Rapport for Heisprosjekt

Torje Nysæther, Hanne Loftesnes

1. Overordnet arkitektur

Heisarkitekturen vår er basert på en heislogikk der heisen prøver å gå så langt mulig i hver retning før den endrer retning. Dette er kanskje den enkleste måten å designe en heis på, da den blant annet ikke trenger å ta hensyn til hvilken rekkefølge bestillingene ankom i, men bare sjekker om den har flere bestillinger i riktig retning. Dette gjør at alle bestillinger uansett blir tatt, men at du kan oppleve at heisen går oppover selv om du går inn i heisen og trykker på en etasje under deg.

Heisarkitekturen er delt inn i fire hovedmoduler, som man kan se i figur 1. Den første, hardware, er tilnærmet lik hardwaremodulen fra den utdelte koden. Modulen er, som navnet tilsier, grensesnittet mellom hardwaren og heisen. Modulen brukes i samtlige andre moduler i systemet. Deretter har vi laget en modul som utvider hardware-funksjonaliteten med litt mer logikk. Den kalles set_hardware, fordi den hovedsakelig blir brukt til å sette hardwaren ved hjelp av de utdelte funksjonene i hardware.h. Køsystemet til heisen er implementert i en egen modul, som tar inn input fra hardwaren, og forteller hva den ønsker å gjøre basert på sine indre tilstander. Til slutt har vi tilstandsmaskinen som bruker de andre modulene og knytter dem sammen med logikken vi ser i figur 3. Samhandlingen mellom modulene våre kan ses i figur 2. Dette diagrammet er noe forenklet, vi har for eksempel ikke tatt med all skrivingen til lys. Det er også noen funksjoner som strengt tatt går via køsystemet, men i dette enkle eksempelet ikke trenger å bruke køen, så derfor kaller tilstandsmaskinen hardware direkte. Dette er for å holde det litt mer oversiktlig.

2. Moduldesign

2.1. Køsystem-design

Kø-modulen er selve hjernen i heissystemet, og det er gjort flere valg av interesse her. Det kanskje aller viktigste er hvordan structen QueueState er bygget opp. Siden køen må kunne lagre bestillinger fra brukere, er det åpenbart at den må kunne lagre tilstander. Vi har valgt å samle disse i et struct, ettersom dette er et oversiktlig alternativ. Kø-designet er basert på tre arrays med fire int-verdier som lagrer inputen fra bestillingsknappene til heisen, der hver knapp har sitt array-element. Ettersom de tre forskjellige knappesystemene fører til forskjellig funksjonalitet, er det naturlig å lagre tilstanden til alle knappene. Lagring av bestillinger har derfor en ganske minimal implementering, ettersom vi minst må lagre ti tilstander. Grunnen til at vi likevel har totalt tolv tilstander i arrayene, og ikke ti, er at det gjør det enklere å indeksere og iterere over arrayene. Det gjør at vi slipper range-baserte sjekker og kommer unna med færre for-løkker, noe som gjør koden mer kompakt, mer leselig og også tryggere, da det introduserer færre muligheter for feilindeksering og out-of-range-problemer. Det å innføre ugyldige variabler er dog selvsagt ikke direkte ønskelig. Likevel, ettersom de aktuelle hardware-funksjonene sjekker for og håndterer ugyldige tilstander, mener vi at variablene ikke utgjør noen stor fare for resten av programmet. Dette gjør at fordelene ved to ugyldige variable er større enn ulempene.

Ellers finnes det fire variabler som lagrer diverse tilstander resten av køen bruker. De to første, saved_floor og current_floor, lagrer den leste verdien av etasjesensorene fra hardware på to forskjellige måter. current_floor er i praksis bare et kall på hardware_read_floor(), og kunne ha vært byttet ut med funksjonskall alle steder den brukes i medlemsfunksjonene. Vi har likevel valgt å bruke en tilstand, ettersom verdien brukes i mange av medlemsfunksjoenen til køen. Dessuten, hvis current_floor bare settes på starten av hver iterasjon kan etasjen ikke endre seg underveis i iterasjonen. Det motsatte kunne ha medført merkelige bugs, en god grunn til å ikke bruke funksjonskall. current_floor får verdien DEFAULT_FLOOR, en macro for -1 definert i harware.h, når heisen er mellom to etasjer. savedfloor lagrer derimot også mellom hvilke etasjer heisen er, der den aktuelle etasjen ganges med to, og mellom-etasjene er en mindre eller større enn de hele etasjene. Dette er nødvendig for å huske hvor heisen er etter alle bestillinger slettes av stoppknappen.

