Heisprosjekt
Gruppe 42

TTK4235 Tilpassede Datasystemer

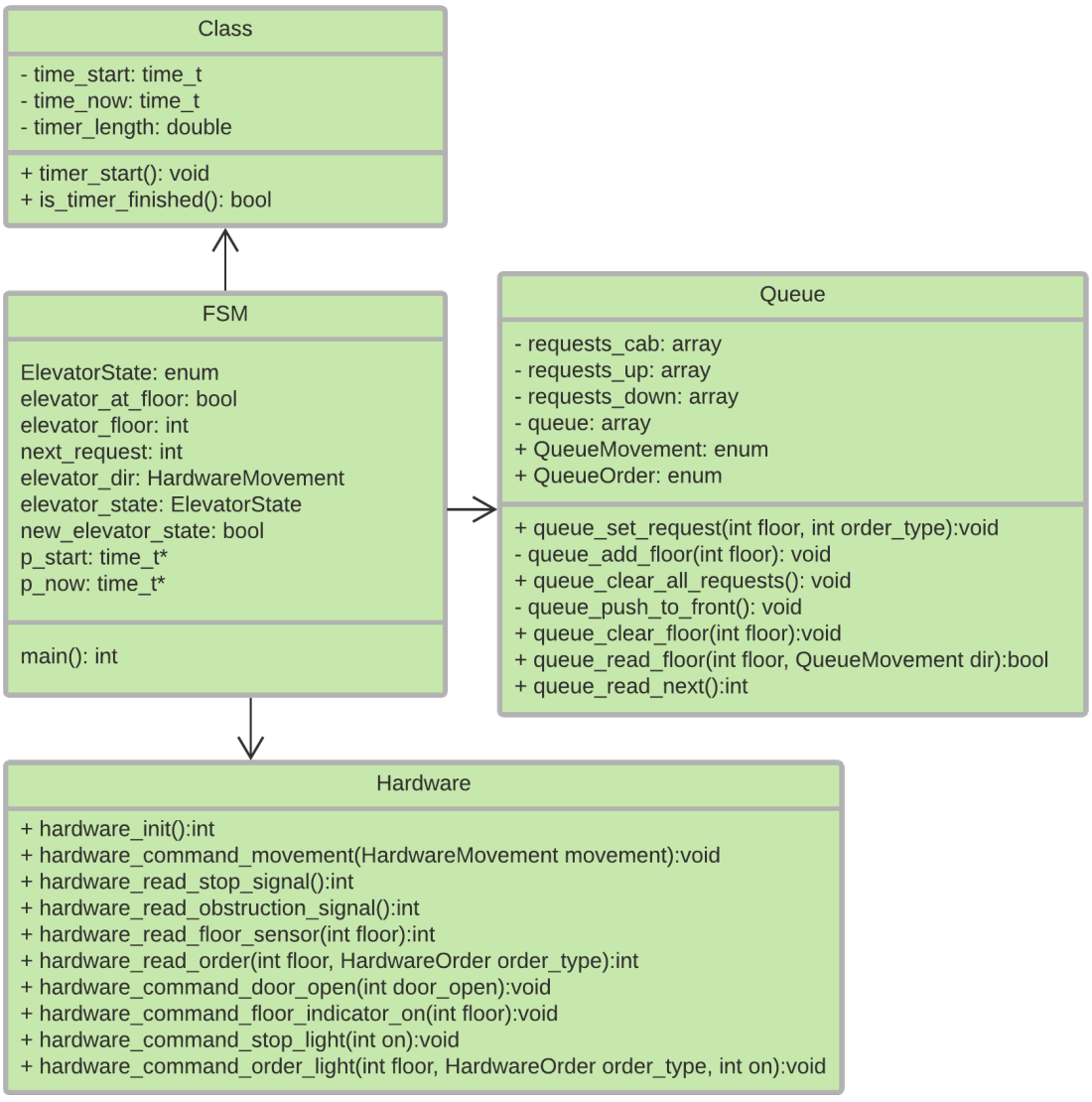
Vår 2020

Skrevet av: Erik R. Furevik og Oliver K. Husby
NTNU: Norges teknisk-naturvitenskapelige universitet
DATO: 01.03.20

