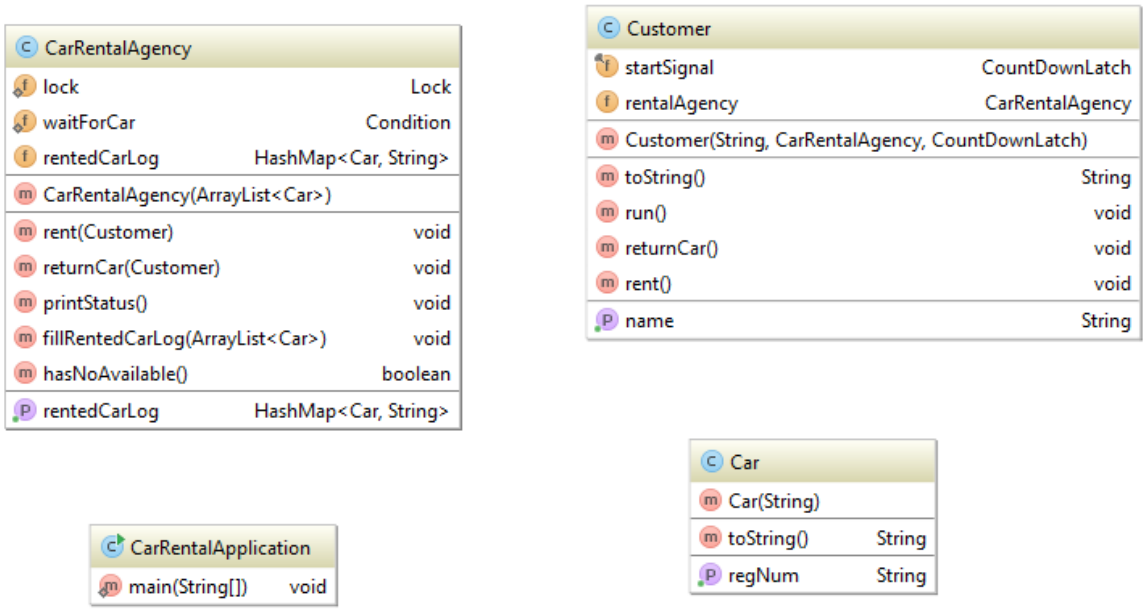
Innlevering 1 / PG4100 (Notat)

I dette notatet vil jeg forklare hvordan applikasjonen jeg har laget er satt sammen og fungerer. Jeg tok utgangspunktet i oppgavens tekst og følgende hoved vilkår som er beskrevet i oppgaven:

1. **Løsningen skal være trådsikker**: flere tråder skal kunne ha adgang til felles ressurser parallelt mens selve ressursene ska hånderes slik at flere tråd ikke kan endre status av samme objekt samtidig.
2. **Løsningen skal testes ved hjelp av unit testing**: det skal bekreftes at funksjonalitet i de kritiske klassene tilfredstille krav fra opppgaveteksten.

Arkitektur

For å lage en løsning som er satt sammen på en fornuftig måte tok jeg utgangspunkt i virkelig modell for et bilutleiefirma. Sånn modell forutesetter at tre hoved objekter er til stedet. Disse objektene er selve utleie firma, klient og bil. Derfor var det logisk å lage klasser som svarer til disse virkelige objektene. Så lenge hovedvekt i oppgaven legges på trådsikkehet og løsningen skal være så pass ”liten”, var det ingen behov for bruk av arv og interfacer, samt noe design patterns. Samtidig finnes det potensiell for videreutvikling av løsningen ved hjelp av de ovennevnte metodene.



Figuren over viser klassene løsningen består av.

Car klassen har bare et felt ”regNum” som svar til bilens registreringsnummer. Denne klassen kan utvides med andre felter som kan inneholde informasjon om forskjellige egenskaper som merke, antall seter, farge ol. Det kan brukes OOM eller klassen kan erstattes med tilkobling til en database.

Customer klassen har tre felt ”navn”, ”startSignal” og ”rentalAgency”. Som Car klassen kan den utvides med flere felt som skal inneholde ekstra opplysninger om en kunde. Klassen implementerer Runnable interface og ved hjelp av private hjelpe metoder rent() og returnCar() som metode run() bruker, renter

biler hos bilutleie firma og returnerer de tilbake. Dette skjer bestandig så lenge brukeren ikke stopper programmet.