Tilstanden preferred_motor_state er innført fordi køen må kunne fortelle de andre modulene hvilken vei den ønsker at heisen skal kjøre. Samtidig må køen vite hvilken vei den kjører, for å vite om den skal stoppe på en etasje eller ikke, ved H2 i kravspesifikasjonen. Ettersom arkitekturen er laget slik at bare køen kan sette tilstandene opp og ned på motoren, vet køen direkte om den kjører opp eller ned når den kommer til en ny etasje, bare basert på indre tilstander. Dette er i henhold til prinsippet om at modulene skal være så løst koblet som mulig, ettersom kø-modulen bare trenger å inkludere funksjonalitet fra hardware.h med denne implementasjonen, og ikke trenger å vite hva som skjer i de to andre modulene.

Tilstanden destination er viktig for at heisen skal stoppe ved endetilstandene. Ettersom heisen ved H2 i kravspesifikasjonen skal ignorere bestillinger i motsatt retning, vil heisen ikke automatisk ta bestillingene på endene, dersom de settes motsatt vei av motorretningen. Dette skjer blant annet alltid for knappene utenfor heisen i endeetasjene. Heisen må, slik vi har designet den, alltid stoppet på den siste bestillingen på vei opp eller ned, og må derfor lokalisere hvilken etasje dette er. Dette gjøres ved destination, som blir satt som den høyeste etasjen som har en bestilling hvis ønsket motorretning er opp, eller laveste i det motsatte tilfellet. Tilstanden settes til DEFAULT_FLOOR hvis det ikke er flere bestillinger.

Resten av modulen er funksjoner som alle tar inn en QueueState-peker som eneste parameter. Hele modulen kan derfor sees på som tilnærmet objektorientert, ettersom alle funksjoner enten manipulerer eller henter ut data fra en kø-objektet. C støtter ikke objektorientert programmering direkte, men vi har likevel valgt å implementere det, ettersom problemstillingen legger opp til større mengder tilstander som må manipuleres og hentes ut, noe som passer dette paradigmet bra. Vi mener medlemsfunksjonene har beskrivende navn og lesbar kodeimplementasjon, og trenger derfor ikke enkeltvis en formell forklaring her. Vi vil likevel gi noen forklaringer på enkelte valg.

I queue_remove_current_floor() skal man fjerne bestillingene på den aktuelle etasjen man er på. Slik tilstandsmaskinen er implementert blir denne funksjonen funksjonen bare kalt når heisen stopper i en etasje. Det vil si at tilstanden current_floor aldri vil være DEFAULT_FLOOR. Vi har likevel valgt å legge inn en sjekk på om etasjen faktisk er gyldig. Grunnen er gjenbruk og vedlikehold, ettersom funksjonen kan føre til range-errors og potensielle segfaults dersom funksjonen brukes andre steder i koden uten denne sjekken. Tiltaket minker derfor sekvensiell kohesjonen, og øker kodekvaliteten.

I queue_get_user_input() har vi valgt å legge inn en betingelse om at køen bare ta inn bestillinger dersom man ikke er på samme etasje som bestillingen kommer på, eller at køen ikke har bestillinger. At køen ikke har bestillinger er representert ved at preferred_motor_state er lik HARDWARE_MOVEMENT_STOP. Dette er logikk for et spesialtilfelle som kravspesifikasjonen ikke har spesifisert. I praksis betyr dette at køen ikke skal ta inn nye bestillinger på samme etasje dersom heisen har en retning. Vi mener dette er en gyldig tolkning av kravspesifikasjonen av flere grunner. For det første spesifiserer kravspesifikasjonen at døren skal åpnes i 3 sekund når "heisen ankommer en etasje det er gjort en bestilling til", altså ikke hver gang det kommer en ny bestilling i den samme etasjen. Samtidig sier H3 at når heisen først stopper i en etasje, skal alle bestillinger i etasjen bli ekspedert. Det står ikke eksplisitt at det bare gjelder orderene som er der når den kommer, så vi mener at tolkningen derfor er gyldig. Vi gjør denne antagelsen så lenge hallelementet leser etasje, ettersom det er ingenting i kravspesifikasjonen som taler direkte imot dette. Dette fører til at i dette tilfellet er det å ekspedere en bestilling i samme etasje praktisk talt det samme som å ikke ta registrere bestillingen, ettersom timeren ikke trenger å settes på nytt. Vi bryter derfor heller ikke med H1, ettersom alle bestillinger i praksis blir tatt. Vi mener derfor at denne implementasjonen ikke bryter med kravspesifikasjonen.

