Heisprosjekt TTK4235

Gruppe 33

Elisabeth Hamre og Anna Gravir

I heisprosjektet har vi fått utdelt et sett med krav for hvordan en heis skal fungere, for deretter å skulle implementere kravene i C-kode for å få en fungerende heis. Utviklingen av prosjektet har i hovedsak bestått av to deler. Først lagde vi grove UML-diagrammer for å skaffe en oversikt over virkemåten til heisen. Deretter implementerte vi disse i C-kode og testet underveis. Prosessen har vært dynamisk da vi har måttet endre på UML-diagrammene underveis ettersom abstraksjonsnivået har blitt lavere og flere av kravene har blitt tatt hensyn til.

UML;

Innledende fase kan deles inn i to underfaser, arkitekturdesign og moduldesign.

Arkitekturdesign:

I arkitekturdesignfasen holdt vi abstraksjonsnivået høyt, og fant innledningsvis kun ut hvilke moduler vi kom til å trenge. Disse ble strukturert i et klasse-diagram. Vi så at vi kom til å trenge en modul som lagret bestillinger, «order», en modul som satte heisen i bevegelse, «state», en modul for hardware som registrerte knappe-trykk, «elevio» (fra utdelt kode), og vi så at vi kom til å trenge diverse funksjonaliteter for styring av lys og en timer. Timeren ble etter hvert en egen modul, «timer». Under arkitekturdesignfasen brukte vi også tid på å finne ut hvilke av modulene som måtte kunne kommunisere med hverandre.

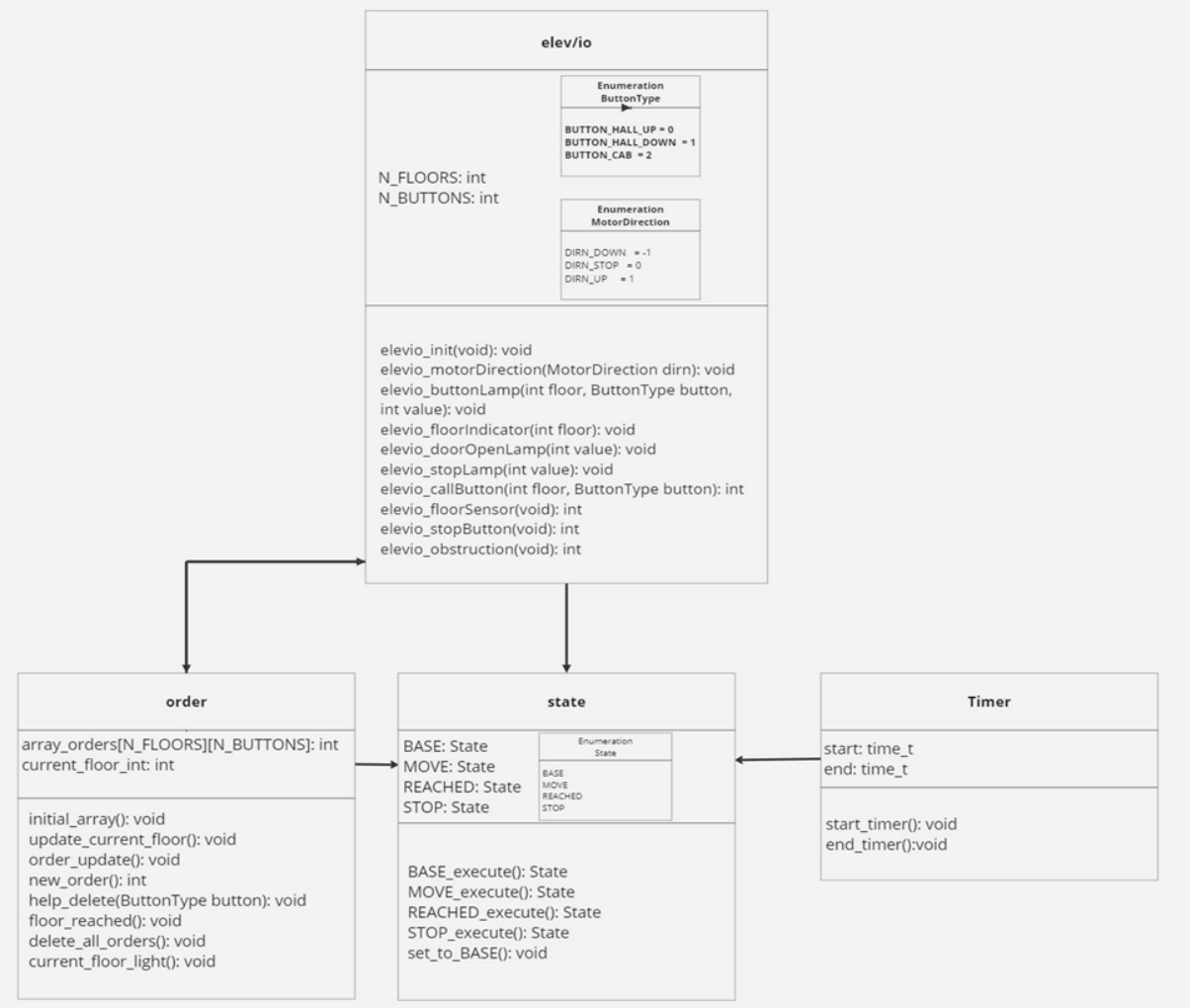
Først så vi at order-modulen måtte kunne lese fra elevio-modulen for å finne ut hvilke knapper som var trykket, da dette indikerer en bestilling. Siden lysene også indikerer en bestilling fant vi ut at vi ville ha lys-funksjonalitetene i order-modulen. Da måtte order-modulen i tillegg kunne skrive til elevio-modulen for å sette signaler til heis-panelene.

