Heislab

Mai Alice Frøyshov Skjelby og Endre Magnar Åsgard

Mars 2025

1 Introduksjon

I faget TTK4235, **Tilpassede datasystemer**, undervist av Terje Jacobsson våren 2025, inngår et heisprosjekt som en obligatorisk del. I løpet av fem uker skal en fungerende heismodell implementeres og testes i Sanntidssalen. Prosjektet innebærer omfattende bruk av UML for systemdesign, programmering i C, og en endelig Factory Acceptance Test (FAT), hvor heisen testes basert på et sett med testspesifikasjoner for å vurdere funksjonaliteten [1].

Arbeidet er strukturert etter den pragmatiske V-modellen, som vist i figur 6. De første to ukene ble brukt til å utarbeide klassediagrammer og definere systemarkitekturen, før fokuset gradvis ble flyttet mot implementasjon etter hvert som systemdesignet ble mer modent. For å sikre strukturert utvikling og samarbeid, ble prosjektet versjonskontrollert med Git, og dokumentasjon ble generert ved hjelp av Doxygen for å oppnå en enkel, men oversiktlig beskrivelse av systemet. Prosjektet ble gjennomført i grupper på to studenter.

Formålet med dette prosjektet er å få en erfaring med alt som inngår i utviklingen et heissystem: UML, C-programmering, V-modellen, og tilstandsmaskiner.

Denne teksten går blant annet inn på funksjonen av moduler og deres metoder. Det er viktig å notere at parameterverdier ikke alltid inkluderes i teksten når en funksjon er nevnt, med mindre de er relevante, men full parameterliste for alle funksjonene vises i figur 1.

2 Systemdesign og implementasjon

2.1 Systemarkitektur

2.1.1 Systemarkitekturen til sluttproduktet

Figur 1 og figur 2 viser klassediagrammet for det ferdigstilte systemet. Klassediagrammet er fordelt i to figurer for å holde det oversiktlig. Sentralt i systemet er en "struct": ElevatorSM. En instans av ElevatorSM, referert til som "sm" i kodebasen, inneholder den sentrale informasjonen til heisen. Nytten med å lagre og oppdatere denne informasjonen i en struct-instans er at denne structen kan sendes ved referanse til funksjoner som trenger informasjonen, og tillater dem å oppdatere den. Dette er en mye smidigere og mindre komplisert løsning enn å sende variabel-verdier "fram og tilbake", og erfaringsmessig forenklet det debugging. Å tillate modulfunksjonene å oppdatere heisinformasjonen

direkte kan by på problemer dersom noe endres ved uhell, men forsøk på å strukturere koden på noen annen måte førte til uoversiktlig og over-komplisert kode. Dermed var den beste løsning den som ble implementert til slutt. Innad i ElevatorSM er det en instans av en Sensors struct og en Queue struct. Bruk av disse structene har samme nytte som ved ElevatorSM; alle viktige variabler lagres i ett sted og oppdateres direkte.

Figur 1 viser "composition" mellom elevator_types og start. Dette er en indikasjon på at instansen av ElevatorSM, Queue og Sensors er avhengig av start, og at de bare eksisterer innenfor levetiden til start. Med andre ord, det finnes ingen instanse av disse uten start-modulen [2]. Kode-messig er dette utført ved at start_init() funksjonen instansierer ElevatorSM (og dermed Queue og Sensors), og at den kontinuerlig kaller TSM_update(&(sm)) i en while-løkke for å oppdatere tilstandsmaskinen.

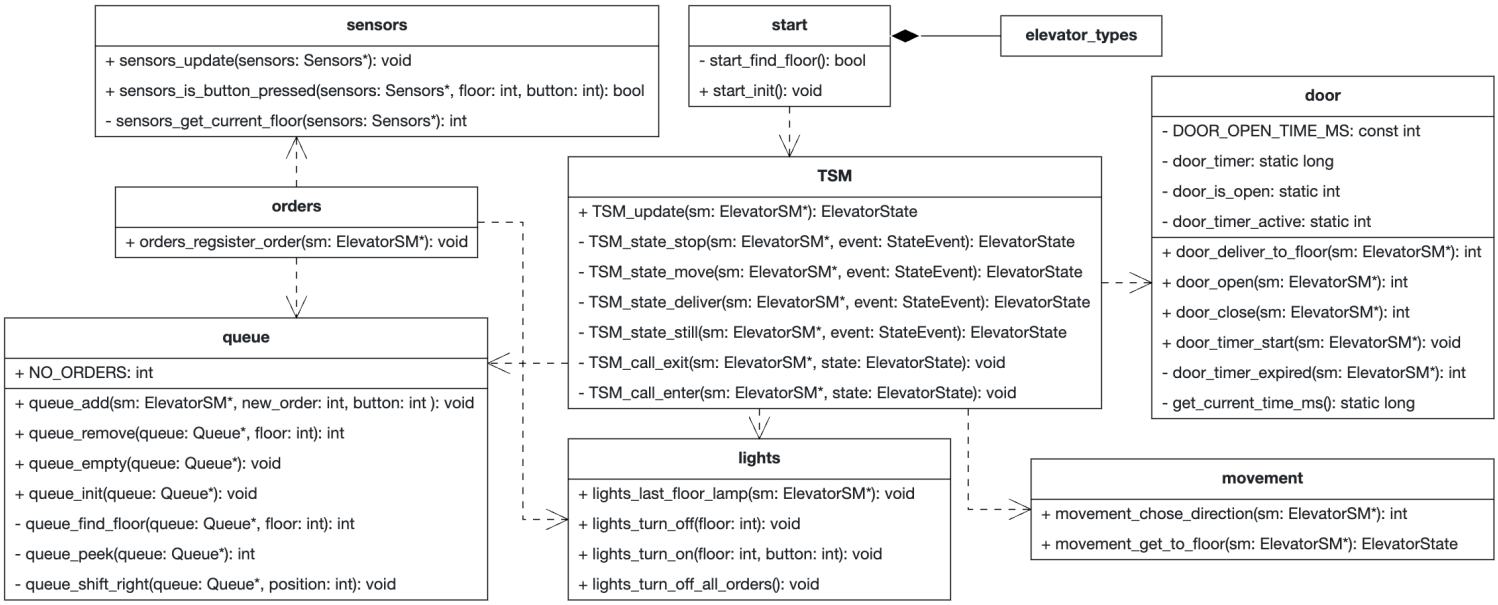

Figure 1: Klassediagram av det fungerende systemet.

Systemet er basert på en tilstandsmaskin. Denne tilstandsmaskinen benytter seg av de andre modulene for å utføre heis-funksjonlaiteten og for å oppdatere heis-informasjonen. Et forsøk ble gjort på å designe modulene på en slik måte at deres brukergrensesnitt med tilstandsmaskinen var så enkel som mulig, og at så mye av heisfunkjsonaliteten som mulig var innebygget i modulene.

Et eksempel på dette er door-modulen: i deliver-staten kalles door_deliver_to_floor() fra tilstandsmaskinen, og all funksjonaliteten beskrevet i kravspesifikasjonene D2 - D4 utføres innad i door-modulen [1]. Når dørene har åpnet og lukket seg, returnerer funksjonen 1. Denne oppbyggingen av systemarkitekturen hadde ført til et oversiktlig og ukomplisert system som var enkelt å debugge - men vi møtte på noen utfordringer som gjorde at dette brukergrensesnittet ble litt mer komplisert. Stop-staten bruker både door_open(), door_close(), og door_timer_start() for å åpne dørene hvis stopp-knappen trykkes mens heisen er i en etasje. Denne funksjonaliteten var implementert slik fordi door_deliver_to_floor() ikke var egnet til denne oppgaven, og på grunn av tidsbegrensningen på prosjektet så måtte vi bruke den kjappeste løsningen. Med mer tid så hadde det å skrive en egen funksjon for dette i door-modulen vært en bedre implementasjon.

Denne utfordringen oppstod flere steder, og gjorde at koden ble mer uoversiktlig og komplisert enn ønsket. Dette kan tydelig sees i mengden offentlige funksjoner i figur 1. Resten av modul-interaksjonene hadde ikke vært like enkle å forenkle som door-eksempelet, men det er verdt å nevne at det er mye som kunne forbedres struktur-messig. Dette hadde innebåret å skape nye moduler og omfordele oppgavene deres.

Det er viktig å nevne at systemet også benytter seg av en driver, "elevio", for å styre heisen og hente

ut informasjon fra heisen. Denne driveren er ikke inkludert i klassediagrammet, men omtrent alle modulene tar i bruk funksjoner fra elevio-modulen. I tillegg brukes en start-modul til å initialisere heisen og sørge for at den er i en definert tilstand før heisen begynner å ta imot bestillinger og reagere på stopp-knappen.