Dette skaper konsekvenser for funksjonen queue_check_if_stop_
Denne må blant annet sjekke om motoren er riktig vei i forhold til bestillingen eller om ønsket retning til køen er stopp. Den siste sjekken er nødvendig, ettersom den gjør at døren kan åpne seg på bestilling fra utsiden, selv om heisen står uten bestillinger i samme etasje med lukket dør. Det ville vært dårlig heisoppførsel hvis man ikke kommer seg inn i en heis uten bestillinger, så dette ville ha brutt med Y1.

2.2. Set_hardware-design

Denne modulen inneholder funkjoner som kalles av tilstandsmaskinen for å sette de fysiske tilstandene til hardwaren. Ettersom de fleste funksjonene, utenom noe dørlogikk, setter hardware direkte, har modulen som nevnt fått dette navnet. De fleste funksjonene er korte og selvforklarende, men vi skal gå litt inn på samspillet mellom dem.

Et av de viktigste poengene fra denne modulen, er hvordan timeren til døren overstyrer alt annet i modulen. Timeren er implementert svært enkelt, ved at den sammenlignes med systemklokken i sanntid etter den er satt. Videre er denne knyttet svært tett med både dør og motor, ved at åpning av dør settes timeren, som holder døren åpen så lenge timeren er aktiv, samtidig som motoren settes til stopp så lenge døren er åpen. En enkel logikk for timeren skaper derfor en enkel logikk for døren, og av den grunn også motor. Dette er et gjennomtenkt valg, ettersom det gjør at flyten i modulen blir strømlinjeformet med få avhengigheter. Det fører til at kodeflyten blir enklere å følge, som da kan hindre feil og gjøre det lettere å gjenbruke kode ved vedlikehold.

I modulen har vi også prøvd å begrense lesing fra hardware til et minimum, for å senke avhengigheten mellom de to modulene og heller la køen ta seg av lesingen. Vi blir likevel ikke kvitt lesing av etasje i oppstartsfasen, stopp og obstruksjon, ettersom disse er nødvendige og unaturlige å ha i kø-modulen. Vi har også prøvd å begrense grensesnittet mellom set_hardware og kømodulen så mye som mulig. Dette er gjort ved at bare den foretrukne motortilstanden, preferred_motor_state, brukes direkte for logikk i set_hardware innenfor supertilstanden i tilstandsdiagrammet 3. Dette gjør at de modulene er ganske løst knyttet, som betyr at det er enkelt å endre på kølogikken uten store endringer i set_hardware-modulen.