Arkitektur

Moduler

Strukturelt er programmet delt opp i 4 moduler: Tilstandsmaskin, Hardware, Queue og Timer, vist i Figur 1. Hardware er uendret fra slik den ble gitt i starten av prosjektet, og brukes til å lese fra eller skrive til heisen. Queue lagrer alle heisens ordre, inkludert rekkefølge, og har en rekke funksjoner til å lese, legge til eller fjerne ordre. Timer brukes til å holde på og bruke nødvendige tidspunkt for å få stoppeklokke-funksjonalitet. FSM er tilstandsmaskinen som tar i bruk alle modulene til å styre heisen. Den har en rekke kontrollvariabler og flagg som brukes i styringslogikken for å sikre korrekt oppførsel.



Figur 1: Klassediagram

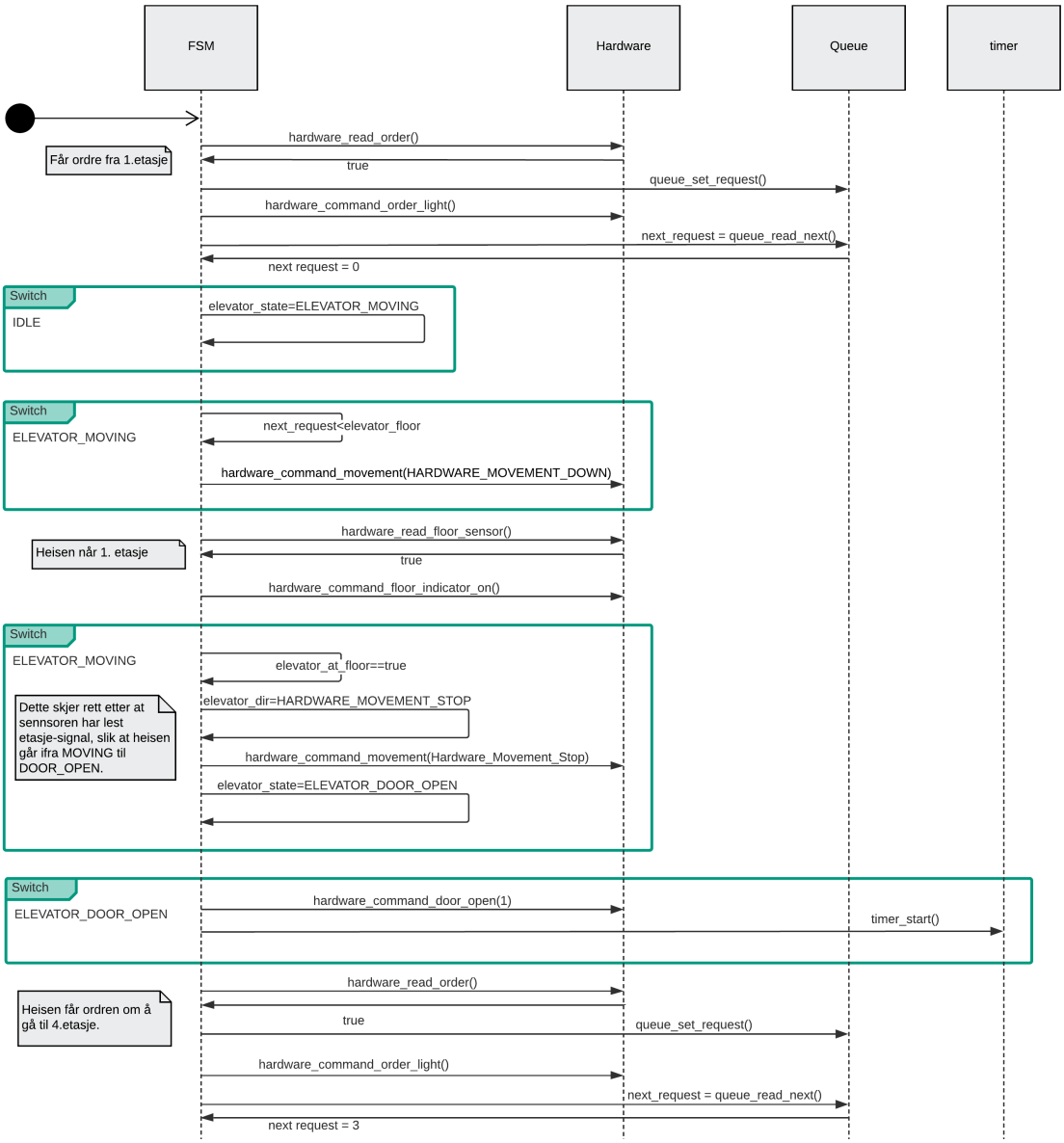
Brukstilfelle
For å vise modulenes samhandling skulle det lages ett sekvensdiagram med følgende use-case:

