er full, og *queue.hasCustomers()* returnerer *false*.

startThread oppretter en ny tråd som kjører *run*-metoden, samt setter *stop* til *false*. Vi tar vare på tråden i en variabel, slik at vi kan stoppe den kontrollert når programmet avsluttes.

stopThread setter *stop* til *true*, og forstyrrer tråden med *queue.interrupt()*, da er vi sikre på at denne tråden er stoppet.

Stian Hvatum – MTDT
(hvatum)

Følgende er gjort i Barber

Dette objektet oppfører seg ganske likt som Doorman, konstruktøren er lik, bortsett fra at den også lagrer *pos*. *startThread* og *stopThread* gjør også akkurat det samme, her også implementer vi *runnable*. Barberens historie er litt mer komplisert en dørmannen, så her starter vi med å dagdrømme (som barbereren alltid gjør før han er klar), deretter venter vi til vi har ledige kunder i køen. Når vi har fått kunder i køen, henter vi ut førstemann, oppdaterer GUI tilsvarende, og forteller dørmannen at det er ledig plass i køen. Frisøren slipper låsen på køen før han begynner å jobbe, hvis ikke får ikke dørmannen hentet flere kunder så lenge noen klippes. GUI oppdateres, og vi går tilbake til dagdrømming.

Følgende er gjort i Cashier

Denne klassen opprettet jeg selv. Klassen er meget enkel, og holder styr på betalingene som er gjort. Klassen har en enkel konstruktør, som setter alle verdier til 0, og holder Gui, slik at vi kan sette inn tekst. Klassen har metodene depositMoney og getTodaysTotal. Sistnevnte returnerer en status-streng til å fortelle hvor mange som er blitt betjent i dag, og hvor mye de betalte til sammen. Metoden depositMoney er en synkronisert metode som setter inn penger. Metoden er synkronisert, da det er en fare for at to kunder prøver å betale samtidig, og det kan fort gå galt. Cashier er en Singleton, fordi det skal kun være en av den, og da slipper vi å endre så mye av rammeverkskoden for å ta den i bruk.

For å oppnå Producer/Consumer

Oppgaven er basert på producer/consumer-mønsteret. Dette går ut på å ha to (eller flere) separate objekter som skriver og leser til samme kø (eller lignende). I denne oppgaven har jeg implementert prinsippet med å bruke CustomerQueue som en monitor, samt benytter jeg syncronized-blokker i Doorman og Barber som bruker denne felles monitoren. De jobber på til køen er tom eller full (ettersom man er dørmann eller frisør), deretter går de på vent, og de vekker hverandre hver gang de har utført en deloppgave. I Javakode blir dette utført med

```
syncronized(monitor) {
        while(condition) monitor.wait();
        ...
        monitor.notify();
}
```