Eirik Mathias Silnes og **Linn Elise Klingenberg** gruppe 6

Heislabb 1

I dette prosjektet har vi fått utdelt firmwarekode (fastvare) og basert på dette programmert en heis som har normal heisoppførsel. Sett fra et pedagogisk perspektiv er heis enkel å dele opp i ulike moduler og abstraksjonsnivåer og på den måten har prosjektet vært en lærerik prosess.

UML og V-modellen

UML-diagram som klassediagram, tilstandsdiagram og sekvensdiagram har vært nyttig, særlig i startfasen. Designprosessen inviterte til en diskusjon rundt en felles forståelse av hvordan heisen skulle implementeres. Uten design av UML-diagram hadde vi risikert at vi begynte å jobbe på hver våre løsninger og på den måten hadde samarbeid blitt en utfordring fremfor en ressurs. Ved å bruke UML ble det derimot tydelig hvordan vi så for oss strukturen til programmet samt hvilke grunnleggende funksjoner vi trengte. Dette var også til hjelp for å komme i gang med programmeringen.

Underveis mens vi skrev koden fant vi ut at vi hadde tenkt litt upraktisk på noen områder og ble derfor nødt til å endre på diagrammene for at de skulle fungere. På dette stadiet hadde vi ikke tenkt på diagrammene på en god stund, slik at dette opplevdes som mer ekstrajobb enn nyttig. På denne tiden ga det med andre ord mening å diskutere med hverandre på bakgrunn av koden og ikke diagrammene.

Avslutningsvis var det interessant å se forskjellen mellom hvordan vi hadde tenkt at programmet skulle se ut og hvordan vi faktisk endte opp med å løse det hele. Under hele feilsøkingsprosessen var den imidlertid ikke til hjelp da vi hadde bedre oversikt over selve koden enn diagrammene.

Bruken av V-modellen har ikke vært hovedfokus under implementeringen av dette. Prosessen med å først designe arkitekturen, deretter modulene og så begynne kodingen har falt naturlig inn. Likevel har vi gått tilbake til arkitekturdesignet og endret dette underveis i programmeringen. I tillegg har enhetstestingen vært vanskelig å gjennomføre og dermed uteblitt. I stedet har vi gått direkte til integrasjonstesting for så å finne feil og endre koden og enkelte ganger også arkitekturen. På den måten har vi fulgt en mer agil struktur enn V-modellen.

Totalt sett har bruk av UML gitt god felles forståelse av problemløsningen og gjort det enkelt å både komme i gang, samt jobbe på ulike deler av programmet hver for oss parallelt uten å komme i konflikt med hverandres arbeid. Den tidligere amerikanske presidenten Dwight D. Eisenhower oppsummerte det slik: “plans are useless, but planning is indispensable” [1]

[1] Oxford Reference https://www.oxfordreference.com/display/10.1093/acref/9780191826719.001.0001/q-oro-ed4-00004005 (funnet 12.03.24)

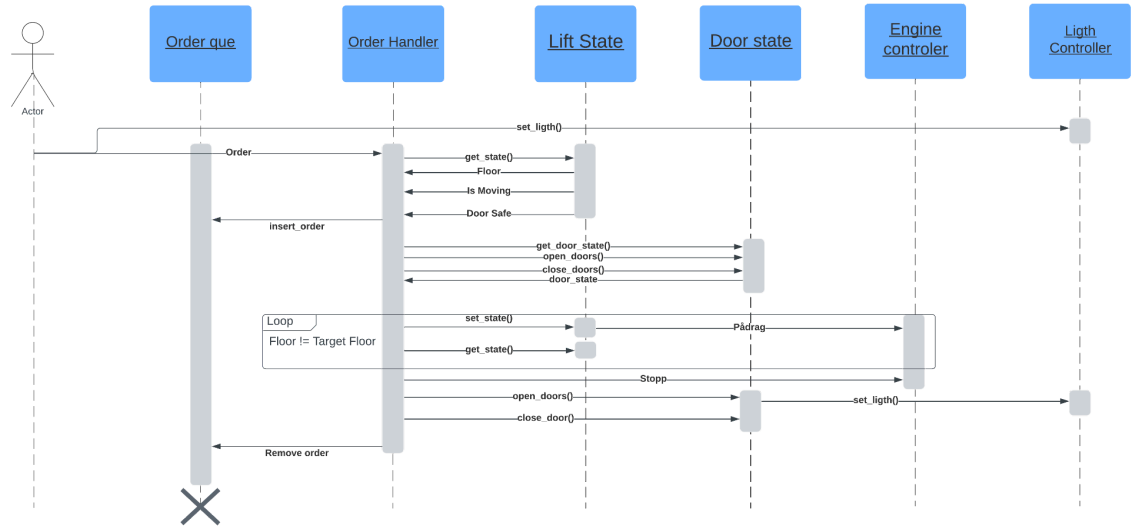
V-modellen har derimot vært fint å lære om, men fungerer dårlig i praksis siden man alltid løser en større oppgave litt feil første gang og det er vanskelig å få frem kompleksiteten en kodespråk tilbyr ved å designe UML-diagrammer.