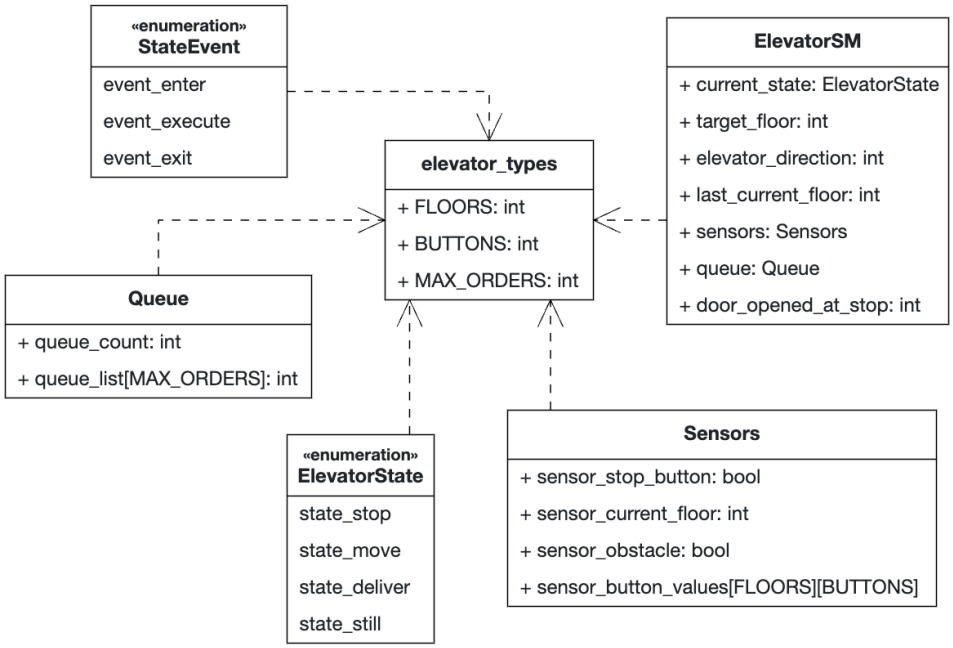

Figure 2: Klassediagram for elevator_types modulen.

2.1.2 Planlegging av systemarkitekturen

Første steget av dette prosjektet var å utarbeide systemarkitekturen utifra kravspesifikasjonene oppgitt i oppgaveteksten [1]. Denne gikk ut på å forsøke å finne en sammensetning moduler som jobber godt sammen og som oppfyller kravene på best mulig måte. Modulene skal ideelt sett ha så isolert oppførsel som mulig. Dette innebærer at de er lite avhengig av hverandre, og at hver modul har en tydelig funksjon [1].

En heis trenger minne for at bestillinger kan bli gjennomført og for at kø-prioritering skal være mulig. I denne oppgaven hadde vi derfor fokus på å skille modulene som trengte hukommelse og ikke, noe som innebar at vi strukturere programmet som en tilstandsmaskin med hjelpemoduler [1], som beskrevet i 2.1.1.

Figur 3 viser det første klassediagrammet som ble utarbeidet. Dette ga oss et godt utgangspunkt å arbeide fra, og strukturen på prosjektet holdt seg relativt lik gjennom implementeringsprosessen. Den største endringen i systemarkitekturen vi gjorde underveis var at vi samlet alle viktige structs og enums i elevator_types og brukte ElevatorSM for å lagre og huske viktig informasjon om heisen. Som vist i figur 3, var den opprinnelige planen å lagre nødvendig informasjon i modulene de var relevante i. Andre moduler kunne da hente ut informasjonen hvis de trengte det. Det ble fort åpenbart at dette var en dårlig løsning, ettersom hver modul trengte minne for at dette skulle fungere. Vi gikk fort over til den løsningen beskrevet i 2.1.1. Utenom dette endret vi en del ved de individuelle modulene, men arkitekturen forble relativt lik, med unntak av tillegget av lights-modulen.

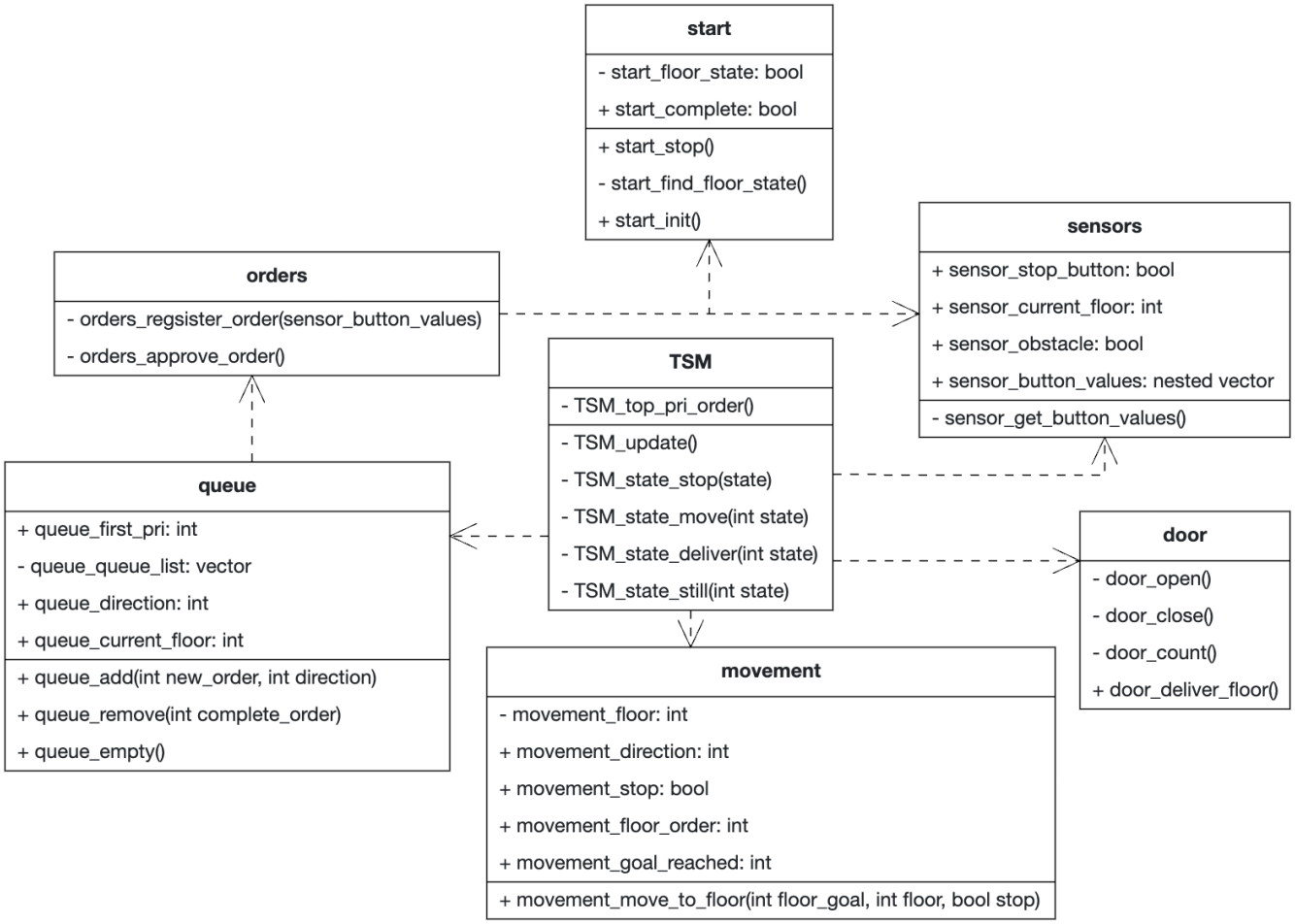

Figure 3: Første utkast av klassediagrammet til systemet, brukt som basis for kodeimplementasjon

2.2 Moduldesign

2.2.1 TSM-modulen

Tilstandsmaskin-modulen (heretter TSM) har i seg selv ikke minne - det er start-modulen som lagrer og husker heisinformasjonen, og TSM benyttes for å bytte mellom stater og utføre tilstandenes funksjonalitet. Figur 4 viser et tilstandsmaskindiagram for systemet. Her er start inkludert som en tilstand, selv om dens funksjonalitet egentlig er implementert i start-modulen og ikke TSM. Dette er for å tydeliggjøre oppførselen til heisen under initialiseringen og for å fremme kravene til oppstart av systemet.

