NTNU

TTK4235 Tilpassede datasystemer

Gruppe 81

Heisprosjektoppgave

Authors:
Martin Borge Heir
Håvard Brenne

1. mars 2020

Introduksjon

I dette prosjektet har vi programmert en modellheis i C til å oppfylle visse spesifikasjonskrav. Vi fikk utdelt driverfiler for heisen og har skrevet et program for hvordan heisen skal reagere på inputs. Vi har også dokumentert programmet i doxygen og laget UML diagrammer for arkitekturen. Denne rapporten har som hensikt å presentere denne arbeidsprosessen og programmet som helhet. Vi har benyttet V-modellen i stor grad i utviklingsprosessen og denne rapporten skal forsøke å gjenspeile det.

Innhold

1 Arkitektur

1.1 Oppførsel og tilstandsmaskinen

Programarkitekturen baserer seg på en tilstandsmaskinmodell. Det vil si at hver tilstand er en funksjon og at programmet vil bytte mellom å kjøre i hver av disse tilstandene, basert på input den får. Programmet vil utføre spesifikke funksjoner ved inngang og utgang av en tilstand, og kjøre en løkke så lenge vi befinner oss i en tilstand.

Init state:

Er tilstanden der heisen blir initialisert. Her blir tilstandsvariabler nullstilt og heisen beveger seg nedover til den treffer en etasje, og dermed når en definert tilstand. Den vil så gå over til standby state. Dersom nødknappen trykkes, vil heisen gå i emergency state.

Standby state:

I standby state vil heisen motta ordre og deretter bestemme neste state. Dersom nødknappen trykkes, vil heisen gå i emergency state.

Going up state:

Heisen vil gå oppover. Når den når en etasje vil den stoppe om det er noen ordre i etasjen i sin retning. Den vil så gå tilbake til standby. Dersom nødknappen trykkes, vil heisen gå i emergency state.

Going down state:

Tilsvarende som going up state, bare i retning nedover.

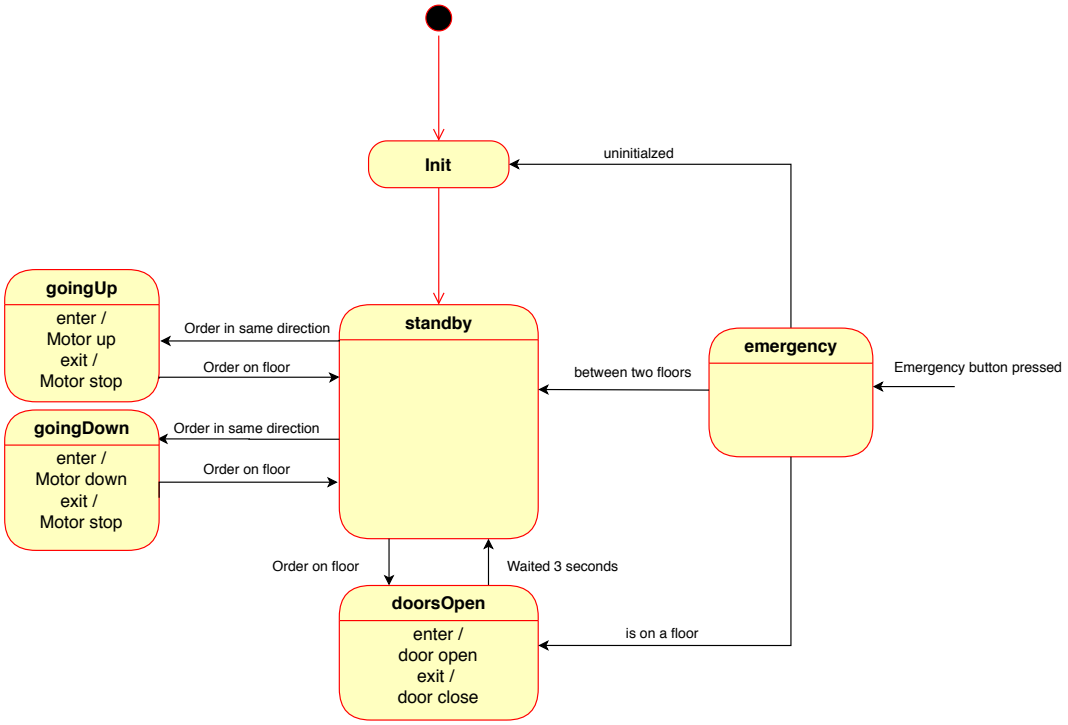
Doors open state:

Dørene på heisen vil åpnes. Etter tre sekunder vil heisen gå tilbake til standby. Dersom obstruksjonsbryteren er togglet vil heisen forbli i tilstanden. Dersom nødknappen trykkes, vil heisen gå i emergency state.