For å snu det på hodet: hvordan hadde prosjektet vært uten bruk av UML-diagrammer og V-modellen? Uten UML hadde samarbeid, og det at vi kunne jobbe hver for oss blitt vanskelig. Likevel hadde vi nok klart oss uten klassediagram. Dette brukte vi ikke underveis i programmeringen og var heller ikke noe som ga noen bedre oversikt over programflyt eller regler for programmet. Det UML-diagrammet vi brukte mest var sekvensdiagrammet da dette ga en større oversikt over programflyten som var nyttig for å dele inn programmet hensiktsmessige deler. V-modellen hadde vi nok imidlertid klart oss uten og hadde heller blitt en utfordring dersom vi hadde fulgt denne mer slavisk enn vi gjorde.

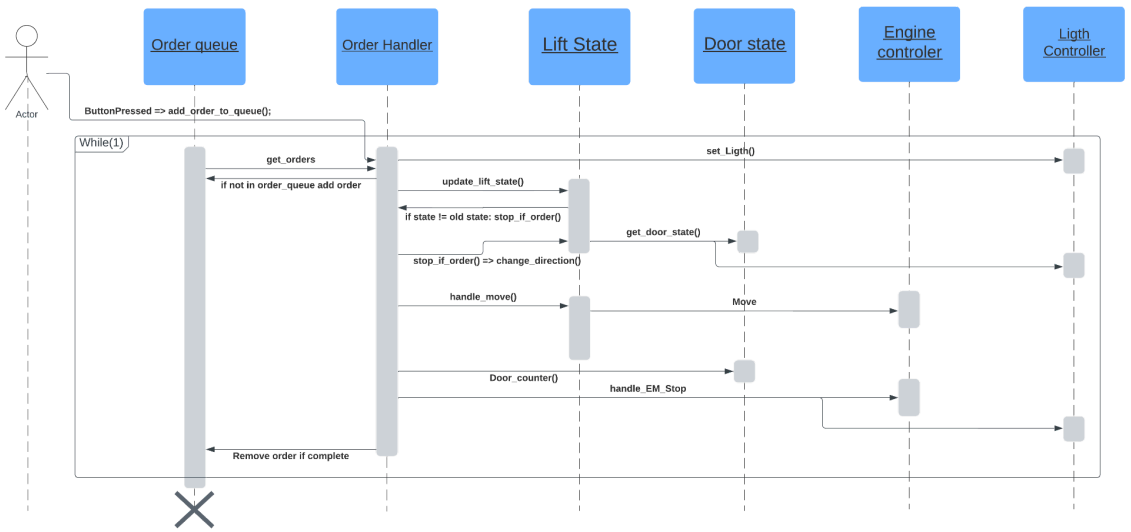
Arkitektur

Arkitekturen har underveis i kode skrivingen endret seg kraftig ettersom den økte forståelsen av systemet i sin helhet ga mer elegante løsninger på problemer som ikke burde vært tilstede ved begynnelsen. Kodemessig ble det bestemt tidlig at main-funksjonen skulle inneholde kun høynivå funksjonskall som gjorde det tydelig hvordan programflyten i programmet var uten for mye dokumentasjon. Etter det bestemte vi oss for at en order_handler skulle være den mest komplekse biten som skulle håndtere ordre og skulle interagere med enklere subsystem som hadde feilhåndtering. Et av eksemplene her er lift_state filen med funksjonen change_direction(direction_t direction) som sørger for at man ikke kan sette heisen i bevegelse når døren er åpen.