TSM_update() kalles kontiuerlig i en while-løkke i start-modulen. I funksjonen utføres aktiviteten i execute-tilstanden til nåværende tilstand ved bruk av en switch-funksjon. I tillegg kalles noen hjelpemodul-funksjoner. Hvis execute-tilstanden oppdaterer heisens tilstand, som f.eks. still-tilstanden som oppdaterer heistilstanden til move hvis en bestilling blir plassert, blir exit utført på daværende tilstand og enter utført på den nye tilstanden. Det er kun stop-tilstanden som har en exit-prosedyre implementert, som vist i figur 4, men alle tilstandene har exit innebygget for å gjøre koden skalerbar. Dette kan være spesielt nyttig hvis flere sikkerhetstiltak skal tas i bruk.

Stop-tilstanden har litt annen implementasjon i TSM_update(), ettersom heisen umiddelbart skal inn i tilstanden om stoppknappen trykkes. Derfor sjekkes dette i begynnelsen av TSM_update(), og heisens tilstand endres manuelt til stopp (utenfor switch-funksjonen) hvis det er tilfellet. Når stopp-knappen slippes blir tilstanden til heisen endret til still.

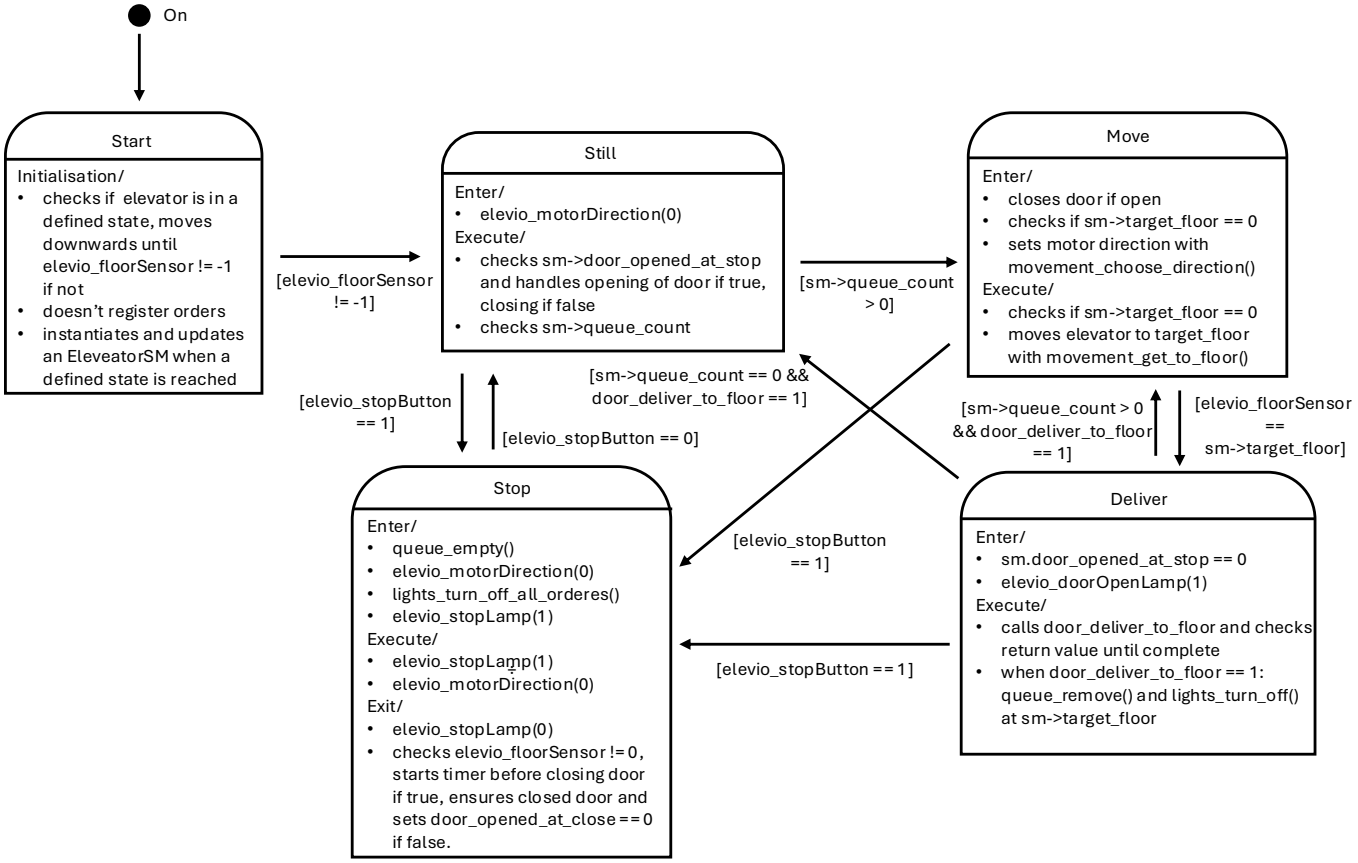

Figure 4: Tilstandsmaskindiagram for systemet

2.2.2 Grunnleggende moduldesign og interaksjoner

Funksjonen til hjelpemodulene skal ikke utdypes individuelt i denne rapporten ettersom den er forklart godt nok av figur 1. Det er viktig å nevne at alle modulene inkluderer header-filen elevator_types, siden alle modulene trenger tilgang til type-deklarasjonene for at koden skal fungere. Dette er utelatt fra klassediagrammet ettersom det hadde gjort for et uoversiktlig diagram, og det er dessuten ikke veldig relevant for strukturen av systemet. Det som er viktig å diskutere er interaksjonene mellom modulene og hvordan de tillater heissystemet å ta imot og gjennomføre bestillinger.

I TSM_update() kalles sensors_update(), en funksjon som itererer gjennom de forskjellige bestillingsknappene i heisen og registrerer om de er trykket inne eller ikke. Denne funksjonen oppdaterer sensors.sensor_button_values. Dermed kalles orders_register_order() i TSM_update(), og denne funksjonen itererer gjennom sensors-vektoren for å hente ut informasjon om hvilke knapper har blitt trykket. Funksjonen sjekker da at betingelsene for å kunne motta en bestilling er oppfylt før queue_add() kalles med den nye bestillingen og lyset for bestillinger er skrudd på. queue_add() funksjonen sjekker at bestillingen ikke allerede eksisterer, og plasserer den riktig sted i køen (også lagret som en ElevatorSM-variabel). Kø-prioriteringen er i henhold til kravspesifikasjonene [1], og i tillegg så prioriteres on-the-way cabin bestillinger. Figur 5 viser en skisse av et sekvensdiagram som visualiserer modulinteraksjonene når en bestilling plasseres. Diagrammet er veldig grunnleggende og illustrerer kun en situasjon: en bestilling plasseres, og køen er tom etter at bestillingen har blitt utført. Fra diagrammet er det tydelig at systemet kan være for komplisert, og at det er unødvendig mange moduler. Loop-en i TSM fremhever at execute kalles om og om igjen på tilstanden.

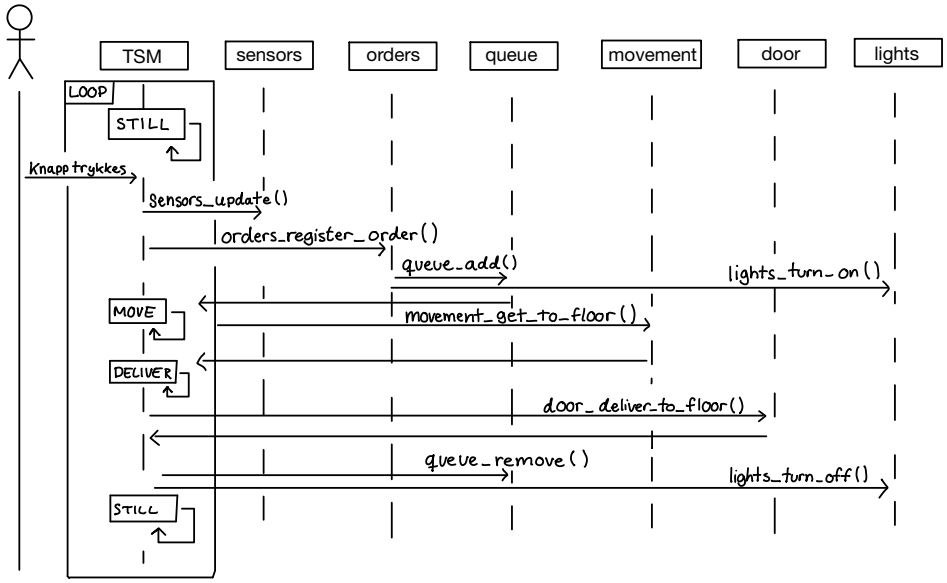

Figure 5: Skisse av et sekvensdiagram for gjennomføring av en bestilling.