Etter at order-modulen har registrert en bestilling skal heisen bestemme hva den skal gjøre. Det vil si, den må bestemme om den skal bevege, i så fall hvor, eller om den skal stå stille. Dette tar states-modulen seg av, slik at order og states-modulen må kunne kommunisere.

States-modulen må vite hvor den står for å vite hvor den skal, og må ha tilgang til motor-funksjonene i elevio-modulen for å sette heisen i bevegelse. Derfor må state-modulen og elevio-modulen også kunne kommunisere. I state-modulen kan vi komme i en tilstand hvor vi trenger å holde styr på tiden, da må state-modulen kunne kommunisere med timer-modulen. Hvilken modul timer skulle kobles til så vi først når vi kom til moduldesignfasen, da vi før dette ikke visste om det egnet seg best å ha timer-modulen til order-modulen eller state-modulen.

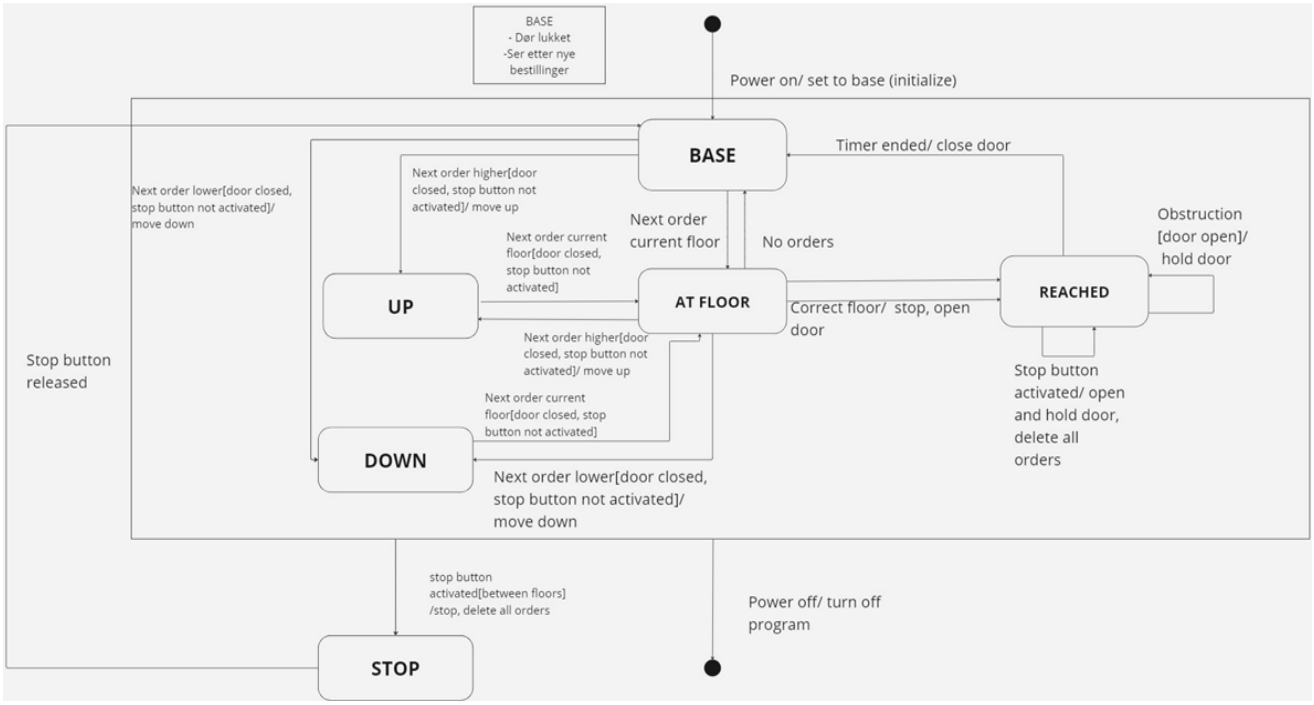
Nedenfor vises klasse-diagrammet (Figur 1) som gir oversikt over modulene våre, samt hvordan de kommuniserer med hverandre. Funksjonene og attributtene som vises kom vi fram til i moduldesignfasen og implementasjonsfasen.

Figur 1: Klasse-diagram



I arkitekturdesignfasen lagde vi også et tilstandsdiagram (Figur 2) for å få bedre oversikt over tilstandsforløpet til heisen. I tilstandsdiagrammet er det seks tilstander, eller «states», som heisen kan være i. Under moduldesignfasen fant vi ut at vi kunne slå tre av dem sammen, UP, DOWN og AT FLOOR til MOVING, for å få en enklere arkitektur. I kravene står det at heisen skal komme i en definert tilstand før den starter å ta bestillinger, og at denne prosessen kun skal gjøres én gang siden heisen videre skal kunne huske hvor den er. Denne definerte tilstanden har vi kalt BASE, og når programmet går inn i hoved-loopen settes heisen til BASE ved hjelp av en enkelt funksjon. Deretter kjører hoved-loopen helt til programslutt. I tilstandsdiagrammet vi lagde initielt hadde vi heller ikke en egen modul for timeren.