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.

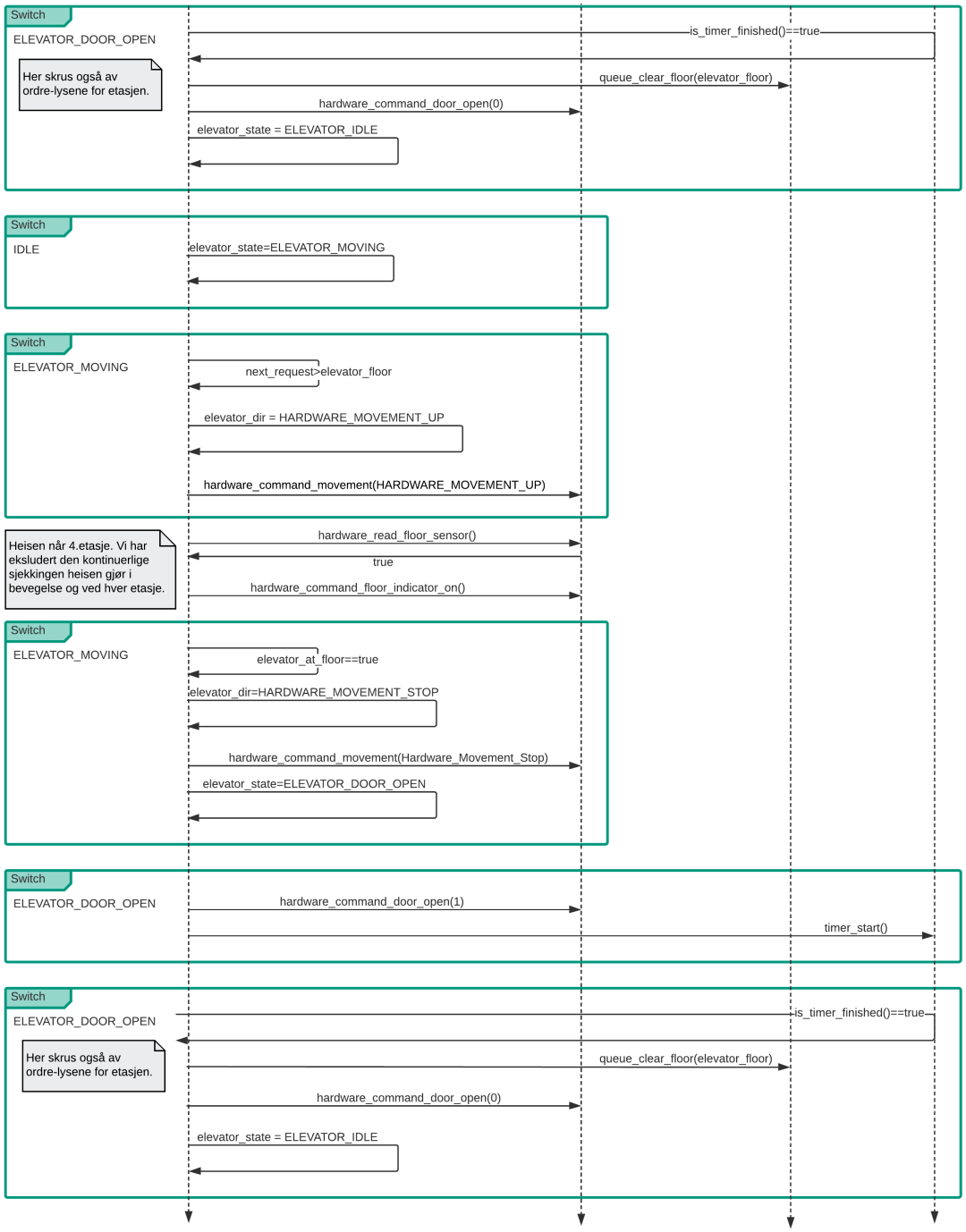
For vårt sekvensdiagram har vi latt være å inkludere en del if-setninger og variabel-settinger som gjøres, da det ikke er relevant for programflyten i dette tilfellet. Fra figur 2 og 3 ser vi at tilstandsmaskinen får alt av informasjon fra de andre modulene, som er ønsket oppførsel. Dette mener vi er en god implementasjon, da det abstraherer modulene fra hverandre, og inndelingen blir slik at hver modul har ett konsentrert fokusområde. Dette resulterer i god konroll over moduelenes oppgaver, og gir god og lesbar programflyt. Vi ser også at det kun er FSM som snakker med de andre modulene, og at ingen av de andre modulene snakker med hverandre. Vi ønsker at ingen av modulene skal være avhengig av hverandre, go de skal kun gi informasjon til tilstandsmaskinen. Dette gjør det lettere og bytte ut enkelte moduler om det skulle være ønsket senere.

