

# 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 ordre 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 `setHardware`, 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.

Modulene er delt opp slik de er for å finne hardware-logikken litt hver for seg, så mye som det gikk an i vår implementasjon, for så å knytte det sammen i tilstandsmaskinen.

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 ordre 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 ordre 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 selvsagt ikke direkte ønskelig. Ettersom de aktuelle hardware-funksjonene sjekker for og håndterer ugyldige tilstander, så mener vi likevel 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, `savedFloor` og `currentFloor`, lagrer den leste verdien av etasjesensorene fra hardware på to forskjellige måter. `currentFloor` er i praksis bare et kall på `hardwareReadFloor()`, 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 de andre medlemsfunksjoenen til køen, som da bare trenger å settes en gang per Det gjør også at `currentFloor` ikke kan endre seg i løpet av en while-iterasjon når den er satt, som kunne ha medført merkelige bugs. `currentFloor` får verdien `DEFAULT_FLOOR`, en macro for -1 definert i `hardware.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 etasjene. Dette er nødvendig for å huske hvor heisen er etter alle ordre slettes av stoppknappen.

Tilstanden `preferredMotorState` er innført fordi køen må kunne fortelle de andre modulene hvilken vei den ønsker at heisen skal kjøre. Samtidig er det andre elementer som overstyrer køen med tanke på motorretning, derav navnet `preferred`. 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 tilstandsmaskinen 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 tilstan-

der. Dette er i henhold til prinsippet om at modulene skal være så løst koblet som mulig, da blant annet dette gjør at kø-modulen bare trenger å inkludere funksjonalitet fra hardware.h.

Tilstanden `destination` er viktig for at heisen skal stoppe ved endetilstandene. Ettersom heisen i gitte tilfeller ved H2 i kravspesifikasjonen skal ignorere bestillinger, vil den ikke automatisk ta ordrene på enden, 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 i den siste ordren på vei opp eller ned, og må derfor lokalisere etasjen denne finnes i. Dette gjøres ved `destination`, som blir satt som den høyeste etasjen som har en ordre hvis ønsket motorretning er opp, laveste i det motsatte tilfellet og `DEFAULT_FLOOR` det ikke er flere ordre.

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 manipulerer eller henter ut data fra en struct, som kan sees på som en klasse. C støtter ikke direkte objektorientert programmering, men vi har valgt å implementere det likevel, ettersom problemstillingen legger opp til større mengder tilstander som må manipuleres og hentes ut, noe som passer dette paradigmet bra. Funksjonene i modulen er også de eneste som kan endre klassen direkte, som er typisk objektorientert programmering. 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 bare denne funksjonen kalt når vi har stoppet på en etasje. Det vil si at tilstanden `current_floor` aldri vil være `DEFAULT_FLOOR`, eller -1, som er den faktiske verdien til macroen. 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 den brukes andre plasser i koden uten denne sjekken. Tiltaket minker derfor sekvensiell kohesjonen i koden, og øker derfor kodekvaliteten.

I `queue_get_user_input` har vi valgt å legge inn en betingelse. Køen skal bare ta inn bestillinger dersom man ikke er på samme etasje som bestillingen kommer på, eller at køen ikke har ordre, representert ved at `preferred_motor_state` er stopp. Dette er logikk for et spesialtilfelle som kravspesifikasjonen ikke har spesifisert. I praksis betyr dette at den ikke tar inn nye ordrer på samme etasje dersom heisen har en annen destinasjon, at den skal en plass etterpå i samme retning. Vi mener dette er en gyldig tolkning ettersom kravspesifikasjonen spesifiserer 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 ordre i etasjen bli ekspedert. Det står ikke eksplisitt at det bare gjelder orderene som er der når den kommer, så vi mener at tolkningen er gyldig. Et alternativ ville vært at man kunne låse heisen til etasjen i evig tid, dersom inputen stadig ressatte dørtimer til 3 sekunder, og ville derfor bryte med H1, ettersom da kan heisen låses til en etasje. Det ville heller ikke gitt mening å kunne legge til bestillinger for en heis i samme etasje som blir tatt senere, så det å ikke ekspedere dem umiddelbart ville bryte med Y1. Vi gjør denne antagelsen så lenge hallelementet leser etasje, ettersom det er ingenting i kravspesifikasjonen som taler direkte imot denne antagelsen. Ettersom det å ekspedere en bestilling samtidig som den kommer i dette tilfellet er det samme som å ikke ta inn bestillingen i praksis, mener vi derfor at denne implementasjonen ikke bryter med kravspesifikasjonen.