Emergency state:

Heisen vil kansellere alle ordre og slutte å motta bestillinger så lenge knappen holdes inne. Dersom heisen er i en etasje vil dørene åpnes og heisen vil gå til doors open state. Dersom heisen er mellom to etasjer vil heisen gå til standby. Dersom heisen er uinitialisert vil den gå til init state.

I tillegg vil alle states utenom Emergency og Init ta bestillinger og oppdatere bestillingslys kontinuerlig.

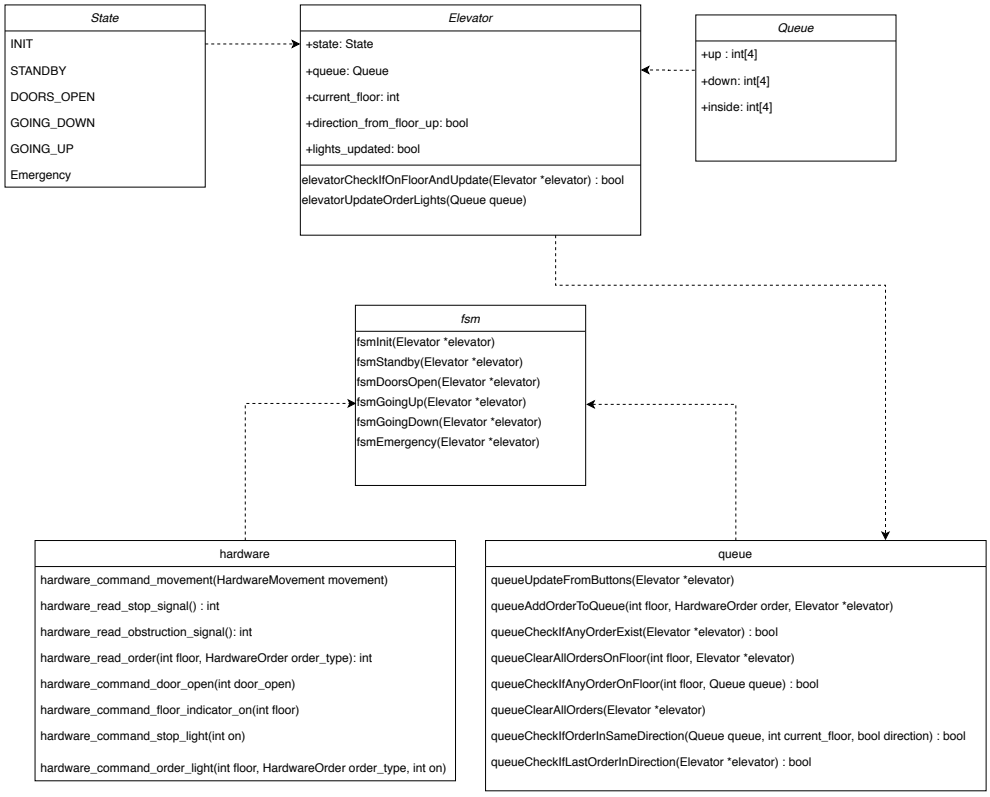


Figur 1: Figuren viser tilstandsdiagrammet for programmet

1.2 Klassestrukturen

Klassestrukturen til programmet er sentrert rundt tilstandsmaskinen. Systemets tilstandsvariabler er organisert i et Elevator objekt med et par tilhørende funksjoner. Tilstandsmaskinen trenger å kunne lese og oppdatere tilstandsvariablene og er derfor avhengig av Elevator objektet. Elevator objektet er i sin tur avhengig av listen alle ordrene er lagret i, Queue, og State strukturen som organiserer systemets tilstander.