I figur 2 og 3 ser vi også at etter at heisen har lukket dørene i en etasje, tilstand eksiterer den til IDLE. Dette er fordi at IDLE fungerer som ett mellomsteg for å sjekke hvilken retning heisen skal gå til basert på ulike variabler og inputs fra de andre modulene.

Noe å være obs på fra sekvensdiagrammet er at den returnerer at neste ordre er 0 og 3 for henholdsvis etasje 1 og 4, noe som kommer av definisjoner i koden.



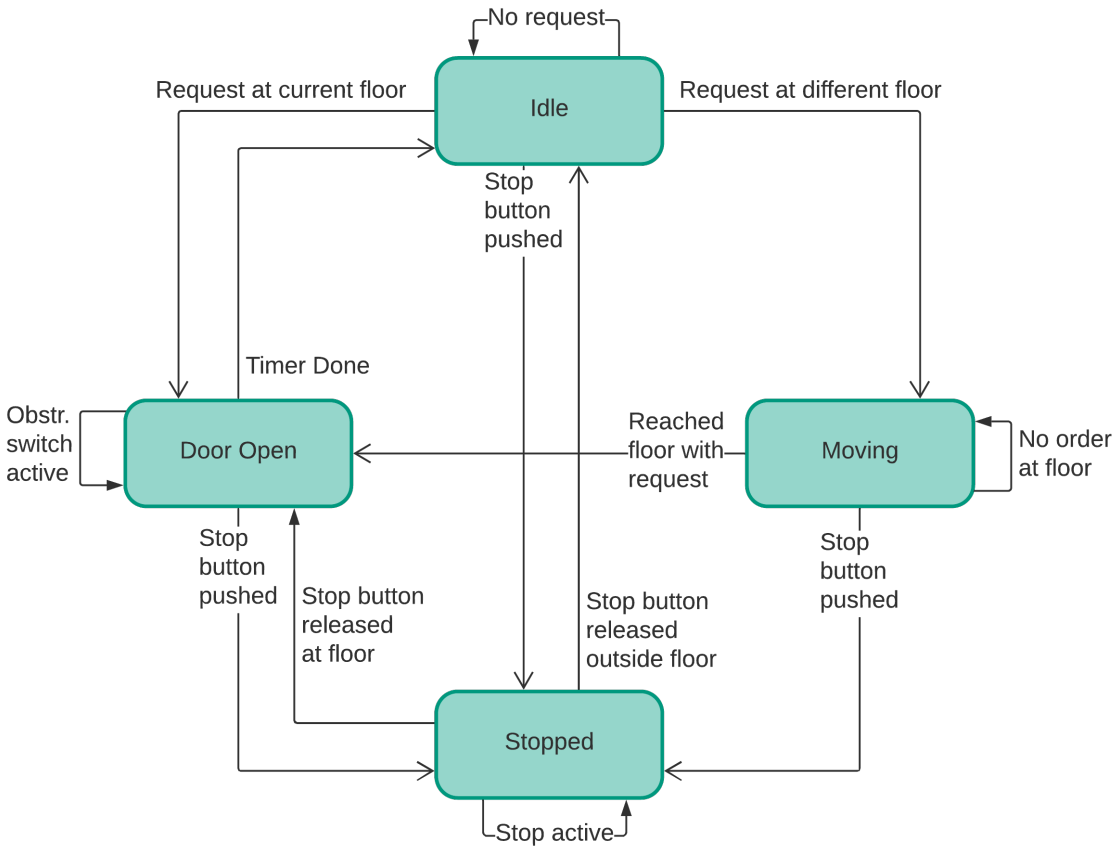
Figur 2: Sekvensdiagram del 1



Figur 3: Sekvensdiagram del 2

Tilstandsmaskin

Fire tilstander brukes til å beskrive heisen, vist i Figur 4. Heisen begynner alltid i tilstanden Idle, og programmet går dit etter hver gjennomførte ordre. Idle finner neste ordre og bestemmer tilstand deretter. Programmet går til tilstanden Moving når den skal til en ordre i en annen etasje. Tilstanden Door Open brukes når heisen når en etasje med en ordre. Den åpner døren i tre sekunder og går tilbake til Idle. Den fjerde tilstanden, Stopped, er aktiv hvis (og bare hvis) stopp-knappen holdes inne, og kan nås fra alle andre tilstander. Den går derfra til Door Open hvis heisen er i en etasje, hvor døren åpnes og lukkes. Ellers går den til Idle og venter på ordre.



Figur 4: Tilstandsdiagram

Moduldesign

Queue