2.3 Testing og verifikasjon

2.3.1 Enhetstesting

For å sikre at hver modul fungerer som forventet før den integreres i systemet, er enhetstesting en viktig del av utviklingsprosessen i V-modellen. Enhetstester isolerer og tester individuelle moduler for å bekrefte at de fungerer som spesifisert. Dette speiler moduldesignfasen, der hver komponent verifiseres før integrasjon [1].

Gode enhetstester bør være små og veldefinerte, med fokus på enkeltfunksjoner. Kvaliteten på testingen handler ikke om antall tester, men om å dekke ulike scenarier, spesielt edge-cases [1]. Systematisk enhetstesting gjør det enklere å identifisere og rette opp feil tidlig, noe som forenkler integrasjonen i neste fase.

I prosjektet var enhetstesting særlig viktig for køsystemet, siden denne modulen kan være utfordrende å implementere riktig. Derfor testet vi alle funksjonene i denne modulen nøye og individuelt. Enhetstesting var ikke gjennomført grundig på alle modulene, og dette førte til tidskrevende og unødvendig komplisert debugging i integrasjonstesting-fasen.

2.3.2 Integrasjonstesting

Etter enhetstesting er det viktig å verifisere at modulene fungerer korrekt sammen. Integrasjonstesting sikrer at de ulike delene av systemet kommuniserer som forventet og at det ikke oppstår feil når de brukes om hverandre. For heisprosjektet så gjelder dette eksempelvis modulene,

- Order-modulen: Håndtering av bestillinger.
- Queue-modulen: Korrekt prioritering og prosessering av ordre.
- Sensor-modulen: Riktig registrering av etasjer, dører og nødstopp.
- Lights-modulen: Håndtering av lys.

Order-modulen benytter alle modulene nevnt ovenfor. Dette gjør det essensielt å teste at de fungerer sammen som en helhet. En feil i én modul kan føre til feil i hele systemet. Nedenfor er et eksempel på hvordan order-modulen registrerer en bestilling, legger den til i køen og oppdaterer lysene,

```
#include "orders.h"

void orders_register_order(ElevatorSM *sm) {
    for (int floor = 0; floor < FLOORS; floor++) {
        for (int button = 0; button < BUTTONS; button++) {
            if (sensors_is_button_pressed(&(sm->sensors), floor, button)) {
                queue_add(sm, floor, button); // Legger til ordre
                lights_turn_on(floor, button); // indikatorlys
...
```

Her ser vi at orders_register_order() sjekker om en knapp er trykket gjennom sensorene, registrerer bestillingen i køen via queue_add(), og aktiverer riktig indikatorlys med lights_turn_on(). Gjennom integrasjonstesting kan vi verifisere at disse modulene samhandler korrekt uten uventede feil.

Det oppstod en del feil ved integrasjonstesting ettersom enhetstestingen ikke hadde blitt gjennomført nøye nok. Løsningen på dette var å skrive en rekke print-statements i de fleste funksjonene som ga informasjon hver gang tilstanden ble endret, dørene ble åpnet og lukket, en bestilling ble motatt, og mer. Køen var også skrevet ut ved hver oppdatering av tilstandsmaskinen. Denne teknikken gjorde det mulig å identifisere en del feil og hvor de oppstod. For eksempel, så hadde ikke dør-timeren ikke blitt konvertert riktig, så den telte 3000 ms i CPU-tid, ikke vanlig tid. Dette gjorde at dørene holdt seg åpen altfor lenge.

2.3.3 Factory acceptance test (FAT)

Den siste og mest avgjørende testen i prosjektet var Factory Acceptance Test (FAT). Formålet med denne testen var å sikre at heissystemet oppfylte de forhåndsdefinerte kravene ved hjelp av testspesifikasjonene oppgitt i oppgaveteksten [1].

Resultatene fra FAT ga blant annet en liten feil i måten bestillinger ble håndtert hvis heisen ble stoppet mellom etasjer. Hvis en bestilling ble plassert i etasjen heisen sist var i, etter stopp-knappen var trykket, så reagerte ikke heisen på bestillingen. Det er trolig fordi heisen trodde den fremdeles var i den etasjen. En eventuell videreføring av dette prosjektet ville rettet opp i dette.

3 Refleksjoner

3.1 Bruk av KI

KI-modeller, mer spesifikt OpenAIs *ChatGPT* [3] og Anthropics *Claude* [4], har vært nyttige verktøy i prosjektet. Vi har blant annet brukt den til å forbedre kode, omformulere setninger, og feilsøke problemfylt kode.

Bruken av KI har vært i lukket sløyfe. KI var benyttet som et støtteverktøy, ettersom åpen-sløyfe bruk av KI er mindre læringsrikt og ofte fører til et verre produkt. Vi har vært kritiske til svarene vi fikk, og har validert løsninger før implementering. En dyp forståelse av systemets oppbygning er avgjørende, og det ville vært uansvarlig å blindt stole på KI-genererte løsninger. Samtidig mener vi at det er masse nytte i å ta i bruk KI-verktøy, så lenge man gjør det riktig.

Bruk av KI gjorde det mulig å skape et bedre produkt med samme tidsbegrensing, ettersom det var nyttig ved problemløsning. Vi brukte for eksempel KI for hjelp med å få heis-simulatoren til å kjøre på Mac, for å generere enkelte Git-kommandoer, og for å hjelpe oss skrive eller endre på enkelte linjer med kode. Vi har ikke brukt KI til å generere store eller sammenhengene deler av koden, men denne lukket-sløyfe bruken av KI gjorde at vi reduserte mengden tid brukt på generell debugging og google deep-dives for å finne akkrurat den ene løsningen som fungerer for oss. Bruk av KI har essensielt gitt oss mer effektivt og bedre arbeidsflyt.

3.2 Bruk av V-modellen

I dette prosjektet har vi fulgt den pragmatiske V-modellen [5] for å strukturere utviklingen av heisen. Bruk av modellen bidrar ikke bare til at prosjektet når sitt mål, men sikrer også en god kobling mellom implementasjon og testing på ulike abstraksjonsnivåer. Vi jobbet i henhold til modellen med måten vi strukturerte ideene våre, implementerte dem, så testet dem.

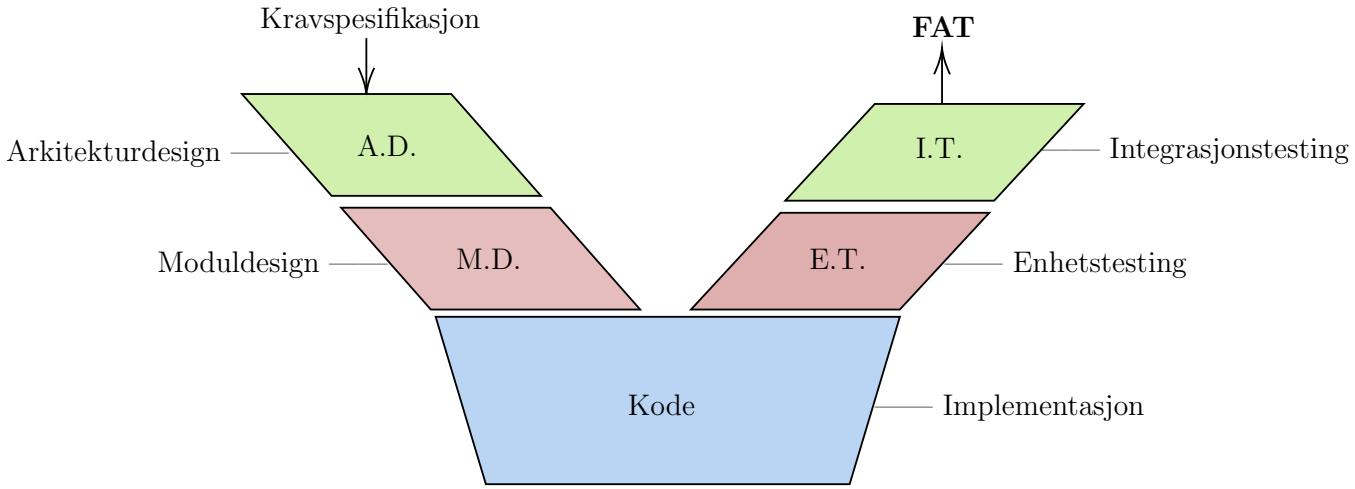

Figure 6: En illustrasjon av den pragmatiske V-modellen fulgt for prosjektet [1].