2.3. FSM-design

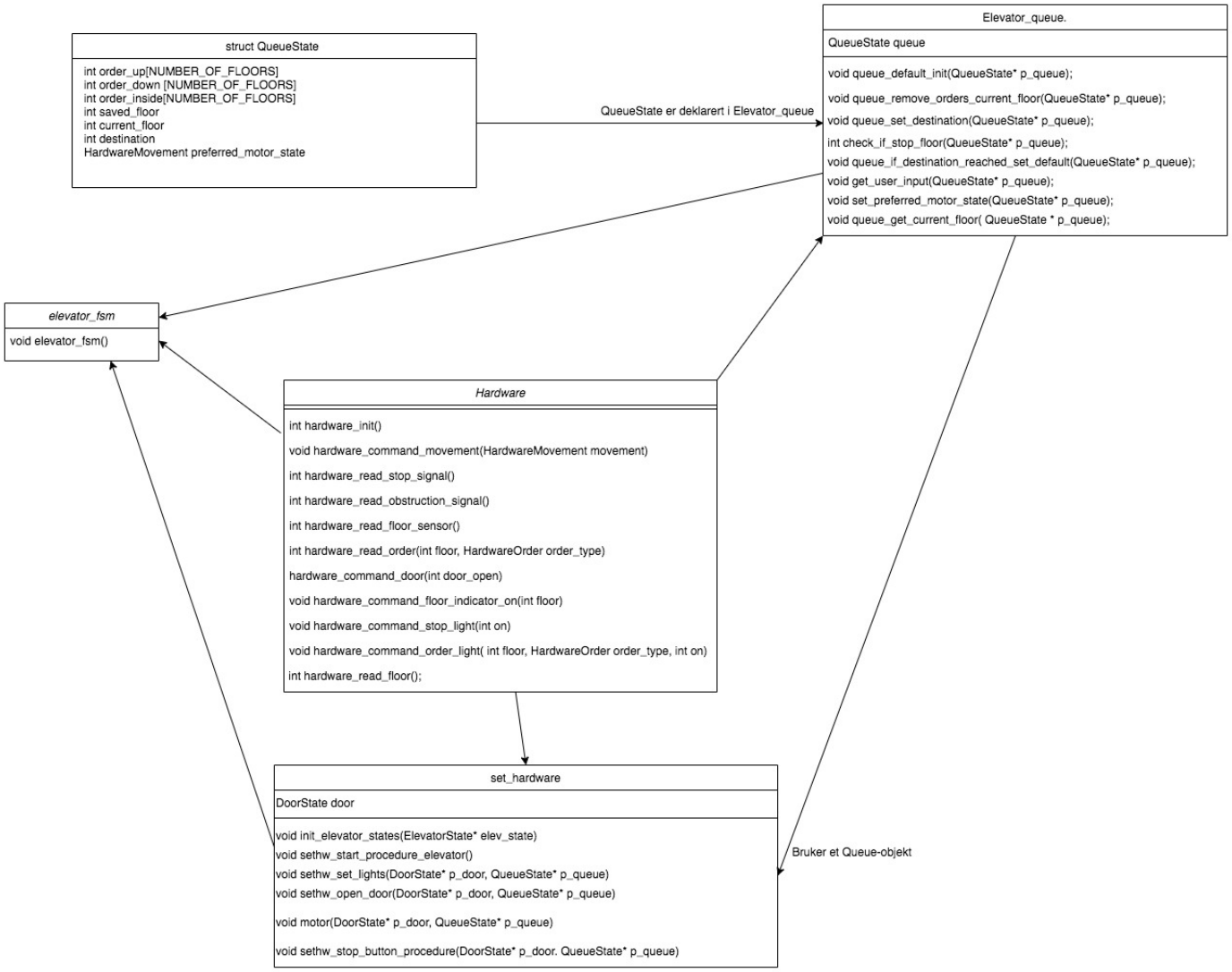
Hele heissystemet styres som nevnt av en tilstandsmaskin, som tar inn de andre modulene i systemet. I denne modulen opprettes de ulike tilstandsobjektene i systemet, køen, døren og timer for døren. Tilstandene manipuleres i en evig løkke, basert på input fra hardware, via modulene. Mye av logikken i tilstandsmaskinen er abstrahert bort i funksjoner, og tilstandsdiagrammet, figur 3, er derfor en god beskrivelse over hvordan systemet er bygget opp.

Alle variabler blir initialisert etter opprettelse, ettersom det er god praksis og reduserer mulighet for feil. Av interesse kan man se at det er brukt to funksjoner for å initialisere køen, ettersom dette var funksjonalitet som fantes i andre funksjoner fra før, samtidig som funksjonene er grunnleggende forskjellige da queue_default_init() initialiserer likt hver gang, mens queue_get_current_floor() settes basert på hardware-måling. Ettersom queue_get_current_floor() kalles i hver iterasjon i while-løkken er det strengt tatt unødvendig å kalle den først i tillegg, men det gjøres for å tydeliggjøre at hele objektet initialiseres ved start.