Queue-modulen lagrer ordrene i tre boolske vektorer, og har i tillegg en køvektor. Alle variablene er deklarert som *static* i c-filen, altså de er lokale og kun tilgjengelig i modulen. I ordrevektorene har hver knapp på styringsboksen en tilhørende verdi. Køen har lengde lik

antall etasjer, og holder styr på hvilken ordre, i form av etasjenummer, som er neste. Når en ordre på en ny etasje er registrert blir etasjen lagt til bakerst i køen. Alle ordre for en etasje fjernes fra alle vektorer når heisen stopper der. Gjenværende etasjer i køen blir så skubbet frem. På denne måten er det garantert at ingen ordre blir ignorert. Merk at selv om heisen alltid vil kjøre mot neste etasje i køen, vil den likevel stoppe for å betjene eventuelle ordre på veien før den fortsetter.

Modulen har relativt mange funksjoner tilgjengelig for FSM, som i utgangspunktet bryter med prinsippene om minst mulig kobling. Men hovedpoenget med modulen er i denne arkitekturen ikke å fungere mest mulig selvstendig. Funksjonene er nemlig skreddersydd for bruksområdet. Poenget er snarere å øke oversiktlighet ved å redusere kompleksiteten og lengden på tilstandsmaskinen, og dette gjør den uten tvil. At køvektorene er lokale i sin egen modul begrenser også tilgjengeligheten og reduserer sannsynligheten for feil.