Av dette så fører det også at funksjonen `queue_check_if_stop_floor` i linje 41 og 42, sjekker om enten motoren er riktig vei i forhold til bestillingen, eller om det ikke er noen aktive bestillinger, altså at ønsket retning er stopp. Den siste sjekken er nødvendig, ettersom den gjør at døren kan åpne seg på bestilling utenifra dersom heisen sto uten bestillinger i samme etasje. Dette ville vært dårlig heisoppførsel hvis man ikke kommer seg inn i heisen når den var uten ordre.

## 2.2. 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 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` plassert utenfor for denne stopp-betingelsen. Dette er fordi begge funksjonene både leser stoppknappen for å sette andre tilstander.

### 2.3. *Set\_hardware-design*

Denne modulen inneholder funksjoner 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. Funksjonene i modulen tar inn forskjellige tilstander, blant annet output fra kø-modulen, dørtilstanden og timeren. De fysiske tilstandene blir så satt med hensyn på disse, med funksjonskall fra hardware-modulen. Flere av funksjonene manipulerer i tillegg dør og timer-tilstandene. 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 settes bare av `open_door()` og i `try_close_door()` av stop- og obstruction-knappene, som spesifisert i kravspesifikasjonen. Samtidig så overstyrer døren køens ønsker om motorretning i `sethw_motor`. Timeren er implementert så enkelt som mulig, ved at det legges til antall klokkesykler tilsvarende 3 sekunder på det nåværende antall klokkesykler hver gang den settes. Den satte timertiden sammenlignes så i hver while-iterasjon med det nåværende antallet klokkesykler i `try_close_door()`, og lukker døra hvis den nåværende tiden er større enn tiden til timeren. Kort fortalt setter altså åpning av dør timeren, som holder døren åpen så lenge timeren er aktiv, som setter motor til stopp så lenge døren er åpen. En enkel logikk for timeren skaper derfor en enkel logikk for dør, og av den grunn også motor. Dette er et gjennomtenkt valg, ettersom det gjør at flyten i modulen blir strømlinjeformet med få avhengigheter. Disse få avhengighetene gjør at koden blir enklere å følge, som da kan hindre feil og gjøre det lettere å gjenbruke kode ved vedlikehold. Her har vi også fokusert på enkel kode, for at disse avhengighetene skal komme tydelig fram.

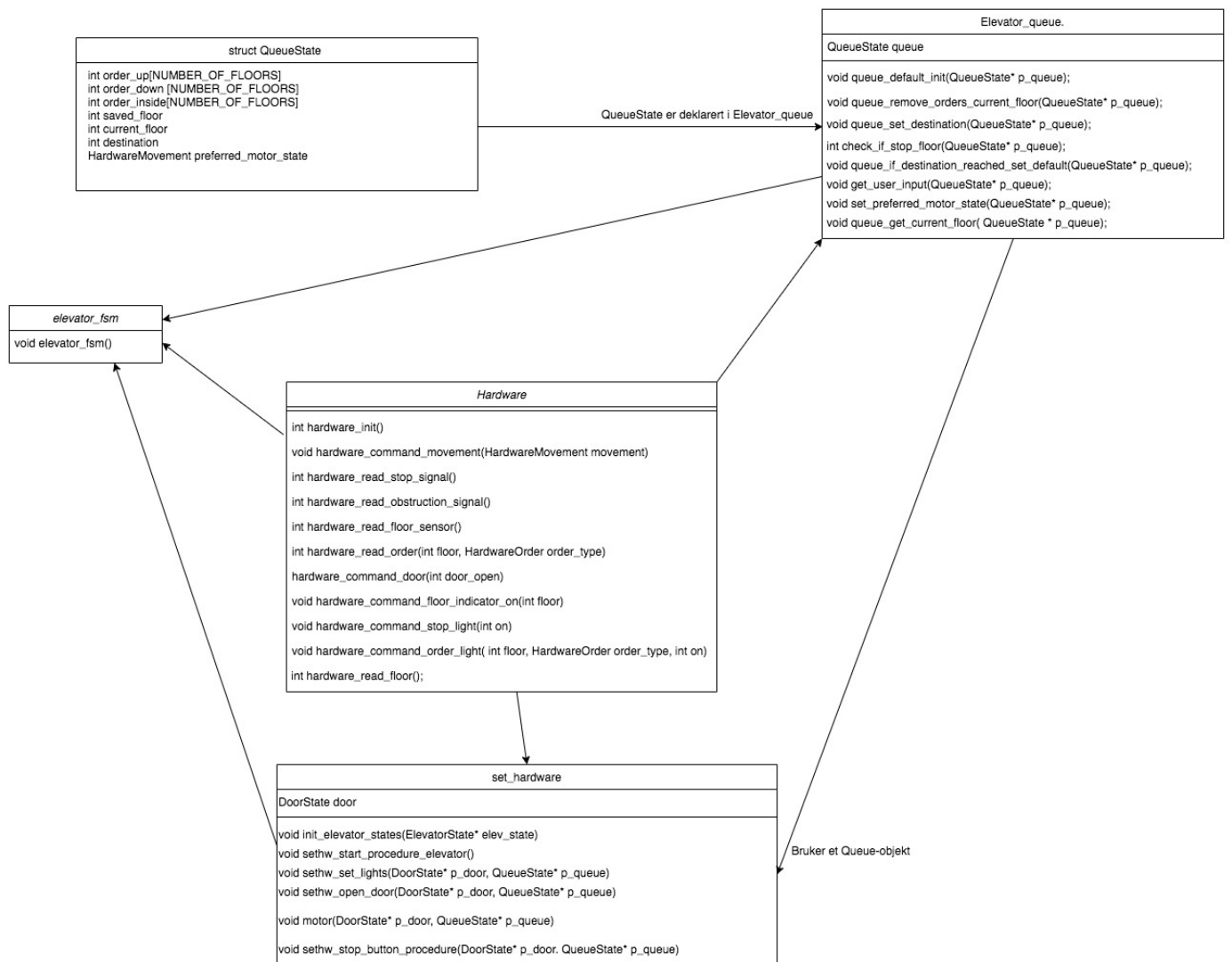
I modulen har vi også prøvd å begrense lesing fra hardware til et minimum, for å senke avhengigheten mellom de to modulene, og heller la køen ta seg av lesingen. Vi blir likevel ikke kvitt lesing av etasje i oppstartsfasen, stopp og obstruksjon, ettersom disse er nødvendige og unaturlige å ha i kø-modulen. 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. Dette gjør at modulene er ganske løst knyttet, som betyr at det er enkelt å endre på kølogikken uten store endringer i `set_hardware`-modulen.