Som nevnt tidligere i rapporten så fulgte vi ikke V-modellen nøye nok. Enhetstesting for hvert modul ble ikke gjennomført grundig, og dette førte til mer tidskrevende debugging senere i prosessen. Tidlig planlegging av tester er en essensiell del av V-modellen [5], og dette var ekstremt nyttig ettersom testspesifikasjonene [1] vi måtte forholde oss til ikke ble endret gjennom prosjektet. Vi kunne implementert dette bedre på modul-nivå for å ha bedre definert oppførsel i hver modul før vi skrev koden. Problemet med forutsetningen at krav ikke vil endre seg gjennom prosessen er at det ikke er en virkelighetsnær forutsetning. Krav til et produkt endrer seg ofte, og om dette hadde skjedd, hadde det bydd på problemer. V-modellens mangel på fleksibilitet er en av de største kritikkene stilt til den [5], og dette var en påkjenning selv om testkravene ikke endret seg.

Gjennom utviklingen ble det tydelig at det kanskje finnes en bedre løsning for systemarkitekturen som kunne gitt mindre avhengighet mellom modulene og mindre kompleksitet. Siden implementasjonen, i følge V-modellen, gikk ut på å skrive hver modul og teste dem individuelt, så hadde vi ikke mulighet til planlegge og implementere en ny løsning innenfor tidsfristen. En bedre løsning kunne vært å ta i bruk en agile-metode. Her hadde vi begynt med et mye enklere system og utviklet den iterativt [5], og dette hadde gjort at vi hadde oppdaget problemene med systemarkitekturen tidligere.

3.3 Bruk av UML

For å strukturere programvaren benyttet vi oss av UML (Unified Modeling Language) i henhold til *UML Distilled* av Martin Fowler [6]. Bruk UML hjalp oss visualisere hvordan systemets moduler skulle samhandle og hvordan tilstandsmaskinen skulle fungere. Vi begynte prosjektet med å utvikle et klassediagram, vist i figur 3, som gjorde det enkelt å fordele arbeidet og holde koden modulær og skalerbar. Klassediagrammet var oppdatert iterativt gjennomarbeidet, og sluttresultatet vises av figur 1 og figur 2. Denne oppdateringen gjorde samarbeid enklere. Et tilstandsmaskindiagram har også blitt utviklet for å tydeliggjøre funksjonen av tilstandsmaskinen, som vist i figur 4, og vi opplevde dette som et nyttig verktøy for å tydeliggjøre funksjonen av tilstandsmaskinen.

Vi tegnet også en skisse av et sekvensdiagram, vist i figur 5, som et hjelpeverktøy for å tydeliggjøre funksjon av systemet. Vi opplevde at denne skissen ikke bidro med spesielt mye mer forståelse for systemet sammenlignet med klassediagrammet og tilstandsmaskindiagrammet, så vi utviklet ikke sekvensdiagrammet videre i prosessen. UML diagrammer er også nyttig i etterkant av prosjektet, ettersom de gjør det enklere å forklare systemets oppbygging og gjør vedlikehold av andre enn skaperne av systemet enklere. Derfor har vi valgt å ha med skissen, ettersom den gjelder for sluttproduktet og den kan bidra til noen andres forståelse.

3.4 Bruk av Git

Git har vært et essensielt verktøy for samarbeid i dette prosjektet, og vi valgte å bruke GitHub. Bruken av Git bidro til et mer organisert prosjekt, og gjorde det lettere å samarbeide ettersom man får nesten umiddelbar tilgang til endringene andre gjør. Det ble også lett å holde oversikt over hvem som har gjort hvilke endringer. Dette har vært spesielt nyttig i et gruppeprosjekt, hvor flere personer har jobbet med ulike deler av samme systemet samtidig. Vi jobbet konsekvent på hovedgrenen gjennom hele prosjektet, da det ikke oppstod et behov for flere grener, men bruk av Git hadde vist veldig nyttig hvis det ble behov for det.

3.5 Robusthet, skalerbarhet og vedlikehold

Selv om oppgaven ikke eksplisitt krevde fokus på robusthet, skalerbarhet og vedlikehold, er dette viktige egenskaper i mange produksjonssystemer. Det er derfor relevant å vurdere hvordan metodene vi brukte i prosjektet har påvirket disse aspektene. Bruken av UML har bidratt til en mer modulær kodebase, noe som er en sentral faktor for skalerbarhet. Klassediagrammene hjalp oss med å definere tydelige grensesnitt mellom modulene, noe som gjør det enklere å utvide systemet uten å introdusere unødvendige avhengigheter. I tillegg fungerer diagrammene som en form for dokumentasjon, som kan være nyttig for videre utvikling og vedlikehold av koden.

Når det gjelder V-modellen, har den bidratt til en systematisk utviklingsprosess der testing har vært en integrert del av arbeidet. Dette har indirekte styrket robustheten ved at feil ble oppdaget og håndtert tidlig i prosessen. Av den grunn hadde robusetheten selvfølgelig vært enda bedre om vi hadde fulgt V-modellen mer nøye.

Samlet sett har bruken av UML og V-modellen trolig vært mer til hjelp enn til hinder for robustheten, skalerbarheten, og vedlikehold av koden. Selv om de ikke var direkte prioriterte i prosjektet, har metodene vi brukte, spesielt gjennom modulær struktur, bidratt til en mer oversiktlig og vedlikeholdbar kodebase.

3.6 Kodekvalitet

Gjennom prosjektet har vi forsøkt å forholde oss til anbefalingene om kodekvalitet i oppgaveteksten [1]. Vi har vært relativt vellykket i å gjøre dette, men kodekvaliteten er verre enn ønsket noen steder. Det er både en modul og en struct som heter queue, og det samme med sensors. Dette oppstod fordi structene egentlig var definert i modulene ved samme navn, men det ble nødvendig å endre på dette for å unngå sirkulær avhengighet mellom modulene. Fra en OOP-vinkling er både structene og modulene klasser, og vi har i praksis to eksempler av klasser med nesten samme navn. Structene har stor forbokstav, men det er den eneste forskjellen. Dette er en ganske stor oversikt som var oppdaget etter FAT-en, og som kan påvirke skalerbarhet og vedlikehold hvis ikke det blir endret.

Som nevnt kunne også modulene vært mindre tilknyttet andre moduler, men funksjonene i koden har ganske god kodekvalitet når det gjelder avhengighet på andre funksjoner og ansvarsområde. Det er noen funkjsoner, som TSM_update(), som har stor avhengighet på andre funksjoner, men dette er unngåelig. Det er få kommentarer i koden, ettersom mye av koden er lesbar og åpenbar, men det er steder det mangler kommentarer. Koden var dokumentert med Doxygen, noe som gjør at mangelen på kommentarer heldigvis ikke påvirker vedlikehold og skalerbarhet betydelig. Doxygen dokumentasjonen er lagt til som appendiks A.

4 Konklusjon

I konklusjon har vi skrevet et fungerende heis-system av relativt god kvalitet. Bruk av UML og V-modellen styrket kodekvaliteten og bidro til at systemet ble robust, skalerbart, og vedlikeholdbart, men V-modellens mangel på fleksibilitet var også et hinder som gjorde at vi oppdaget store feil for sent. Systemet er naturligvis ikke perfekt, og det er mye som kunne blitt forbedret. Systemarkitekturen kunne vært enklere, med mer isolerte moduler, og det er visse aspekter av kodekvaliteten som burde forbedres. Koden skulle også vært mer grundig kommentert. I tillegg var det litt funksjonalitet som manglet, som oppdaget under FAT-en. Til tross for dette er vi fornøyd med sluttproduktet. Formålet med prosjektet, å få erfaring med tilstandsmaskiner, UML, V-modellen, og C-programmering, er oppfylt.

References

[1] K. B. et al, "Ttk4235 - tilpassede datasystemer: Heisprosjektet," Norges teknisk-naturvitenskapelige universitet (NTNU), Institutt for teknisk kybernetikk, Tech. Rep., 2025.

[2] T. Informatik, *Uml composition - definition, explanation, examples*, Accessed: 20 March 2025, 2025. [Online]. Available: `https://t2informatik.de/en/smartpedia/uml-composition/`.

[3] OpenAI, *Chatgpt: Optimizing language models for dialogue*, `https://openai.com/blog/chatgpt`, Accessed: 2025-03-21, 2025.