Merk at modulen har ingen funksjonalitet for å håndtere lys, dette gjøres utelukkende i main. Selv om lysene skal tilsvare ordrene trengs det teknisk sett ikke trengs innsyn i ordrene for å sette lysene riktig. De blir nullstillt ved start og deretter satt ved knappetrykk og nullstillt ved stopp i en etasje.

Timer
Timer-modulen er bygget på *time.h* bibliotekets funksjoner for å kunne bygge en enklest mulig implentasjon av en tidtaker. Timer-modulen har to static time_t-variabler, en som lagrer tidspunktet når stoppeklokken starter, en variabel som lagrer nå-tidspunktet for å sjekke om tiden til stoppeklokken har gått ut. Den har også en flyttall variabel som inneholder tiden stoppeklokken skal vare i sekunder.

Modulen fungerer slik at når heisen ankommer en etasje, settes startverdien til det tiden PC'en viser akkurat da. Kontinuerlig mens heisen står stille oppdateres nå-tidspunktet, og differansen mellom start og nå regnes ut. Når forskjellen blir større enn ønsket stoppeklokketid, vil lyset til døren skrus av. Om det skulle komme et obstruksjon- eller stoppsignal underveis mens heisen står i en etasje (med døren åpen i obstruksjontilfelle) vil start-verdien kontinuerlig oppdateres helt til signalet blir satt lavt, slik at start-tidspunktet for stoppeklokken blir det samme tidspunktet signalet går lavt.

Det å lagre alle variabler inne i Timer-modulen gjør selve tilstandsmaskinen mer oversiktlig, da den ikke trenger å huske noen tidspunkter, men bare kaller funksjoner som gjør alt dette. Dette gir ryddig og oversiktlig kode, og en god uavhengihet mellom modulene da stoppeklokke-verdiene ikke går inn og ut av ulike moduler.

FSM
Tilstandsmaskinen er implementert direkte i main-funksjonen. Funksjonen har først en initialiseringsfase. Her blir alle variabler definert, og heisen blir initialisert og ført til nærmeste

etasje i retning ned. Så følger en evig løkke som driver heisen. Først i denne er en for-løkke som går gjennom etasjene og leser all informasjon fra hardware. Ordre blir sendt videre til Queue-modulen mens lys blir satt direkte. Stopped-tilstanden aktiveres herfra, slik at den kan aktiveres uansett tilstand.

Så kommer selve tilstandsmaskinen, implementert som en switch-statement. Hver tilstand gjennomfører en initialisering når den blir aktivert, samt en overgang til neste tilstand når nødvendige krav er møtt. Ett flagg sørger for at hver del kun blir gjort en gang mellom hvert tilstandsbytte. Dette designvalget har gjort koden lett og naturlig å skrive.

Merk at mulige overganger i tilstandsmaskinen er noe begrenset. For eksempel, ved trykk på stopp og deretter en etasje, vil tilstandene skifte gjennom Stopped, Door Open, Idle og så Moving istedet for direkte til Moving. Det kan ved første øyekast virke som en svakhet i maskinen, men begrensningene er i virkeligheten en fordel med det at logikken kan skrives enklere. På denne måten har hver tilstand en enkel og tydelig definert oppgave. Idle tolker neste ordre, Moving beveger og stopper heisen, Door Open åpner og lukker døren, Stopped venter bare til knappen slippes. Flere overganger ville sannsynligvis bare ha gitt høyere kompleksitet, mindre oversikt, og lengre kode grunnet koderepetisjon over flere tilstander.

Testing

Enhetstesting
De to egenproduserte hjelpemodulene Timer og Queue ble utviklet først. De ble laget for å utføre funksjonene som ble sett på som nødvendig fra designfasen, og begge gjennongikk enkel testing. Virkemåten til disse modulene er såpass enkel at en rekke enkle operasjoner og påfølgende utskrift til terminal ble sett på som tilstrekkelig. Modulene ble videre tilpasset underveis i utviklingen av tilstandsmaskinen, men det ble ikke gjort mer enhetstesting.