I oppstartsfunksjonen har vi noe tilfeldig valgt å sette motortilstanden til ned til den finner en definert etasje. Denne kunne like fint vært opp, ettersom funksjonen starter

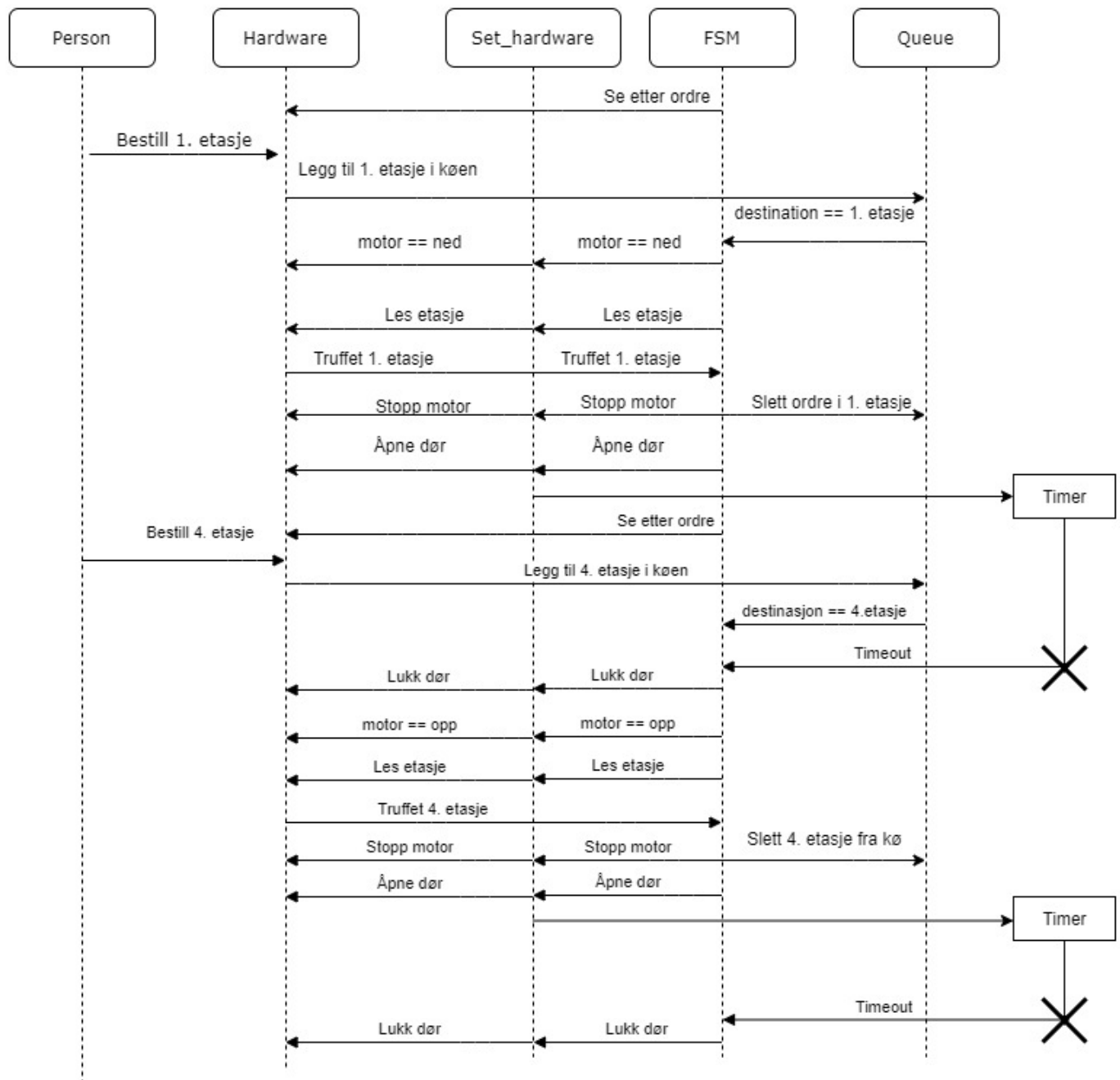
innenfor heisrommet uansett, men må være en av delene for å komme til en definert tilstand.