CarRentalAgency klassen holder styr på kritiske data som info om biler som er ledige eller opptatt, om det er flere bil som kan rentes og om det er registrert at bil er returnert. Denne klassen som inneholder metodene rent() og returnCar() som skal programmeres trådsikkert. CarRentalAgency instansieres med array list av biler som biluteleie firma disponerer. Ved hjelp av metoden fillRentedCarLog() påføres det informasjon om alle bilene (dvs registrerings numrene) inn i register og det markeres at alle bilene er tilgjengelige for utleie. Registeren er realisert som HashMap hvor nøklene er registreringsnumrene og verdier svarer til kundenes navn eller settes til ”available” hvis respiktiv bil ikke er utleid.

CarRentalApplication klass instansierer nødvendige objekter og starter trådene som representerer kunder. Her lagges det array list av bilene som brukes videre for å instansiere CarRentalAgency klassen. Array listen er hardcodet men løsningen kan endres slik at den kan fylles med data fra en database. I dette tilfelle kan Car klassen fjernes fra applikasjonen. Det er brukerern som bestemmer hvor mange kunder skal bruke bilutleiefirma. Dette realiseres ved bruk av basic JOptionPane funlksjonalitet med default innstilling som forutser at det skal opprettes 10 kunder. Da informasjonen om antall kunder er innført, skal brukeren ved hjelp av JOptionPane dialog boksen taste navn på alle kundene. Opplysninger fra brukerens input, samt CarRentalAgency objektet og CountDownLatch objektet brukes for å instansiere Customer objekter. Når brukeren innfører navn på 5. kunde starter kundene å leie biler mens brukeren fortsetter med input. Da alle kunbdene blir opprettet, får brukeren forespørsel om å stoppe applikasjonen ved trykk på en ”OK” knappen (realiseres ved hjelp av JOptionPane).

Trådhåndtering

Customer klassen har basic trådhondterings elementer. I metode run() som man skal overskrive når en klasse implementerer Runnable interface, brukes det static metode sleep() for å sikre at en tråd venter konkret tid før den prøver å leie bil og før den returnerer bilen. Tråden kan bli avbrytet mens den ”sover” derfor er det nødvendig å bruke enten try/catch blokk eller deklarere at metoden kaster InterruptException. Ifølge oppgave teksten skal trådene starte å leie biler når det er opprettet 5 tråder. Dette realøiseres ved hjelp av CountDownLatch klassen som instansieres i CarRentalApplication klassen og sendes til hver enkel tråd under instansiering.

CarRentalApplication klassen starter tråder som svarer til kundene og gir beskjed til dem å begynne å leie og returnere biler når 5 tråder er opptretet. Trådene informeres om at de skal starte ved hjelp av countDown() metoden til CountDownLatch klassen.

CarRentalAgency har to metoder rent() og returnCar() som endrer registrer med bilenes status når en tråd prøver å lei / returnere en bil. Registeren er kritisk ressurs som skal være tilgjengelig bare for en tråd. Er det ikke garantert, kan flere tråd leie samme bil siden de kan henvende seg til bilutleiefirma samtidig. Dette er asktuelt for innlevering av en bil. For å sikre de ovennevnte krav kunne man bruke forskjellige måter. Siden det kritiske ressurset er HashMap kan det erstattes med en trådsikker datastruktur som man får ved bruk av statisk metode synchronizedMap(). Denne løsningen kunne gjøre klassen kortere og evt. mer forståelig, men begrense mulighet til å bruke Condition klasse for å informere andre tråd når kritiske ressurser er tilgjengelige ighen. Det var også mulig å gjøre rent() og returnCar() metodene synkroniserte, men det anbefales nå å bruke Lock objekter for å gjøre tårdhåndering sikker. Lock objekter gir mer fleksibilitet og denne måten å håndtere tråd på er mer modern og avansert. Det kunne også være mulig å bruke synchronize expretion for å synkronisere bare

en del av metode, men denne måten har samme ulemper som bruk av synkroniserte metoder og samtidige bruker mer minneressurser enn andre metoder for trådsikker programmering.