Integrasjonstesting
Systemet ble først testet uten funksjonalitet for stoppknapp og obstruksjonsbryter, dette fordi heisen da fortsatt kan kjøre, men antall feilkilder blir noe redusert. Når oppførselen var tilfredsstillende ble disse også aktivert og testet. Systemet som helhet ble først testet systematisk, direkte og punkt for punkt etter kravspesifikasjonen gitt i seksjon 2 i oppgaveteksten, og så etter ulike andre fantasifulle og tilfeldige kombinasjoner av omstendigheter. Den siste formen for testing tjener til å teste punkt Y1 som påpeker at heisen skal oppføre seg som forventet.

For å teste for eksempel krav H2 ble det trykket på heispanel 2, 4, og deretter ned 3 på heispanel, og det kan observeres at heisen går fra andre til fjerde etasje uten stopp. Det ble avdekket enkelte feil underveis, noen rimelig åpenbare og enkle, andre litt mer subtile. For eksempel oppdaget vi en feil som var at heisen alltid gikk til 1.etasje ved oppstart. Grunnen til denne feilen ble imidlertidig avdekket kjapt ved bruk av GDB, og fikset rimelig enkelt.

Ellers var ingen feil så store og stygge at de ikke kunne innsnevres rimelig kjapt med litt strategisk utskrift til terminal. At hver tilstand kun kjører koden en enkelt gang ved transisjon gjør det enkelt å oppdage feil, da de fleste feil var på grunn av en tilstand som enten ikke avsluttet eller avsluttet for tidlig.

Diskusjon

En mulig kritikk av arkitekturen er at tilstandsmaskinen er implementert direkte i mainfunksjonen, noe som øker kompleksiteten på denne. En mulig løsning ville vært å separere de to i ulike moduler. Man kunne brukt en slags sirkelarkitektur, der en Reader-modul leser fra Hardware og oppdaterer Queue, mens Tilstandsmaskinen i en egen modul leser fra Queue og styrer Hardware. Dette kunne tydeliggjort informasjonsflyten i programmet. Eller en arkitektur der hver tilstand i tilstandsmaskinen er implementert som en funksjon i en egen States-modul. Dette ville i det minste ha redusert lengden på main. Men selv om alt dette er interessant fra et designperspektiv, er det vanskelig å se det praktiske behovet for eller nytteverdien i slike løsninger. Alle modulene ville ha trengt større sammenkobling, og mye mer informasjon måtte blitt delt. Vi er dermed ikke overbevist om at et alternativt designvalg ville resultert i et betydelig enklere eller bedre program.

I starten hadde vi Timer-modulen implementert ved hjelp av dynamisk allokerte pekere. Pekerne var da lagret i tilstandsmaskin-modulen, noe som vi i etterkant så var ugunstig med tanke på at modulene skal være så uavhengige som mulig. Med implementasjonen som den står nå vil det derfor være mye enklere å endre Timer-modulen uten å måtte endre noe i tilstandsmaskinen, som er ønsket ved modul-implementasjon. Stoppeklokken kunne nok fortsatt hadd blitt løst med dynamisk allokerte pekere i selve Timer-modulen, men vi så ingen direkte og stor fordel med dette ovenfor bare vanlige variabler i dette tilfellet.

Konklusjon

For å konkludere synes vi selv at vår arkitektur er god, enkel og oversiktlig. Den er ikke overkomplisert av for mange moduler, og modulene i seg selv gjør bare det de skal og ikke noe mer. Det oppstod noen få problemer på veien, men ingen store nok til å fasilitere betydelige endringer. Arkitekturen vi hadde fra starten har holdt seg konstant gjennom prosjektet, noe som kommer av at vi brukte mye tid i starten til å designe ulike UML-diagram for å få god oversikt over hvordan vi ønsket at heisen skulle fungere. Vi mener dermed at vi har endt opp med en god arkitektur med gode moduler, og dermed en godt fungerende heis.