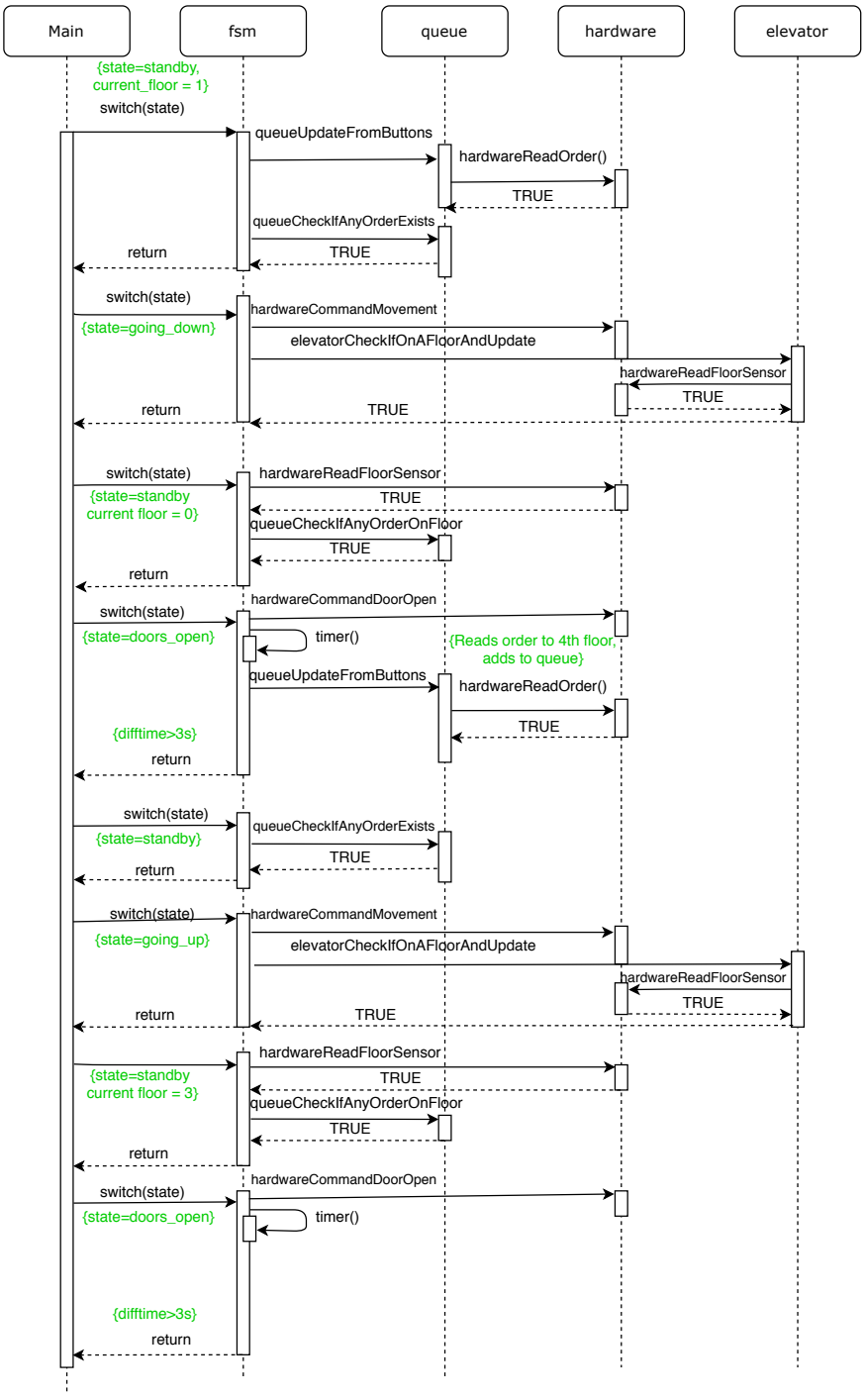
Videre har tilstandsmaskinen (fsm) tilgang til to API biblioteker. Det ene biblioteket "hardware" gir direkte tilgang til heisens aktuatorer, lysenheter og input-knapper. Det andre biblioteket "queue" inneholder funksjoner for sjekking og endring av bestillingslistene.



Figur 2: Figuren viser klassediagrammet for programmet

1.3 Modulsammensetting

For å se hvordan modulene fungerer sammen er det satt opp et sekvensdiagram. For enkelhets skyld er kun deler av funksjonaliteten fremhevet, men det er nok til å se hvordan modulene fungerer sammen. Fra diagrammet ser man at de fleste funksjonskall starter i fsm, bruker funksjoner i queue og elevator, som videre interakterer med hardware.