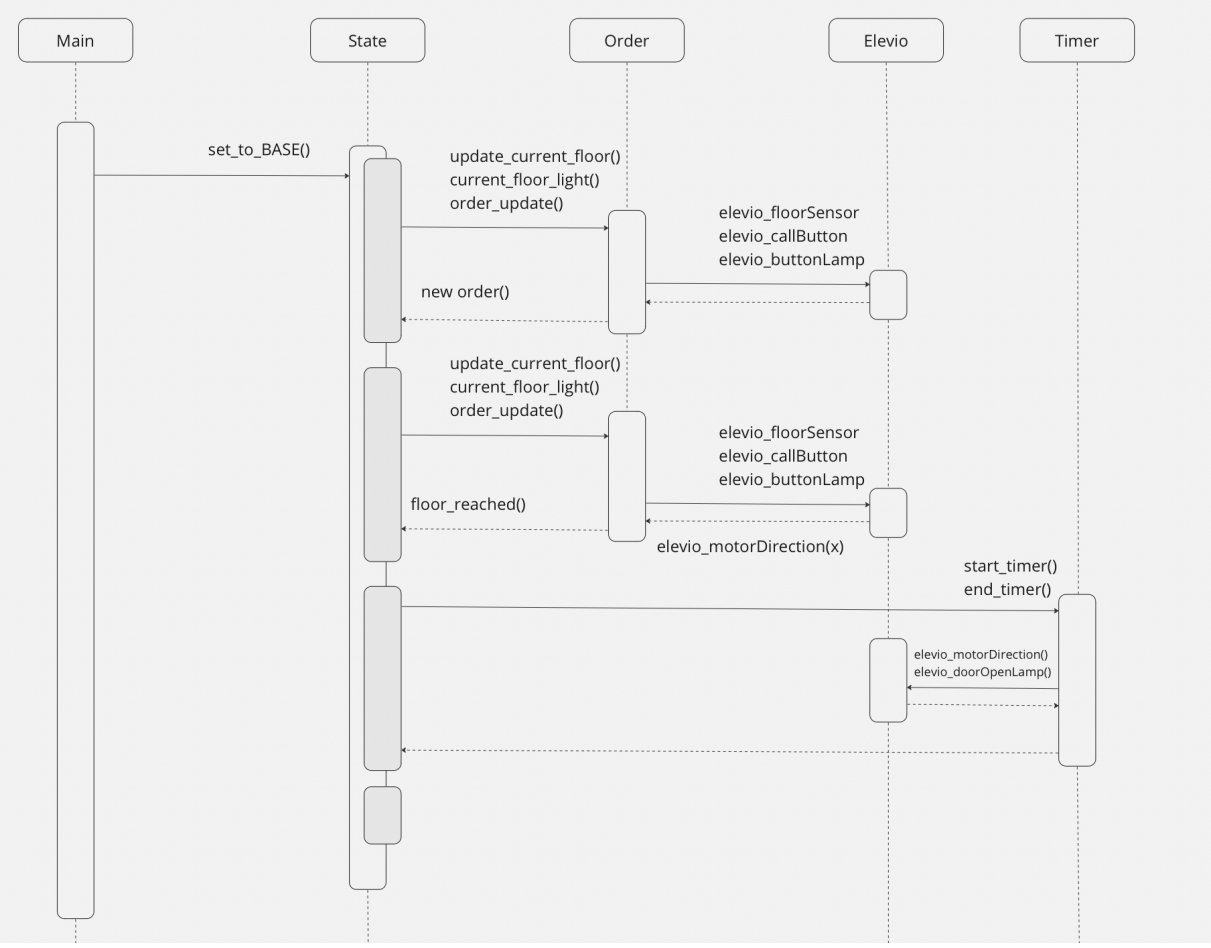
Figur 2: Tilstands-diagram



Etter vi hadde lagd et tilstandsdiagram satte vi oss inn i konkrete situasjoner for å finne ut hvordan modulene og de ulike funksjonene kommuniserte med hverandre. Vi lagde i denne fasen et sekvens-diagram (Figur 3). Vi satte oss først inn i enkle situasjoner der målet var å få en indikator på hvor mye de ulike modulene måtte kommunisere for enkle oppgaver. Denne kommunikasjonen ønsket vi skulle være så minimalt som mulig. Figur 3 viser et tilfelle der heisen settes i state BASE og får én bestilling. Sekvensdiagrammet ble utviklet underveis i prosessen da det fokuserte på hele systemet sitt funksjonalitet, som naturligvis ble forandret