[4] Anthropic, *Claude: An ai assistant by anthropic*, `https://www.anthropic.com/index/introducing-claude`, Accessed: 2025-03-21, 2025.

[5] G. Mathisen, *Systemutvikling i henhold til v-modellen*, `https://ntnu.blackboard.com`, Forelesningsslides i TTK4235 - Systemutvikling, NTNU, 2025.

[6] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd. Addison-Wesley Professional, 2003, ISBN: 978-0321193681.

A Doxygen Dokumentasjon

Doxygen-dokumentasjonen er inkludert som en vedlagt seksjon nedenfor.

My Project

Generated by Doxygen 1.13.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 ElevatorSM Struct Reference

Structure containing the essential information about the elevator that the elevator itself cannot provide (since the current floor can be provided by the driver, it is not included here.)

`#include <elevator_types.h>`

Public Attributes

- ElevatorState **current_state**
- int **target_floor**
- int **elevator_direction**
- int **last_current_floor**
- Sensors **sensors**
- Queue **queue**
- int **door_opened_at_stop**

3.1.1 Detailed Description

Structure containing the essential information about the elevator that the elevator itself cannot provide (since the current floor can be provided by the driver, it is not included here.)

The documentation for this struct was generated from the following file:

- source/elevator_types.h

3.2 Queue Struct Reference

Contains the queue list for handling orders.

`#include <elevator_types.h>`

Public Attributes

- int queue_count
- int queue_list [MAX_ORDERS]

3.2.1 Detailed Description

Contains the queue list for handling orders.

@estruct Queue

3.2.2 Member Data Documentation

3.2.2.1 queue_count

```
int Queue::queue_count
```

Amount of orders in the list

3.2.2.2 queue_list

```
int Queue::queue_list[MAX_ORDERS]
```

The queue list, containing orders for the elevator to service

The documentation for this struct was generated from the following file:

- source/elevator_types.h

3.3 Sensors Struct Reference

Contains the sensor information for the elevator. Useful for debugging in case of hardware problems.

```
#include <elevator_types.h>
```

Public Attributes

- bool sensor_stop_button
- int sensor_current_floor
- bool sensor_obstacle
- bool sensor_button_values [FLOORS][BUTTONS]

3.3.1 Detailed Description

Contains the sensor information for the elevator. Useful for debugging in case of hardware problems.

@estruct Sensors

3.3.2 Member Data Documentation

3.3.2.1 sensor_button_values

```
bool Sensors::sensor_button_values[FLOORS][BUTTONS]
```

A matrix containing the information for each of the order buttons in the elevator.

3.3.2.2 sensor_current_floor

```
int Sensors::sensor_current_floor
```

The elevators current floor

3.3.2.3 sensor_obstacle

```
bool Sensors::sensor_obstacle
```

Obsutruction status

3.3.2.4 sensor_stop_button

```
bool Sensors::sensor_stop_button
```

Stop button status

The documentation for this struct was generated from the following file:

- source/elevator_types.h

Chapter 4

File Documentation

4.1 source/door.h File Reference

Module that handles opening and closing the door, including holding it open for 3 seconds and checking for obstructions.

```
#include "elevator_types.h"
#include "driver.h"
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
```

Functions

- int door_open (ElevatorSM *sm)

  Opens the elevator door if the elevator is in a defined state (at a floor). The elevator will not open if it is not at a floor.
- int door_close (ElevatorSM *sm)

  Attempts to close the elevator door, checking for an obstruction. If an obstruction occurs, the door will wait 3 more seconds before it closes.
- void door_timer_start (ElevatorSM *sm)

  Starts a timer, used to hold the door open for 3 seconds.
- int door_timer_expired (ElevatorSM *sm)

  Checks if the door timer has expired.
- int door_deliver_to_floor (ElevatorSM *sm)

  Main door control function for floor delivery. To be called my the state machine. Designed to be called continuously until it returns 1.

4.1.1 Detailed Description

Module that handles opening and closing the door, including holding it open for 3 seconds and checking for obstructions.

4.1.2 Function Documentation

4.1.2.1 door_close()

```
int door_close (
            ElevatorSM * sm)
```

Attempts to close the elevator door, checking for an obstruction. If an obstruction occurs, the door will wait 3 more seconds before it closes.

Parameters

in	*sm*	Pointer to the elevator state machine

Returns

1 if door was closed, 0 if not.

4.1.2.2 door_deliver_to_floor()

```
int door_deliver_to_floor (
            ElevatorSM * sm)
```

Main door control function for floor delivery. To be called my the state machine. Designed to be called continuously until it returns 1.

Parameters

in	*sm*	Pointer to the elevator state machine

Returns

0 if door process still in progress, 1 if completed

4.1.2.3 door_open()

```
int door_open (
            ElevatorSM * sm)
```

Opens the elevator door if the elevator is in a defined state (at a floor). The elevator will not open if it is not at a floor.

Parameters

in	*sm*	Pointer to the elevator state machine

Returns

1 if door was successfully opened, 0 if it failed

4.1.2.4 door_timer_expired()

```
int door_timer_expired (
            ElevatorSM * sm)
```

Checks if the door timer has expired.

Parameters

in	*sm*	Pointer to the elevator state machine

Returns

1 if timer has expired, 0 otherwise

4.1.2.5 door_timer_start()

```
void door_timer_start (
            ElevatorSM * sm)
```

Starts a timer, used to hold the door open for 3 seconds.

Parameters

in	*sm*	Pointer to the elevator state machine

4.2 door.h

Go to the documentation of this file.

```
00001
00006 #pragma once
00007
00008 #include "elevator_types.h"
00009 #include "driver.h"
00010 #include <stdio.h>
00011 #include <stdbool.h>
00012 #include <time.h>
00013
00021 int door_open(ElevatorSM *sm);
00022
00030 int door_close(ElevatorSM *sm);
00031
00037 void door_timer_start(ElevatorSM *sm);
00038
00045 int door_timer_expired(ElevatorSM *sm);
00046
00054 int door_deliver_to_floor(ElevatorSM *sm);
```

4.3 driver.h

```
00001 #include "driver/elevio.h"
```

4.4 source/elevator_types.h File Reference

Defines neccessary structs and enums containing information about the elevator.

```
#include <stdio.h>
#include <stdbool.h>
```

Classes

- struct Queue

  Contains the queue list for handling orders.
- struct Sensors

  Contains the sensor information for the elevator. Useful for debugging in case of hardware problems.
- struct ElevatorSM

  Structure containing the essential information about the elevator that the elevator itself cannot provide (since the current floor can be provided by the driver, it is not included here.)

Macros

- #define **FLOORS** 4
- #define **BUTTONS** 3
- #define **MAX_ORDERS** 5

Enumerations

- enum ElevatorState { state_stop , state_move , state_deliver , state_still }

  Enumeration of elevator states.
- enum StateEvent { event_enter , event_execute , event_exit }

  Enumeration of the different stages (substates) of each state.

4.4.1 Detailed Description

Defines neccessary structs and enums containing information about the elevator.

4.4.2 Enumeration Type Documentation

4.4.2.1 ElevatorState

`enum ElevatorState`

Enumeration of elevator states.

Enumerator

state_stop	The stop button is pushed in
state_move	The elevator is moving towards an order
state_deliver	The elevator stops at the floor to serive the order
state_still	The elevator is awaing an order

4.4.2.2 StateEvent

`enum StateEvent`

Enumeration of the different stages (substates) of each state.

Enumerator

event_enter	Mechanisms while entering a state ("one-time" things)
event_execute	Execute state event is repeatedly updated to continue carrying out the states actions.
event_exit	The exit procedure for the state.

4.5 elevator_types.h

Go to the documentation of this file.

```
00001
00006
00007 #pragma once
00008 #include <stdio.h>
00009 #include <stdbool.h>
00010
00011 #define FLOORS 4
00012 #define BUTTONS 3
00013 #define MAX_ORDERS 5
00014
00019 typedef struct {
00020     int queue_count;
00021     int queue_list[MAX_ORDERS];
00022 } Queue;
00023
00028 typedef struct {
00029     bool sensor_stop_button;
00030     int sensor_current_floor;
00031     bool sensor_obstacle;
00032     bool sensor_button_values[FLOORS][BUTTONS];
00033 } Sensors;
00034
00039 typedef enum {
00040     state_stop,
00041     state_move,
00042     state_deliver,
00043     state_still
00044 } ElevatorState;
00045
00050 typedef enum {
00051     event_enter,
00052     event_execute,
00053     event_exit
00054 } StateEvent;
00055
00062 typedef struct{
00063     ElevatorState current_state;
00064     int target_floor;
00065     int elevator_direction;
00066     int last_current_floor;
00067     Sensors sensors;
00068     Queue queue;
00069     int door_opened_at_stop;
00070 } ElevatorSM;
00071
```

4.6 source/lights.h File Reference

A module to be used to handle various light functionality.

```
#include "elevator_types.h"
#include "driver.h"
#include <stdio.h>
#include <stdbool.h>
```

Functions

- void lights_last_floor_lamp (ElevatorSM *sm)

  Moves the elevator to the target floor, making sure to only move when movement is permitted.
- void lights_turn_on (int floor, int button)

  Turns on the correct order light when a button is pressed.
- void lights_turn_off (int floor)

  Turns off all order lights belonging to a give floor. To be used when order is completed on said floor.
- void **lights_turn_off_all_orders** ()

  Turns off all order lights. To be used for initilsation and if the stop button is pressed.