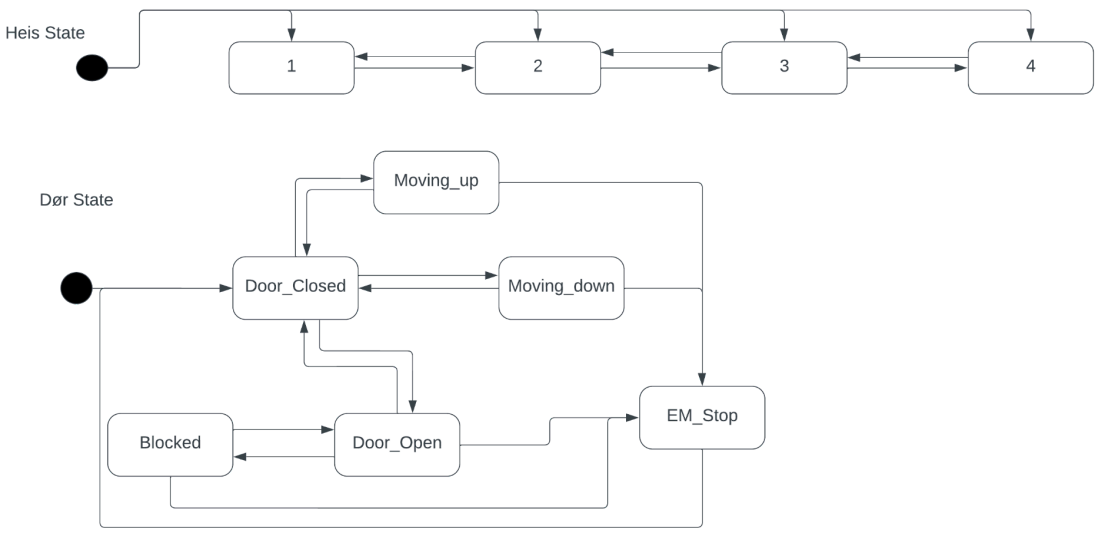
Sekvensdiagram V1:
Ved første gjennomgang så ville vi ha at alle funksjonskall kom ut i fra order_handler og kontrollerte alt fra motor til de to forskjellige tilstandsmaskinene. Det var også et ønske om at alt av besluttninger skulle tas i order_handler og at man da hentet mye info opp til order_handler og tok beslutninger der. Det var da svært få parametere i alle funksjonskall og skulle man endre noe så måtte man bruke get og set funksjoner.



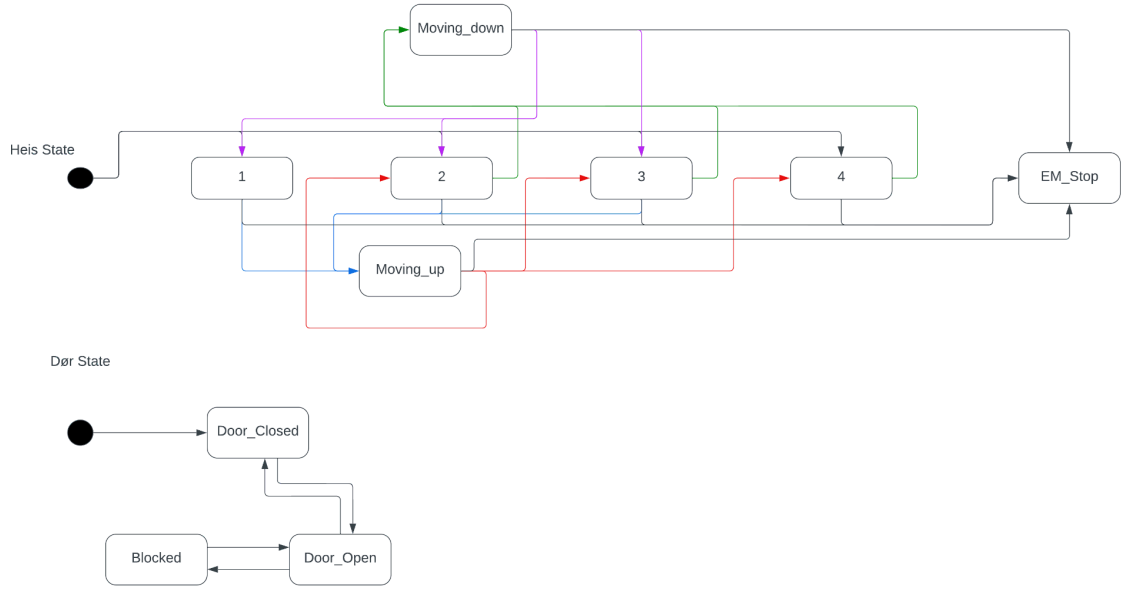
Sekvensdiagram V2:
I versjon to av diagrammet så ble det mer kommunikasjon mellom enkeltmoduler for å ha en mer effektiv programflyt. Den ble nødvendigvis ikke enklere å forstå ved første øyekast, men informasjonsflyten gikk bedre. En typisk endring her var at vi gikk fra å hente både door_state og lift_state opp til order handler for så å sjekke om kombinasjonene av lift_state og door_state var en gyldig kombinasjon. Til at lift_state sjekket door_state og returnerte sant dersom kombinasjonen var gyldig og gjorde ingenting dersom kombinasjonen var ugyldig. Dette gjorde høynivåkoden mye mer oversiktelig ettersom mange av feilhåndteringene var gjemt i den underliggende koden og man kunne fokusere mer på den “avanserte” programflyten.