underveis etter hvert som de ulike modulene og abstraksjonsnivåene ble testet. Likevel hjalp det oss i oppstartsfasen med å innse at vi ønsket en så enkel kommunikasjon som mulig mellom de ulike modulene da dette ville føre til et enkelt og oversiktlig sekvensdiagram, som igjen ville føre til en enklere eventuell feilsøking og integrasjonstesting. I den ferdige koden gjorde vi en endring fra kommunikasjonen mellom timer-modulen og elevio-modulen som vist i Figur 3, og endret til at timer-modulen bare kommuniserte med state-modulen. Dette var for å få mindre krysskommunikasjon mellom de ulike modulene, slik at all hovedkommunikasjon gikk gjennom state-modulen.

Figur 3: Sekvensdiagram



Både sekvens-diagram og tilstandsdiagrammet hjalp oss stort i moduldesignfasen når vi skulle lage states-modulen. I sekvensdiagrammet så vi tydeligere hvilke moduler som var avhengige av hverandre, og tilstandsdiagrammet gjorde oss observante på alle tilstandene og situasjonene vi kunne havne i. Det ga oss en pekepinn på hvilke funksjoner vi kom til å måtte trenge for å håndtere de ulike tilfellene. For eksempel håndtering av obstruksjonsbryteren.

Moduldesign:

Første steg i moduldesignfasen var å bli kjent med utdelt kode, elevio-modulen. Vi fant da ut at elevio benytter seg av polling. Dette ga oss utgangspunktet for hvordan vi kunne hente ut informasjon fra heis-panelene, hvilket hadde mye å si for hvordan order-modulen ble designet.

I order-modulen lagde vi en 4x3-matrise siden det er fire etasjer og (stort sett) tre tilhørende knapper til hver etasje: opp, ned og inne i heisen. Denne matrisen holder bestillingene som er blitt gjort (hvilke knapper som har blitt trykket). Initielt settes alle plassene i matrisen til null. Dersom en bestilling blir gjort, blir plassen i matrisen satt høy, dvs. til 1. For å hente ut bestillingene itereres det gjennom plassene i matrisen. I det itereringen treffer et 1-tall, blir etasjen lagret og holdt i en lokal variabel. Dette er neste etasje heisen skal til. Vi har også en global variabel som kontinuerlig holder etasjen heisen befinner seg i. Disse to variablene blir sammenlignet for å gi state-modulen beskjed om hva den skal gjøre. Dersom heisen befinner seg høyere eller lavere enn etasjen den skal til går den inn i MOVING-tilstanden, i det heisen er i rett etasje går den til REACHED-tilstanden, og fra REACHED går heisen til BASE hvor den itererer gjennom matrisen på nytt for å finne neste bestilling. Vi valgte en enkel prioritering av neste bestilling. Siden itereringen av matrisen alltid starter i 4.etasje, blir alltid bestillingene i den høyeste etasjen prioritert.

STOP-tilstanden i states-modulen skal kunne skje hvor som helst i hendelsesforløpet, og skal håndteres umiddelbart. Dette har vi løst ved at STOP kan returneres fra alle tilstander i states-modulen. Fra STOP skal heisen alltid gå tilbake til BASE og se etter nye bestillinger, slik at heisen fortsetter etter stopp-knappen er blitt sluppet.

V-modell/testing:

I utviklingsprosessen tok vi utgangspunkt i den pragmatiske V-modellen. I moduldesignprosessen ble vi som nevnt kjent med elevio-modulen via enhetststesting. Her gjorde vi små definerte tester av funksjonene slik at vi fikk oversikt over det som skulle være verktøykassa for funksjonene i order-modulen. Dette innebar å lage enkle funksjoner som blant annet skru av og på lys, samt teste hvordan vi kunne bruke og håndtere de ulike enum-elementene i elevio-modulen i fremtidige funksjoner. For eksempel hvordan styre motoren og skru av og på lyset til konkrete knapper.