4.6.1 Detailed Description

A module to be used to handle various light functionality.

4.6.2 Function Documentation

4.6.2.1 lights_last_floor_lamp()

```
void lights_last_floor_lamp (
            ElevatorSM * sm)
```

Moves the elevator to the target floor, making sure to only move when movement is permitted.

Parameters

sm	A pointer to the Elevator's state machine. The function needs access to this to update the elevator_direction and access the target_floor.

Returns

bool Returns true when the elevator has reached the target floor.

4.6.2.2 lights_turn_off()

```
void lights_turn_off (
            int floor)
```

Turns off all order lights belonging to a give floor. To be used when order is completed on said floor.

Parameters

floor	The floor in which lights should be turned off.

4.6.2.3 lights_turn_on()

```
void lights_turn_on (
            int floor,
            int button)
```

Turns on the correct order light when a button is pressed.

Parameters

floor	The floor at which the order is placed
button	The type of button pressed (up, down, or cabin).

4.7 lights.h

Go to the documentation of this file.

```
00001
00006 #pragma once
00007
00008 #include "elevator_types.h"
00009 #include "driver.h"
00010 #include <stdio.h>
00011 #include <stdbool.h>
00012
00021 void lights_last_floor_lamp(ElevatorSM *sm);
00022
00029 void lights_turn_on(int floor, int button);
00030
00037 void lights_turn_off(int floor);
00038
00043 void lights_turn_off_all_orders();
```

4.8 source/movement.h File Reference

A module to be used for handlinging the movement of the elevator, particularly in servicing an order.

```
#include "elevator_types.h"
#include "driver.h"
#include <stdio.h>
#include <stdbool.h>
```

Functions

- ElevatorState movement_get_to_floor (ElevatorSM *sm)

  Moves the elevator to the target floor, making sure to only move when movement is permitted.
- int movement_choose_direction (ElevatorSM *sm)

  Decides the direction the elevator should move based on the current floor and the target floor.

4.8.1 Detailed Description

A module to be used for handlinging the movement of the elevator, particularly in servicing an order.

4.8.2 Function Documentation

4.8.2.1 movement_choose_direction()

```
int movement_choose_direction (
            ElevatorSM * sm)
```

Decides the direction the elevator should move based on the current floor and the target floor.

Parameters

sm	A pointer to the Elevator's state machine. The function needs access to this to update the elevator_direction and access the target_floor.

Returns

int Returns the chosen direction, either -1 (down), 0 (stop), or 1 (up).

4.8.2.2 movement_get_to_floor()

```
ElevatorState movement_get_to_floor (
            ElevatorSM * sm)
```

Moves the elevator to the target floor, making sure to only move when movement is permitted.

Parameters

sm	A pointer to the Elevator's state machine. The function needs access to this to update the elevator_direction and access the target_floor.

Returns

ElevatorState Returns state_move if the elevator isn't at the destination yet, and state_deliver when it is.

4.9 movement.h

Go to the documentation of this file.

```
00001
00006 #pragma once
00007
00008 #include "elevator_types.h"
00009 #include "driver.h"
00010 #include <stdio.h>
00011 #include <stdbool.h>
00012
00022 ElevatorState movement_get_to_floor(ElevatorSM *sm);
00023
00032 int movement_choose_direction(ElevatorSM *sm);
00033
00034
```

4.10 source/orders.h File Reference

A module that registers orders.

```
#include "sensors.h"
#include "queue.h"
#include "driver.h"
#include "lights.h"
#include <stdio.h>
#include <stdbool.h>
```

Functions

- void orders_register_order (ElevatorSM ∗sm)

  Iterates through the nested vector contaning the order button information and registers orders if they are placed.

4.10.1 Detailed Description

A module that registers orders.

4.10.2 Function Documentation

4.10.2.1 orders_register_order()

```
void orders_register_order (
            ElevatorSM * sm)
```

Iterates through the nested vector contaning the order button information and registers orders if they are placed.

Parameters

∗*sm*	A pointer to elevator state machine. Needed to give access to sensor information.

4.11 orders.h

Go to the documentation of this file.

```
00001
00005
00006 #pragma once
00007 #include "sensors.h"
00008 #include "queue.h"
00009 #include "driver.h"
00010 #include "lights.h"
00011 #include <stdio.h>
00012 #include <stdbool.h>
00013
00021 void orders_register_order(ElevatorSM *sm);
```

4.12 source/queue.h File Reference

A module containing the queue functionality for handling and prioritising orders.

```
#include "elevator_types.h"
#include "driver.h"
#include <stdio.h>
#include <stdbool.h>
```

Macros

- #define **NO_ORDER** -1

Functions

- void queue_init (Queue *queue)
  Initialize the queue.
- int queue_find_floor (Queue *queue, int floor)
  Check if a floor is already in the queue.
- void queue_add (ElevatorSM *sm, int new_order, int button)
  Add an order to the queue with proper prioritization.
- int queue_remove (Queue *queue, int floor)
  Remove a specific floor from the queue.
- void queue_empty (Queue *queue)
  Empty the entire queue.
- int queue_peek (Queue *queue)
  Get the next order from the queue without removing it.
- void queue_print (Queue *queue)
  Debug function to print the current queue state.

4.12.1 Detailed Description

A module containing the queue functionality for handling and prioritising orders.

4.12.2 Function Documentation

4.12.2.1 queue_add()

```
void queue_add (
            ElevatorSM * sm,
            int new_order,
            int button)
```

Add an order to the queue with proper prioritization.

Parameters

sm	Pointer to the ElevatorSM structure
new_order	Floor number of the new order
button	Button type: 0=up, 1=down, 2=cabin

Priorities:

1. If elevator is moving and a new order from outside is in the same direction and within the current range of motion, it becomes the next order.
2. All other orders go to the end of the queue
3. Duplicate floor orders are ignored, unless they meet priority 1, in which case they are moved.

4.12.2.2 queue_empty()

```
void queue_empty (
            Queue * queue)
```

Empty the entire queue.

Parameters

queue	Pointer to the Queue structure

4.12.2.3 queue_find_floor()

```
int queue_find_floor (
            Queue * queue,
            int floor)
```

Check if a floor is already in the queue.

Parameters

queue	Pointer to the Queue structure
floor	Floor number to find

Returns

Index of the floor in the queue if found, -1 otherwise

4.12.2.4 queue_init()

```
void queue_init (
            Queue * queue)
```

Initialize the queue.

Parameters

queue	Pointer to the queue instance to initialise.

4.12.2.5 queue_peek()

```
int queue_peek (
            Queue * queue)
```

Get the next order from the queue without removing it.

Parameters

queue	Pointer to the Queue structure

Returns

Next floor in queue or NO_ORDER if queue is empty

4.12.2.6 queue_print()

```
void queue_print (
            Queue * queue)
```

Debug function to print the current queue state.

Parameters

queue	Pointer to the Queue structure

4.12.2.7 queue_remove()

```
int queue_remove (
            Queue * queue,
            int floor)
```

Remove a specific floor from the queue.