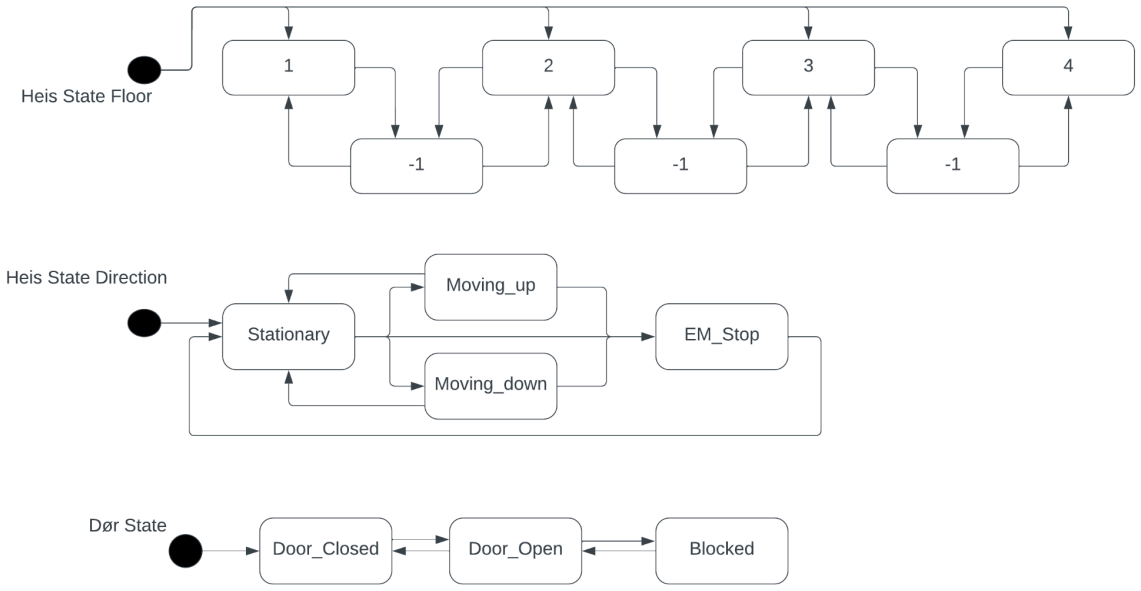
Tilstandsdiagram V1:
Ved første øyekast så virket det som det var en god idé å ha bevegelsene og døren sine tilstander sammen for det ville jo sørge for at man ikke kunne begynne å bevege seg før døren var lukket, så man aldri ville kunne kjørt med døren åpnet. Dette viste seg fort å være upraktisk når man skulle begynne å programmere ettersom bevegelses håndteringen og tilstanden for bevegelse ble separert. Utifra det første tilstandsdiagrammet så er det også tydelig å se at vi ikke hadde fått med oss de mulige returverdiene fra elevio sensoren. Det å få -1 kastet inn i tilstandsmaskinen under viste seg fort å ikke fungere og vi hoppet over til diagram to.