Med hjelp ifra de oppgitte kravspesifikasjonene lagde vi ulike tester for de ulike modulene hver for seg. Etter at vi hadde blitt kjent med funksjonene i elevio-modulen, kunne vi lage og teste funksjonene fra order-modulen som skulle tilfredsstille kravspesifikasjonene. Flere av disse funksjonene ble endret på senere i prosessen, etter hvert som vi la til flere funksjonaliteter til heisen. Vi lagde for eksempel funksjonen som flyttet heisen til riktig bestilling basert på en funksjon vi lagde tidlig i prosessen. Denne funksjonen flyttet heisen til en etasje ved at vi manuelt skrev inn etasjenummer som input i koden. Dette ga oss mulighet til å teste enkeltfunksjoner og enkeltmoduler før vi satte sammen de ulike modulene.

State-modulen står for kommunikasjonen mellom de ulike modulene, så testingen av denne skulle vise oss om samhandlingen mellom de ulike modulene var som ønsket. Måten vi kvalitetssikret og feilrettet programflyten vår var ved å printe ut meldinger underveis som programmet kjørte. Vi brukte ikke debugging da vi fant ut at å printe ut i terminalen fungerte best for oss. Dette kunne være meldinger som at en etasje var nådd eller at en bestilling var tatt. Spesielt å printe ut hvilken tilstand heisen befant seg i hjalp oss med å løse programflyten. Pollingen som ble brukt i elevio-funksjonene svekket likevel denne metoden til en viss grad, da det var veldig mye meldinger som ble skrevet ut i terminalen.

Til slutt la vi til timer-modulen, og dermed fikk vi testet systemet som helhet. Siden vi hadde kvalitetssikret at de andre modulene fungerte som de skulle kunne vi teste denne modulen basert på de andre. Vi testet derfor at klokken og timer-modulen fungerte ved å se på om dørlyset og motoren holdt i tre sekunder.

Bruk av KI:

Som hjelpemiddel har vi benyttet oss av KI. Tjenesten vi har brukt er OpenAI sin ChatGPT. I all hovedsak har vi spurt etter eventuelle innebygde funksjoner i C, for eksempel når vi skulle lage timer-modulen. Når ChatGPT har foreslått noe har vi deretter søkt det opp i en vanlig søkemotor, som Google, for å undersøke funksjonene nærmere og se på eventuelle andre alternativer. Vi har altså brukt KI i lukket sløyfe. Vi har også brukt ChatGPT til å vise oss kode-syntaks i C, for eksempel hvordan en generell “switch-case” skal se ut, eller i tilfeller der vi ikke husker kommandoen i terminalen, men vet hva den gjør.

Bruken av KI i prosjektet har ført til at spørsmål som omhandlet koding har blitt løst fortere. Å søke i ChatGPT etter en funksjon vi vet vi har brukt i terminal tidligere går fortere enn å lete gjennom dokumenter og øvinger. Når ChatGPT har foreslått innebygde funksjoner har vi fått en pekepinn på hvilke nøkkelord eller biblioteker vi skal fortsette å søke innenfor. På en annen side kan det også føre til at vi snevrer oss mer inn, og at vi kunne ha funnet andre løsninger dersom vi ikke spurte ChatGPT først.

Refleksjon:

Vi valgte å holde designet og koden så enkel som mulig underveis. Å ha en matrise som holdt alle bestillingene funket bra da det ikke var noen forskjell på prioriteringen om det var trykket innenfor eller utenfor heisrommet. Siden alle bestillinger fra en etasje skulle regnes som tatt i det du ankom etasjen var det også lett å fjerne bestillingene med denne løsningen. Da satte vi bare hele raden til null når vi ankom etasjen. Vi er også fornøyd med prioriteringen vi valgte, med at høyest etasje alltid blir prioritert. Selv om prioriteringen er svært enkel, og ikke nødvendigvis den mest effektive, oppfyller den kravspesifikasjonene.