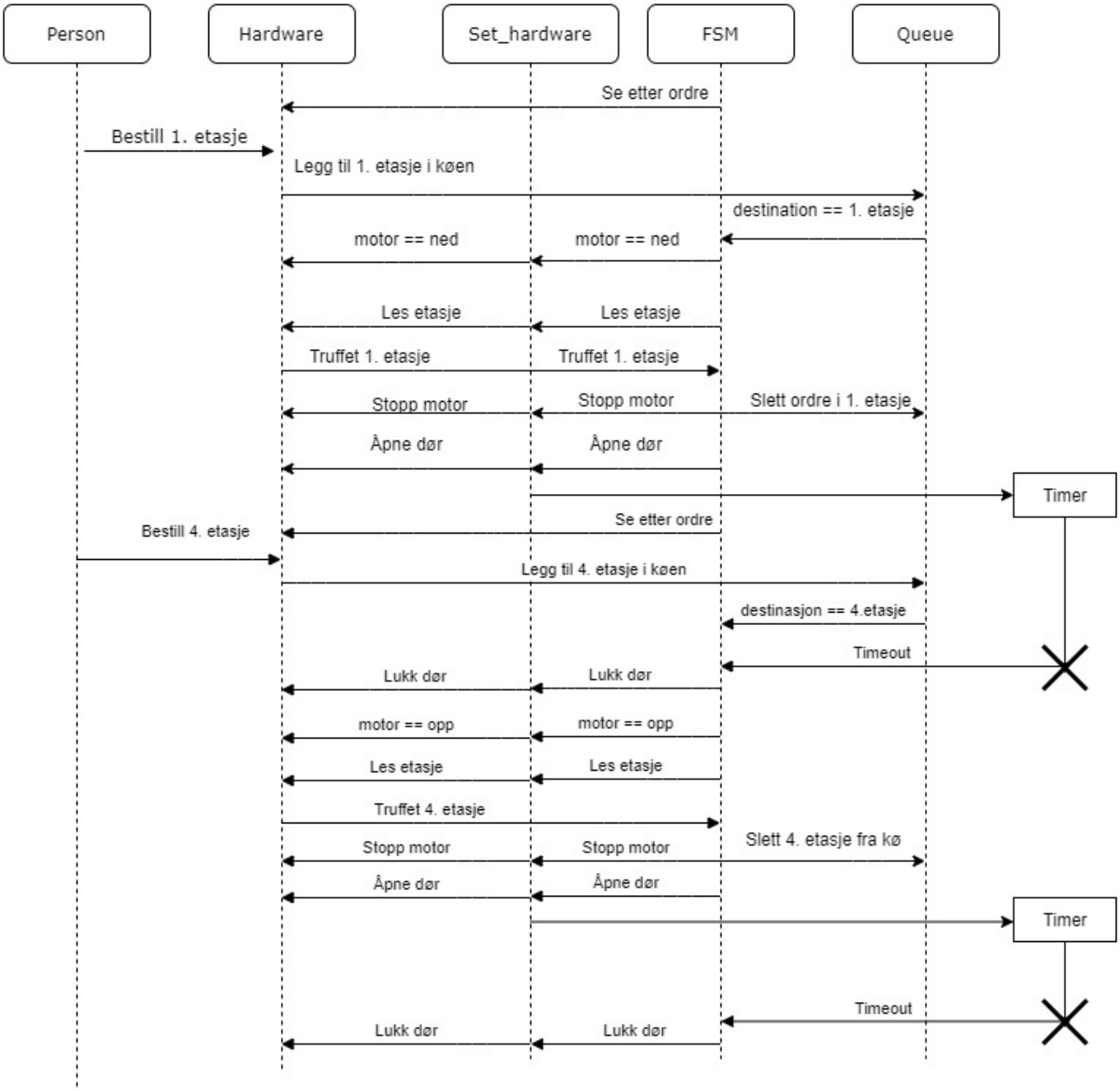
Stoppfunksjonaliteten er løst ved en if/else-setning, basert på input fra stoppknappen. Betingelsen kjører enten stopprosedyren, eller den vanlige prosedyren, som tar inn input, og kjører motoren basert på systemets tilstander. Dette stemmer godt overens med tilstandsdiagrammet, der stopp-tilstanden er utenfor supertilstanden. Stoppknappen hindrer derfor at den vanlige prosedyren blir kalt, og er derfor en sikker måte å hindre vanlig funksjonalitet når den er trykket inne, gitt av S4 og S5. Funksjonskallene sethw_try_close_door() og sethw_lights() er plassert utenfor for denne stopp-betingelsen. Dette er fordi begge funksjonene både leser stoppknappen og de andre tilstander i systemet.