Parameters

queue	Pointer to the Queue structure
floor	Floor number to remove

Returns

1 if successful, 0 if floor not found

4.13 queue.h

Go to the documentation of this file.

```
00001
00006
00007 #pragma once
00008
00009 #include "elevator_types.h"
00010 #include "driver.h"
00011 #include <stdio.h>
00012 #include <stdbool.h>
00013
00014 #define NO_ORDER -1
00015
00021 void queue_init(Queue *queue);
00022
00030 int queue_find_floor(Queue *queue, int floor);
00031
00045 void queue_add(ElevatorSM *sm, int new_order, int button);
00046
00054 int queue_remove(Queue *queue, int floor);
00055
00061 void queue_empty(Queue *queue);
00062
00069 int queue_peek(Queue *queue);
00070
00076 void queue_print(Queue *queue);
```

4.14 source/sensors.h File Reference

A module that processes and updates sensor information.

```
#include <stdbool.h>
#include <stdio.h>
#include "elevator_types.h"
#include "driver.h"
```

Functions

- void sensors_update (Sensors *sensors)

  Iterates through all order button values. Neccessary for regestering orders. Updates sensor information for the stop button and floor sensor as well.
- bool sensors_is_button_pressed (Sensors *sensors, int floor, int button)

  Used to know if a button in the sensors vector is pressed. Useful for regestering orders.
- int sensors_get_current_floor (Sensors *sensors)

  Used to get sensor information about the current floor.

4.14.1 Detailed Description

A module that processes and updates sensor information.

4.14.2 Function Documentation

4.14.2.1 sensors_get_current_floor()

```
int sensors_get_current_floor (
            Sensors * sensors)
```

Used to get sensor information about the current floor.

Parameters

*sensors	A pointer to the sensors struct instance for the elevator state machine.

Returns

int The current floor.

4.14.2.2 sensors_is_button_pressed()

```
bool sensors_is_button_pressed (
            Sensors * sensors,
            int floor,
            int button)
```

Used to know if a button in the sensors vector is pressed. Useful for regestering orders.

Parameters

*sensors	A pointer to the sensors struct instance for the elevator state machine.
floor	The floor to be checked.
button	The type of button to be checked.

Returns

bool Return true (1) if the button is pressed, and false (0) if not.

4.14.2.3 sensors_update()

```
void sensors_update (
            Sensors * sensors)
```

Iterates through all order button values. Neccessary for regestering orders. Updates sensor information for the stop button and floor sensor as well.

The function is continuously called and this nested vector containing the order button information updated. This nested vector is then used in processing orders.

Parameters

∗sensors	A pointer to the sensors struct instance for the elevator state machine. The function needs access to this to update the sensors information in the instance of the elevator state machine.

4.15 sensors.h

Go to the documentation of this file.

```
00001
00005
00006 #pragma once
00007 #include <stdbool.h>
00008 #include <stdio.h>
00009 #include "elevator_types.h"
00010 #include "driver.h"
00011
00022 void sensors_update(Sensors *sensors);
00023
00033 bool sensors_is_button_pressed(Sensors *sensors, int floor, int button);
00034
00041 int sensors_get_current_floor(Sensors *sensors);
```

4.16 source/start.h File Reference

A module that initialises and sustain the elevator state machine.

```
#include "elevator_types.h"
#include "TSM.h"
#include "driver.h"
#include "queue.h"
#include "lights.h"
#include <stdbool.h>
#include <stdio.h>
```

Functions

- bool start_find_floor ()

  Ensures the elevator enters a defined state before the state machine is initialised. The elevator moves down until it reaches a floor.

- void **start_init** ()

  Instansiates the elevator state machine and updates the state machine in an endless while loop.

4.16.1 Detailed Description

A module that initialises and sustain the elevator state machine.

4.16.2 Function Documentation

4.16.2.1 start_find_floor()

```
bool start_find_floor ()
```

Ensures the elevator enters a defined state before the state machine is initialised. The elevator moves down until it reaches a floor.

Returns

bool Returns true when the elevator is in a defined state (at a floor).

4.17 start.h

Go to the documentation of this file.

```
00001
00005
00006 #pragma once
00007
00008 #include "elevator_types.h"
00009 #include "TSM.h"
00010 #include "driver.h"
00011 #include "queue.h"
00012 #include "lights.h"
00013 #include <stdbool.h>
00014 #include <stdio.h>
00015
00024 bool start_find_floor();
00025
00030 void start_init();
00031
```

4.18 source/TSM.h File Reference

Main functionality for the elevator state machine. Handles updating and switching between states.

```
#include <stdbool.h>
#include <stdio.h>
#include "elevator_types.h"
#include "driver.h"
#include "movement.h"
#include "door.h"
#include "sensors.h"
#include "orders.h"
```

Functions

- ElevatorState TSM_state_stop (ElevatorSM ∗sm, StateEvent event)

  Handles state actions for the stop state. The elevator is in the stop state when the stop button is held in.
- ElevatorState TSM_state_move (ElevatorSM ∗sm, StateEvent event)

  Handles state actions for the move state. The elevator is in the move state when moving towards an order.
- ElevatorState TSM_state_deliver (ElevatorSM ∗sm, StateEvent event)

  Handles state actions for the deliver state. The elevator is in the deliver state when servicing an order at a floor.
- ElevatorState TSM_state_still (ElevatorSM ∗sm, StateEvent event)

  Handles state actions for the still state. The elevator is in the still state when there are no orders to be serviced.
- void TSM_call_exit (ElevatorSM ∗sm, ElevatorState state)

  Executes the exit action for a given state.
- void TSM_call_enter (ElevatorSM ∗sm, ElevatorState state)

  Executes the enter action for a given state.
- ElevatorState TSM_update (ElevatorSM ∗sm)

  Updates the state machine.

4.18.1 Detailed Description

Main functionality for the elevator state machine. Handles updating and switching between states.

4.18.2 Function Documentation

4.18.2.1 TSM_call_enter()

```
void TSM_call_enter (
            ElevatorSM * sm,
            ElevatorState state)
```

Executes the enter action for a given state.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs to be able to pass sm to the functions it calls.
state	The state for which to call the enter function.

4.18.2.2 TSM_call_exit()

```
void TSM_call_exit (
            ElevatorSM * sm,
            ElevatorState state)
```

Executes the exit action for a given state.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs to be able to pass sm to the functions it calls.
state	The state for which to call the exit function.

4.18.2.3 TSM_state_deliver()

```
ElevatorState TSM_state_deliver (
            ElevatorSM * sm,
            StateEvent event)
```

Handles state actions for the deliver state. The elevator is in the deliver state when servicing an order at a floor.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs access to be able to change member varaibles of sm.
event	The state event to process.

Returns

ElevatorState The new state after processing the event.

4.18.2.4 TSM_state_move()

```
ElevatorState TSM_state_move (
            ElevatorSM * sm,
            StateEvent event)
```

Handles state actions for the move state. The elevator is in the move state when moving towards an order.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs access to be able to change member varaibles of sm.
event	The state event to process.

Returns

ElevatorState The new state after processing the event.

4.18.2.5 TSM_state_still()

```
ElevatorState TSM_state_still (
            ElevatorSM * sm,
            StateEvent event)
```

Handles state actions for the still state. The elevator is in the still state when there are no orders to be serviced.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs access to be able to change member varaibles of sm.
event	The state event to process.

Returns

ElevatorState The new state after processing the event.

4.18.2.6 TSM_state_stop()

```
ElevatorState TSM_state_stop (
            ElevatorSM * sm,
            StateEvent event)
```

Handles state actions for the stop state. The elevator is in the stop state when the stop button is held in.

Parameters

sm	A pointer to the ElevatorSM variable sm. The function needs access to be able to change member varaibles of sm.
event	The state event to process.

Returns

ElevatorState The new state after processing the event.

4.18.2.7 TSM_update()

```
ElevatorState TSM_update (
            ElevatorSM * sm)
```

Updates the state machine.

Dispatches the current state's execute event and handles transitions by calling exit on the old state and enter on the new state. Stop-events are also first priority, and if the stop button is pressed in, it it handeled immediately.

Parameters

sm	Pointer to the ElevatorSM instance.

Returns

ElevatorState The new state after the update.

4.19 TSM.h

Go to the documentation of this file.

```
00001
00006
00007 #pragma once
00008
00009 #include <stdbool.h>
00010 #include <stdio.h>
00011
00012 #include "elevator_types.h"
00013 #include "driver.h"
00014 #include "movement.h"
00015 #include "door.h"
00016 #include "sensors.h"
00017 #include "orders.h"
00018
00019
00029 ElevatorState TSM_state_stop(ElevatorSM *sm, StateEvent event);
00030
00040 ElevatorState TSM_state_move(ElevatorSM *sm, StateEvent event);
00041
00051 ElevatorState TSM_state_deliver(ElevatorSM *sm, StateEvent event);
00052
00062 ElevatorState TSM_state_still(ElevatorSM *sm, StateEvent event);
00063
00071 void TSM_call_exit(ElevatorSM *sm, ElevatorState state);
00072
00080 void TSM_call_enter(ElevatorSM *sm, ElevatorState state);
00081
00092 ElevatorState TSM_update(ElevatorSM *sm);
00093
00094
00095
```