For å sikre at kritiske ressurser kan være tilgjengelig for bare en tråd brukes det Lock objektet i CarRentalAgency klassen. En tråd som kaller på rent() metoden før de andre tilegner låsen og løslater den når tråden er ferdig. Samme låsen brukes både for å leie og returnere bil. Dette sørger for at de kritiske ressursene kan endres bare av en tråd.

For å sikre at låsen blir løslat selv om en tråd er avbryt under eksekvering brukes det try/finally blokk i løsningen. Siden en tråd kan vente på signal fra en annen tråd, som endrer de kritiske ressursene, og være avbrut mens den venter brukes det try/catch block for å fange evt. InterruptedException.

Alle disse ovennevnte tiltak sørger for at CarRentalAgency klassen håndtrer tråd på sikker måte og forhindrer upålitelig endring av de kritiske ressursene.

Testing

Løsningen innholder en mappe ”test” hvor det ligger test suites for alle tre hovedklassene. Kodedekning finnes i mappe ”coverage_report”. I følge rapporten ble CarRentalApplication klassen ikke testen. Dette ansees som unødvendig siden denne klassen bare instansierer nødvendige objekter og starter tråder. Customer klassen har også begrenset kodedekning siden den ikke har metoder med vanskelig logikk som må testes. Testene for Car og Customer klassene er proforma tester og ble brukt bare i første fasen under Test Driven Development av utkastet til løsningen.

Det viktigste test suitet er CarRentalAgencyTest klassen som innholder metoder som tester kritisk funksjonalitet av den respektive klassen.

Det brukes mock objekter for å instansiere array list med biler og mock objekter for kundene når det lar seg gjøre.

Metodene testCustomerRentsCar_CustomersNameIsRegistered() og testCustomerReturnsCar_ CustomersNameIsDeletedFromRegister() bekrefter at rent() og returnCar() metoder fungerer som tiltenkt.

Metode testMultipleThreadsRentCars_JustOneCarCanBeRentedByOneCustomer() tester trådsikkerhet av rent() metoden. Det oppretthåldes tre Customer objekter med ekte navn og instans av CarAgency klasse og et mock objekt av CountDownLatch klassen. Instanser av Customer klassen brukes bare for å lage annonime objekter av Runnable klasse som kaller på rent() metode til CarRentalAgency klassen, siden rent() metode må kalles med parameter av klasse Customer.Etterpå startes det alle trådene og hoved tåden venter på de andre trådene blir ferdige. Etterpå sjekkes det hvor mange ganger navn på kundene er registrert i regiseren hos CarRentalAgency objketet. Det forventes at alle navn på alle tre kundene er registrert bare en gang og at det finnes ikke noen bil som er ledig. Med andre ord bekreftes det at det finnes ikke noen bil som ble leid av flere kunder samtidig. Hvis dette skjer, blir da det ledige viler i registeren og ikke alle kunbder er registrert.

Metoden hugeNumberOfCustomersTryToRent() bruker samme fremgangsmåte som sist nevnte metoden men tester at 1000 kunder (tråder) prøver å leie 1000 biler. Testen eliminerer risiko for at sikker trådhåndtering ikke er tilegnet for store mengder av tråder.

Metoden testMultipleCarsReturned_AllCarsAreAvailable() tester funksjonalitet i returnCar() metoden. Den baseres på samme forutsetninger som de to sist nevnte metoder og bruker samme fremgangsmåte med nødvendige tilpassniger.

Videreutvikling

Løsningen har god kapasitet for videreutvikling ved bruk av database tilkopling. Car klassen kan erstattes med en metode i CarRentalApplication som lager array list av biler på bakgrunn av data som man får fra en tabell i en database.

Det kan også lages bedre GUI med utvidet funksjonalitet (f.eks. lagring av debug utskrift til en fil, felt for å innføre info om biler som bilutleiefirma disponerer o.l.).

Det kan evt. brukes Faktory design pattern for å lage Car objekter om denne klassen utvides med forskjellige typer biler.

Konstruktøren i Customer classen kan endres slik at den kreer bare en parameter – name mens andre to parametere kan settes ved hjelp av setter metoder.