2.4. Hardware-design

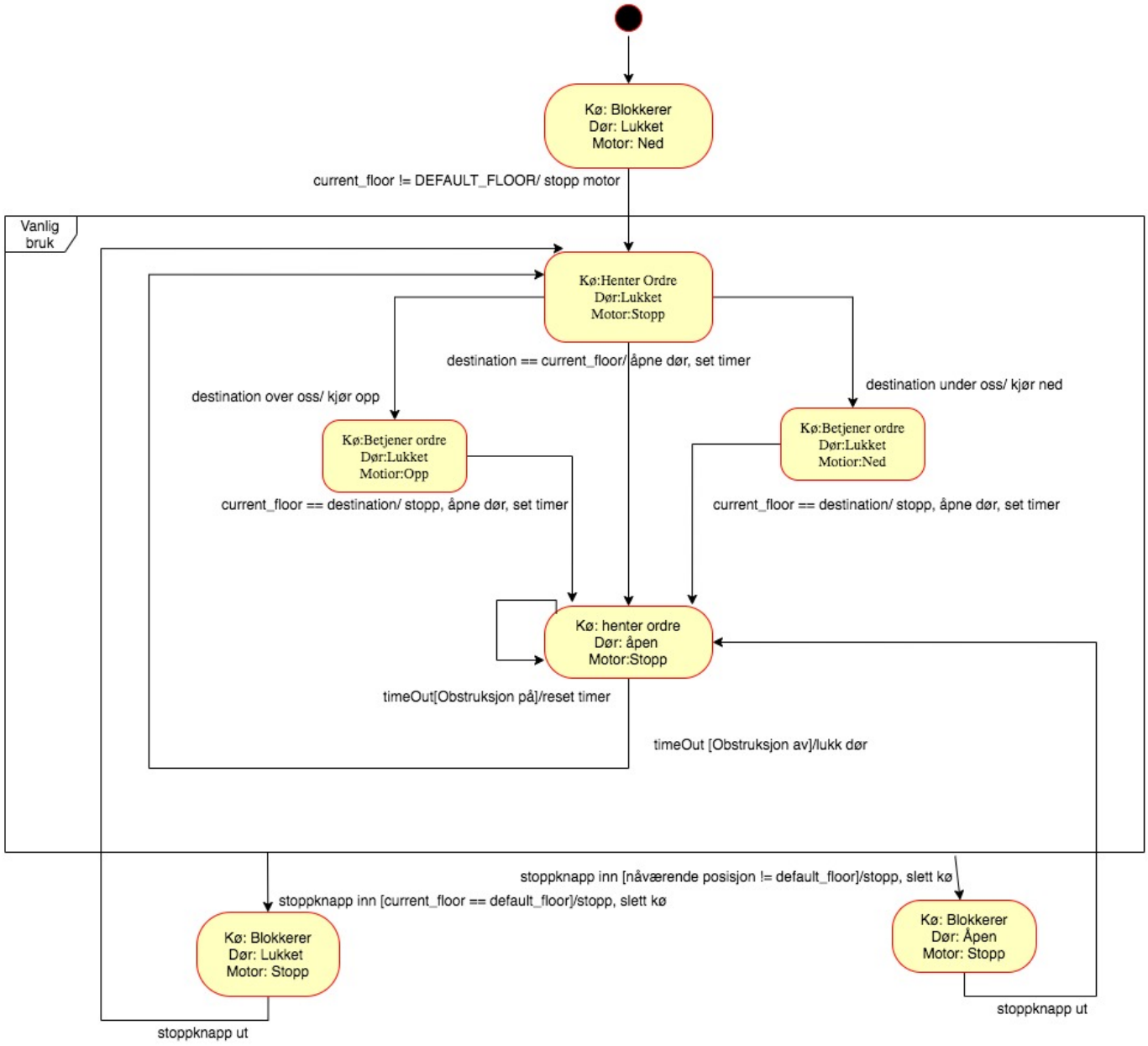
Det er også gjort en liten endring i hardware-filen, da vi har lagt til en funksjon her, hardware_read_floor() og definert en macro, DEFAULT_FLOOR. Denne er definert her ettersom den brukes både i queue og set_hardware. Ettersom den ikke påvirket køen direkte, passet den ikke inn i kømodulen. Den hørte heller ikke hjemme i set_hardware, ettersom queue ikke inkluderer set_hardware, av grunner argumentert for i kø-modulen. Ettersom funksjonen overordnet bare er en logisk utvidelse av hardware_read_floor_sensor(), er den derfor plassert i hardware. Den definerte macroen følger av at den brukes i denne funksjonen. Vi har laget denne macroen for å øke lesbarhet til tilstanden "ingen etasje".