### 2.4. *Hardware-design*

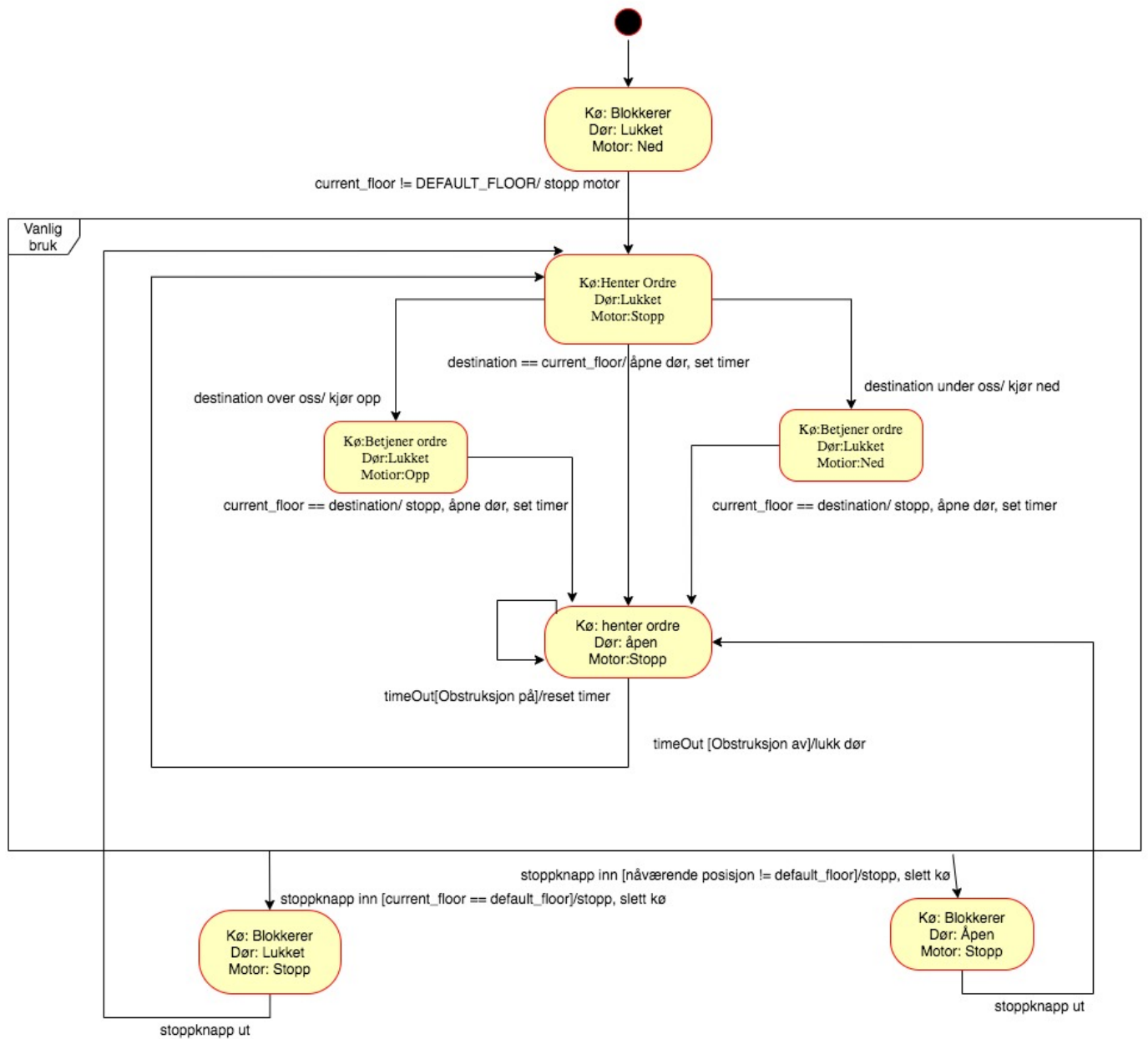
Det er også gjort en liten endring i hardware-filen, da vi har lagt til en funksjon `hardware_read_floor()` og definert en macro, `DEFAULT_FLOOR`. Denne er definert her ettersom den brukes både i queue og `set_hardware`. Ettersom den ikke påvirket køen direkte, passet den ikke inn i kømodulen. Den hørte 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()`, passet den derfor inn 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:** Klassesdiagram