Noe vi burde gjort annerledes var å fokusere mer på å beholde et høyt abstraksjonsnivå i arkitekturdesignfasen. Det var lett forhaste seg i starten, for eksempel ved å begynne å tenke på funksjoner man trengte i de forskjellige modulene. Dette førte til at vi brukte tid på å tenke på funksjoner og hvordan de skulle samhandle i og utenfor modulene, før vi egentlig visste hvordan koden så ut. Mye av dette arbeidet ble derfor noe vi ikke fikk bruk for. Dette kan ses i sammenheng med at vi også burde satt oss inn i utdelt kode tidligere. Eksempelvis brukte vi unødvendig tid på å tenke på hardware-løsninger som det viste seg vi hadde fått utdelte funksjoner for. Vi kunne også hatt flere moduler, for eksempel en egen modul som tok seg av lys. Dette kunne kanskje ført til lettere vedlikehold av koden og høyere lesbarhet.

Det har vist seg at V-modellen er et godt hjelpemiddel for å sette seg ordentlig inn i kravene før implementasjonsfasen. Å ha brukt tid på arkitekturdesign forkorter tiden brukt moduldesign og implementasjon da man allerede har en høy forståelse av hva systemet krever av funksjonalitet, og hvordan man kan oversette dette. Vi har også lært at det kan være vanskelig å holde abstraksjonsnivået høyt nok i starten, men at dette er lurt da man ofte ender opp med å endre på oppsettet underveis når abstraksjonsnivået blir lavere, og flere krav blir tatt hensyn til. Da er det dumt å ha brukt tid på funksjonalitet det viser seg at man ikke trenger, ikke får til, eller som kan løses mer effektivt på andre måter. En annen ting vi har lært er

viktigheten av å være konsekvent helt fra start, spesielt med tanke på kodekvalitet. Eksempelvis å ha forklarende variabelnavn. Det er lett å glemme fra gang til gang hva en variabel gjør, og ikke minst å forstå for andre. I tillegg blir det mye å endre på til slutt dersom man ikke har vært konsekvent med konvensjoner.

UML og V-modellen har i vårt tilfelle bidratt til at koden er mer robust og lettere å vedlikeholde enn den hadde vært dersom vi ikke hadde brukt disse metodene i utviklingen. Det har blant annet sørget for at vi har definert funksjoner med funksjonalitet vi trenger flere steder i den hensikt at det kan gjenbrukes, i stedet for å skrive koden flere ganger. Dette var for eksempel tilfellet ved "timer" hvor vi først skrev funksjonaliteten til end_timer()-funksjonen flere steder i koden, før vi innså at det var lettere å skrive det om til en funksjon og bruke denne i stedet. Motivasjonen vår for å endre til å bruke en funksjon var i utgangspunktet å gjøre diagrammene penere, men vi skjønte at det også øker lesbarheten av koden og gjør vedlikeholdet av koden lettere. Bruken av UML har også gjort at vi under hele prosessen har tenkt at vi må ha klare og avgrensede moduler. Disse hadde antagelig ikke vært like gjennomtenkt dersom vi ikke hadde brukt UML-diagrammer og V-modellen. Bruk av V-modellen har ført til nøye testing av de individuelle funksjonene underveis, og funksjonene i samhandling med hverandre. Dette har ført til at koden ble mer robust, og at vi har plukket opp eventuelle feilkilder og grensetilfeller underveis. Samtidig har vi ikke tatt noen hensyn utover kravspesifikasjonene, så vi kunne ha skrevet mer kode som sikret oppførselen til heisen dersom den fikk uventet input, og slik fått en mer robust kode, dersom vi hadde tatt flere hensyn. Vi tror også koden er skalerbar og at det burde gå greit å legge til ekstra funksjonalitet uten å måtte endre mye på koden.