Figur 1: Klassediagram



Figur 2: Sekvensdiagram



Figur 3: Tilstandsdiagram

3. Testing

3.1. Enhetstesting

For å teste at heisen oppfylte kravene begynte vi først med å teste de ulike enhetene. vi begynte med køsystemet, som er viktigst.Dette ble gjort litt underveis, da vi skulle debugge køen for å finne feil, og til slutt, da alt endelig virket som det skulle.
Køsystemet ble testet i GDB ved å legge til og fjerne bestillinger og se at det oppførte seg som forventet. Eksempler på forventet oppførsel vil da være at bestillinger ble lagt til på riktig plass i køen, destination ble satt til riktig etasje, køsystemet visste til enhver tid hvor heisen var og at bestillinger ble slettet riktig når en bestilling ble tatt eller når stoppknappen ble trykket inn.
Set_hardware ble også testet ganske tidlig, men litt senere, slik at vi kunne bruke køsystemet til å teste det. Først skrudde vi bare på systemet for å se at oppstartssekvensen så riktig ut (O1). Deretter ble alle lys tilhørende knapper testet ved å trykke dem inn og ut. (L1, L6). Etasjelysene ble testet ved å kjøre/dra heisen gjennom alle etasjene og se at det riktige lyset lyste (L3, L4, L5). Den første testen av dørlyset innebar egentlig bare å se at den tente seg på riktig tidspunkt og slukket etter tre sekunder (D1). Til slutt ble det litt enkel testing av stopp-sekvensen, hovedsakelig for å se om motoren stoppet når knappen ble trykket inn (S4).

3.2. Intergrasjonstesting

Testing av FSM-modulen vil tilsvare det samme som integrasjonstesting, siden FSM-en er modulen som knytter de andre modulene sammen. Integrasjonstest gjøres ved å sjekke at den oppfyller kravspesifikasjonen, så derfor gikk vi gjennom hele kravspesifikasjonen for å teste alt på nytt. Dette er da bare de testene som allerede ikke er gjengitt av enhetstestene, siden de dekker en god del av kravspesifikasjonen fra før.
Vi startet med å sette den i gang og lot den kjøre ned, samtidig som vi prøvde å gjøre bestillinger. Disse ble da ikke lagt til. Heisen stoppet i nærmeste etasje og ble der(O2).
Etter dette la vi på mange forskjellige bestillinger for å sjekke flere ting. Først, at alle bestillingene vi satte på ble betjent (H1), deretter at heisen ikke tok f.eks. nedadgående bestilling i 3. etasje når den gikk fra 1. etasje til 4. etasje (H2). Deretter, at alle bestillingene i en etasje ble betjent. Dette kunne vi se at de riktige lysene ble slukket og at heisen ikke dro tilbake til den samme etasjen (H3). Og til sist testet vi at den ble stående stille når den ikke hadde noen bestilling. Dette skjedde da både etter oppstart og da den var ferdig med alle bestillingene vi ga den. (H4). Da dette skjedde var også døra lukket (D3).
Etter dette fulgte litt grundigere testing av stopp-knappen. Vi trykket den inn i tide og utide for å se at den stoppet og åpnet døra når den skulle. Vi sjekket også at den ikke la inn nye bestillinger så lenge stopp var trykket inn. (S4,S5,S6,S7). Til slutt la vi til mange bestillinger og satte obstruksjonen høy i tide og utide for å se at den ikke påvirket systemet annet enn når døra var åpen (R1).
Noen av kravene var litt vanskelig å teste eksplisitt (O3, H1, R1 og R2), men vi mener det skal være kodet på en slik måte at alle disse kravene er oppfylt. Vi har dessuten aldri sett noe som tyder på at dette ikke skal være oppfylt. Til sist har vi konkludert med at etter grundig testing ser heisen ut til å fungere bra, som en vanlig heis, og kanskje litt bedre enn den i gamle elektro (Y1).