Figur 3: Figuren viser sekvensdiagrammet for oppgitt oppførsel

Figur 3 viser følgende sekvens:

1. Heisen står stille i 2. etasje med døren lukket.
2. En person bestiller heisen fra 1. etasje.
3. Når heisen ankommer, går personen inn i heisen og bestiller 4. etasje.
4. Heisen ankommer 4. etasje, og personen går av.
5. Etter 3 sekunder lukker dørene til heisen seg.

1.4 Valg av arkitektur

Hovedstyrken med en tilstandsmaskinarkitektur slik vi har designet det er at det skaper god oversikt og er enkelt å debugge når man kan lokalisere feil og mangler til hver enkelt state. Man kan også utvikle states uavhengig av hverandre, slik at man kan skrive og teste ut de ulike statsene hver for seg uten at det påvirker resten av programmet. Det at de er uavhengige gjør også at en endring i en state som regel kun vil påvirke den aktuelle staten.

En annen hovedstyrke med vår arkitektur er elevator objektet som holder på de ulike tilstandsvariablene til systemet. Det er ryddig å ha tilstandsvariablene samlet på et sted, samtidig som at det gjør det enkelt for API'er å lese og endre tilstandsvariablene. Da slipper man også å benytte seg av globale variable som fort kan lage rot i strukturen.

I tillegg er systemet designet slik at det svært enkelt kan utvides til flere etasjer. Det er omtrent så enkelt som å endre en variabel.

2 Moduldesign

2.1 Queue

Ordrelisten vår valgte vi å organisere i et objekt med tre arrays av typen bool, basert på om ordren er en oppover-, nedover-, eller inni-heisen-ordre. Posisjonen i arrayen angir etasjen og 1 eller 0 angir om det er en ordre der eller ikke. Dette gir en enkel oversikt over hvilke ordre som finnes, samt at det er lett å legge til og fjerne ordre.

2.2 State

Vi implementerte et objekt av typen enum for å angi de gyldige tilstandene til programmet. Dette er en lurt da det gjør det vanskelig å gi systemet en udefinert tilstand

og det er enkelt å implementere tilstandsmaskin ved bruk av switch funksjon.

2.3 Elevator

Vi valgte å samle alle systemets tilstandsvariabler i en struct. Her er både *Queue* og *State* lagret som beskrevet over. I tillegg holder Elevator objektet på *current_floor*, *direction_from_floor_up*, og *lights_updated*. Nåværende etasje og siste retning fra forrige etasje er to tilstandsvariable som er intuitive å ha med. Inkluderingen av *lights-updated* er lurt da denne tilstanden forteller programmet når det trenger å oppdatere lysene på en enkel måte.

2.4 queue

Vi trengte en modul som raskt og enkelt kunne gi tilgang til den viktigste informasjonen og gjøre endringer i ordrene. Vi tenkte det lureste var å oranisere disse API'ene i et bibliotek med dette formålet, da det er oversiktlig og gir et enkelt grensesnitt.

2.5 hardware

Hardware biblioteket fungerer som en API forbindelse til heisdriverene. Siden vi ikke skrev disse funksjoene skal vi heller ikke kommentere de ytteligere.

2.6 Tilstandsmaskinen (fsm)

Tilstandsmaskinen er selve hovedkjernen i programmet. Hver state fungerer i prinsippet likt med entry og exit funksjoner samt inside hendelser som kjører i en while løkke. Vi ønsket å strukture de så likt som mulig for å øke lesbarheten. Alle statsene tar inn Elevator objektet som argument. Ettersom vi allerede har dekket hva de ulike statsene gjør skal vi bare kjapt kommentere noen implementasjonsvalg.

Et felles implementasjonsvalg for tilstandsmaskinen var at vi valgte å basere endringen av bestillingslys basert på hvilke ordre som er i køen, og oppdatere de kontinuerlig i hver løkke i de aktuelle statsene i stedet for å toggle manuelt hver gang en knapp trykkes inn eller en ordre fjernes. Dette gjør håndteringen av bestillingslys svært enkelt og gjør at vi kun trenger å endre ordenslistene.

Et alternativt implementasjonsvalg kunne vært å initialisere heisen før programmet

nådde tilstandsmaskinen. Dette gikk vi bort ifra da vi innså verdien av å kunne ha muligeten til å reinitialisere heisen dersom vi eller andre utviklere skulle ha behov for det senere.