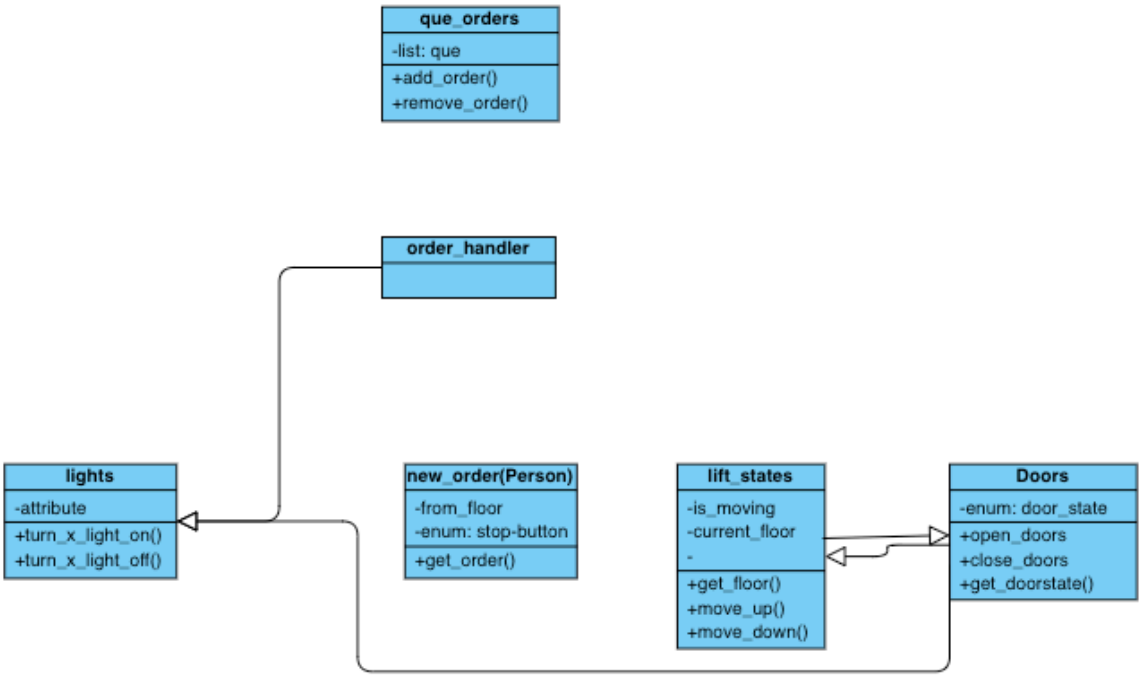
Tilstandsdiagram V2:
Her ble det tydeligere når man befant seg mellom etasjer ved hjelp av Moving_up og Moving_down, og det var mulig å implementere at når sensoren leste -1 så måtte vi bevege oss enten oppover eller nedover. Det å ha EM_Stop som egen tilstand i bevegelsen gjorde det enkelt å aktivere ekstra funksjonalitet når tilstandsmaskinen traff den.



Tilstandsdiagram V3:
Vi endte opp med å splitte opp tilstandene i tre ting lagret i to forskellige structs hvor litft_state structen inneholdt heis Floor og Direction i tillegg til den forrige verdien til de tilstandene. Dette gjorde det veldig enkelt å kunne sette Driection tilstanden til EM_Stop og i tillegg vite at du var mellom etasjer, hva forrige etasje var og at vi var på vei nedover/oppover. Dette gjorde oppstartslogikken etter EM_Stop relativt enkel.



Dette er klassediagrammet. Det ble ikke brukt underveis i programvareutviklingen.



Robusthet, skalerbarhet og vedlikehold

Klare retningslinjer for kommunikasjon mellom moduler er essensielt for robusthet, skalerbarhet og vedlikehold. Dette har design av UML-diagrammer bidratt til. Her er det særlig sekvensdiagrammet som bidratt mest. Bruk av V-modellen derimot ville bidratt til å svekke robustheten siden den krever at man tenker en perfekt løsning første gang.

For å lett kunne få nye utviklere til å raskt forstå programflyt, og for å gjøre vedlikehold og bug-hunting lett, så ble det tatt noen valg om hvordan koden skulle skrives. Main-funksjonen skulle være det eneste stedet i koden hvor det lå en while() løkke, dette for å forhindre at koden ble fastlåst et sted og at man lett skulle kunne sette seg inn i programflyten.

```
20
21      while(1){
22          check_new_order();
23          if(lift_state_update()){
24              stop_if_order();
25          }
26
27          handle_movment();
28
29          door_counter();
30
31          if(elevio_obstruction()){
32              elevio_stopLamp(1);
33          } else {
34              elevio_stopLamp(0);
35          }
36
37          if(elevio_stopButton()){
38              handle_EM_stop();
39          }
40
41          nanosleep(&(struct timespec){0, 20*1000*1000}, NULL);
42      }
43
```

Koden ble deretter segmentert på fysiske handlinger som gjorde det lettere når man skulle lete etter feil. Typisk hvis heisen ikke stoppet så var det noe galt i stop_if_order(), mens hvis heisen bevegde seg rart så skjedde det noe i hadle_movment(). Dette gjorde det relativt enkelt å fikse udefinert og tilfeldig oppførsel og skape struktur i programmet.

Bruk av KI

Vi har ikke brukt noe KI-generert kode i programmet vårt og heller ikke i rapporten. Den eneste gangen vi åpnet ChatGPT var for å finne funksjonalitet i time-biblioteket. Dette uten hell og vi brukte heller stack overflow for inspirasjon til løsning.

“Jeg liker det best på penn og papir” -Linn Elise 2024

Kilder:

Kilde 1: Oxford Essential Quotations (4 ed.), Dwight D. Eisenhower 1890–1969 American Republican statesman, 34th President 1953–61: https://www.oxfordreference.com/display/10.1093/acref/9780191826719.001.0001/q-oro-ed4-00004005