4. Diskusjon

Generelt så er vi fornøyde med både arkitekturen og implementasjonen til heisen. Hver modul har definerte, logiske ansvarsområder og vi mener at modulene er løst nok knyttet, blant annet ved at h-filer inkluderer hverandre, og at det er mulig å for eksempel endre køsystemet fundamentalt uten for store endringer i set_hardware-modulen. Arkitekturen og implementasjonen er robust nok til å ikke måtte ta hensyn til en mengde spesialtilfeller, og de som finnes er stort sett argumentert for i kø-seksjonen i rapporten. Vi har brukt en del tid å på å fjerne redundant kode, og kildekoden vår er derfor vesentlig mindre omfattende enn den kunne ha vært. Dette gjør den enklere å lese, og derfor også enklere å vedlikeholde. En objektorientert implementasjon gjør det også enklere å følge når og hvordan tilstandene i heisen blir påvirket, da den samme rekken med funksjoner blir kjørt hver gang så lenge man forblir inne i supertilstanden til tilstandsdiagrammet. Fordelen med dette er at det blant annet reduserer spagettikode"ved funksjoner som kaller hverandre på en uoversiktlig måte.

Å implementere objektorientering i C kan likevel være noe problematisk. Blant annet er det mulig å kritisere bruken av et relativt stort struct i kømodulen. Denne tas inn som referanse i mange funksjoner i koden, også utenfor sitt navnerom. C har ikke innebygde metoder for sikkerhet i objektorientert programmering som for eksempel c++, blant annet ved at man ikke kan gjøre medlemsvariabler private, og styre hvilke funksjoner som kan endre variablene i klassen. Dette kan føre til feil som kan være vanskelige å finne, dersom objektet blir endret ved en feil på obskure steder i koden. Dette kunne man ha sikret seg mot i større grad ved å lage get-funksjoner til klassen slik at bare medlemsfunksjoner kan hente ut tilstandene. Blant annet for preferred_motor_state kunne dette vært aktuelt i dette tilfellet, ettersom denne tilstanden hentes av en funksjon i set_hardware.c. Vi har likevel ikke valgt å gjøre det, ettersom det kan sees på som unødvendig når alle medlemsvariable er offentlige uansett.

Implementasjonens kanskje største svakhet er likevel at den er preget av noe sekvensiell kohesjon i tilstandsmaskinmodulen. Dette er fordi når medlemsfunksjoner endrer på kø-objektet, kan rekkefølgen på funksjonskallene ha noe å si, og det kan oppstå merkelig oppførsel. Et godt eksempel på dette er i linje 32-33 i kildefilen til elevator_fsm, der det å kalle de to funksjonene i motsatt rekkefølge ville ført til helt gal oppførsel. I dette tilfellet er det fordi sletting

av bestillinger før destinasjonen settes kan føre til at feil destinasjon blir satt. Å endre på dette ville fort ha ført til vesentlig endring av implementasjonenn, og vi måtte muligens ha delvis forlatt hele objektorientert-paradigmet. Det kunne nok også vært løst med flere betingelser i koden, men dette vil føre til styggere og mer komplisert kode. Et eksempel på at vi likvel har gjort dette er i funksjonen queue_remove_current_floor(), som beskrevet i modulseksjoenen.

Vi har altså laget et heisdesign, som er enkelt, både konseptuelt og implementasjonsmessig, men som holder kravspesifikasjonen med god margin. Vi har vist flere eksempel på at vi har tatt valg med tanke på å skrive vedlikeholdbar kode i modulene, og bundet modulene sammen på en tilfredstillende måte, med lav grad av avhengighet i forhold til det uunngåelige minimumet. Koden er delvis objektorientert, ettersom det en naturlig måte å strukturere kode på i et slikt problem. Dette paradigmet fører likevel med seg flere problemer, som at C ikke eksplisitt er designet for dette, og at det innfører sekvensiell kohesjon i implementasjonen. Dette er dog mindre potensielle sikkerhetsutfordringer ved endring av kode og ikke et stort direkte problem for implementasjonen.