**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 ordre ble lagt til på riktig plass i køen, destination ble satt til riktig etasje, køsystemet visste til enhver tid hvor heisen var og at ordre 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 oppstartsssekvensen 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 oppfylder 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 ordre for å sjekke flere ting. Først, at alle ordrene vi satte på ble betjent (H1), deretter at heisen ikke tok f.eks. nedadgående ordre i 3. etasje når den gikk fra 1. etasje til 4. etasje (H2). Deretter, at alle ordrene 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 ordre. Dette skjedde da både etter oppstart og da den var ferdig med alle ordrene 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 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 ingen 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 modulseksjonen av rapporten. Vi har brukt en del tid 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.

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. Dersom de ulike variablene i køen blir endret på i feil rekkefølge, kan dette føre til merkelig oppførsel det kan være vanskelig å finne. 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 ordre før

destinasjonen settes kan føre til at feil destinasjon blir satt. Å endre på dette ville fort ha ført til vesentlig endring av implementasjonen, og vi måtte muligens ha delvis forlatt hele objektorientert-paradigmet. Det kunne nok også vært løst med mange flere betingelser i koden, men dette vil føre til styggere og mer komplisert kode. Et eksempel på at vi har gjort dette er likevel

Det hadde også vært mulig å gjøre deler av koden enda mer kompakt. Vi har som nevnt valgt å abstrahere bort mye av funksjonaliteten til tilstandsmaskinen til de øvrige kø-modulene, ettersom dette gir et bedre overblikk over hvilke overordnede operasjoner tilstandsmaskinen gjør. Denne abstraksjonen er dog ikke fulstendig, da det for eksempel ikke er laget wrappere for å samle funksjonene, med hensikten færre funksjonskall i tilstandsmaskinen. Blant annet linje 36-40 i `elevator_fsm.c` er bare medlemsfunksjoner til køen, og kunne vært samlet i et funksjonskall og ikke fire. Vi har likevel valgt å ikke gjøre dette, da det sannsynligvis ikke ville ført til mer lesbar kode. I tillegg er ikke det å lage mindre generelle funksjoner og funksjoner som gjør mange ting på en gang et poeng i seg selv. Koden kan likvel kritiseres for å ikke være helt konsekvent med hvilken funksjonalitet som er abstrahert bort og ikke i tilstandsmaskinen. Dette er valg vi har måttet ta for å finne en balansegang mellom å legge så mye implementasjon som mulig i sine respektive moduler, samtidig som det skal være mulig å følge funksjonaliteten i `elevator_fsm.c`.

De fleste valgene i denne modulen er som vi kan se basert på å finne de enkleste mulige løsningene, både basert på funksjonalitet og kodekvalitet, ettersom slik implementasjon rett og slett er bedre.

Vi ser blant annet en ny god grunn til at kø-structen er bygget opp slik som den er, da ordrelsene kan settes direkte fra de tre ordre-arrayene i `QueueState`.