Vi valgte å implementere standby slik at den fungerer som en valgstasjon i tilstandsmaskinen. Dermed springer de fleste states ut fra standby og ender opp igjen i standby. Dette gjør det enkelt å se og endre oppførselen til heisen da denne koden er samlet i en state.

3 Testing

Med vår tilstansmaskinarkitektur er de fem statene relativt uavhengige, som betyr at de lar seg teste svært godt hver for seg uten at de påvirker resten av programmet.

Da vi implementerte koden, kodet vi statene hver for seg og testet grundig at den enkelte state oppførte seg som den skulle før vi skrev neste state. For hver state la vi til nødvendige API'er i bibliotekene som denne staten trengte for å fungere. Vi startet med første state Init, skrev så standby, osv. Da vi hadde skrevet en ny state kunne vi enkelt teste at den fungerte med forrige state og på denne måten sikret vi at programmet oppførte seg som forventet gjennom hele utviklingsprosessen frem til siste state var skrevet. Vi trengte derfor ikke ha en egen integrasjonstesting da vi alt hadde testet alle statsene sammen kontinuerlig. Dette var en enkel måte å teste prosjektet på med tanke på oppkobling til heisen, initialiseringen og så videre.

Å teste en enkelt state viste seg å ikke være så vanskelig. Som regel gikk prosedyren ut på å sette breakpoints i GDB ved det aktuelle området i den aktuelle staten og deretter bruke print funksjonen i GDB til å printe Elevator objektet med alle tilstandsvariablene. Da hadde vi full oversikt over hvordan ting hadde endret seg, og vi kunne så fortsette programmet, gi nye innputs, stoppe ved breakpoint, og printe det samme objektet om nødvendig. Slik testet vi blant annet at køsystmet, etasjen, og retningen oppdaterte seg som forventet.

Til slutt ble hele systemet testet grundig opp mot FAT. Her ble det kjørt en rekke edge-cases, for eksempel svært mange bestillinger, trykke på flere knapper samtidig, nødstoppe flere ganger mellom en etasje osv.

4 Diskusjon

Styrken med at alle API'er og moduler som kaller en Elevator peker kan endre fritt på alle tilstandsvariable, er også en stor svakhet i programmet. Dette kan skape store problemer hvis man er uforsiktig i sin design av API'er. Det man egentlig ønsker seg er en tydelig avgrensning på hvilke tilstandsvariabler en funksjon kan endre på og om den kan endre noe i det hele tatt eller bare lese. I et objektorientert språk som i C++ kunne man enkelt implementert klasser og arvet både private og public, og man kunne definert medlemsfunksjoner som const. Mange av våre API'er kunne med fordel hatt tydligere argumenter i funksjonsdeklarasjone enn å bare kalle hele Elevator pekeren. Det kunne blitt gjort ved å deklarere tilstandsvariablene som pekere i Elevator objektet.

I tilstandsmaskinen hadde vi flere muligheter vi vurderte å implementere. Blant annet kunne vi ha gått direkte fra en av bevegelsestilstandene og rett til *open_doors* hvis man visste at det var en bestilling på etasjen. Det kan argumenteres for at heisen egentlig ikke befinner seg i noe *standby* i denne transisjonen. Likevel gjør dette valget grensesnittet enklere, og mindre innviklet. Det må uansett være en måte å komme fra en "ekte" *standby* til *doors_open*, og dette er nøyaktig den samme sjekken som man bruker fra en bevegelsestilstand til *doors_open*. I tilegg slipper man å sjekke noe mer enn om det eksister en ordre i en gitt etasje i bevegelsestransisjonene.

Plasseringen av definisjonen til struckten Queue hadde kanskje mest intuitivt sett vært i queue.h. Ettersom Elevator inneholder en Queue, må elevator vite hvor mye plass en Queue tar i minne når Elevator initialiseres. Hvis Queue er definert i queue.h og den inkluderes i elevator.h, ender vi opp med sirkulær inkludering, ettersom queue.h også trenger å inkudere elevator.h. Foroverdeklarering vil heller ikke løse problemet ettersom Elevator-structen derefereres i mange funksjoner i queue. Et annet alternativ kunne vært å ha en egen headerfil som hadde inneholdt structdeklareringene. Vi ønsket å beholde programmet så enkelt og oversiktlig som mulig og deklarerte i stedet structene i elevator.h.