TTK4235 - Heisprosjekt (Lab 2)

Gruppe 53
Ole Sørensen Dalsgård, Marius Steen Rasmussen

19. mars 2024

Innhold

Sammendrag

Et heisprosjekt utført i forbindelse med emnet TTK4235 Tilpassede datasystemer [1]. Factory acceptance test (FAT-Test) av heissystem med etasje- og heispanel der muligheter for bestilling opp/ned med etasjelys samt stopp og obstruksjonsbryter var utført. Rapporten formidler metoden som ble brukt og resultatet av den. Det avsluttes med en drøfting over effekten av bruken av V-modellen og UML diagram i utviklingsløpet.

1 Introduksjon

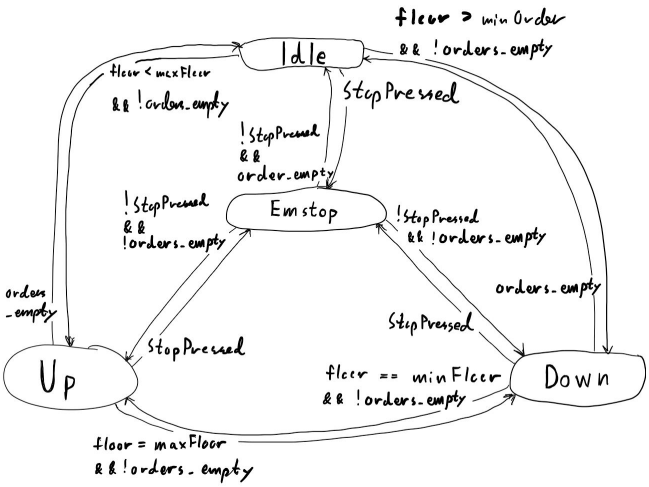
Denne rapporten skrives i forbindelse med heisprosjektet [2], tilknyttet emnet TTK4235 Tilpassede datasystemer [1]. Målet med laben er å få innblikk i hvordan V-modellen og UML diagram kan benyttes for å gjennomføre et kodeprosjekt for en heis. Det stilles ulike krav til hvordan heisen skal oppføre seg i ulike situasjoner eller tester i en såkalt "Factory acceptance test" (FAT-test).

Vi skulle lage kode til en fysisk modell av en heis. Programmet hadde forhåndsdefinerte funksjoner for blant annet hvilken retning motoren skulle ha, hva en sensor målte, hvilket lys som skulle være på osv. Det var disse vi tok utgangspunkt i da vi skrev koden.

2 Metoder

For heisen brukte vi den generelle *heisalgoritmen* [4] for normal drift, samt algoritmer for håndtering av nødstopp. For normal drift er det tenkt at heisen fortsetter i en retning helt til alle bestillingene i den retningen er betjent. Hvis alle bestillingene i retningen er betjent kan den velge å bli inaktiv. Alternativt velger heisen å gå i motsatt retning, gitt at det er registrert en bestilling der. Hvis stoppknappen blir trykket under denne normale driften skal heisen stoppe momentant, uavhengig av hvor den er.

For å sette algoritmen til verks laget vi et system med fire tilstander: hvorav *up*, *down* og *idle*. Hovedpunktet med retningstilstandene er at hver gang heisen er i bevegelse og støter på en etasje skal den bestemme om den skal stoppe eller ikke. Disse tilstandene har innebygd funksjonalitet for å midlertidig stoppe ved en etasje. Men er man ved siste bestilling i retningen kan man initiere et mer langvarig stopp ved å gå over i tilstanden *idle*. I samsvar med *heisalgoritmen* kan man også da velge å bytte til den andre retningstilstanden. Siden *up* og *down* er svært like har vi valgt å bare inkludere diagrammet for *up*



Figur 1: Overordnet og forenklet tilstandsdiagram over heissystemet.

Den fjerde tilstanden ga vi navnet *emstop*, forkortelse for *emergency stop*. Man havner i denne tilstanden hvis man når som helst trykker på stoppknappen. Dette kan hende når heisen befinner seg mellom to etasjer eller ved en etasje. Uansett vil heisen stoppe. Etter at den har kommet til ro og stoppknappen er sluppet vil den sjekke for bestillinger. En ny bestilling vil føre til at tilstanden endres til *up*, *down* og *idle* alt etter som.

Implementasjonen vår baserte seg på fullstendig på *polling*. Ideen er at systemet sjekker og oppdaterer regelmessig forhåndslagede variabler. En endring av en sensorverdi eller et knappetrykk kan føre til endringer av disse variablene. Dette kan igjen føre til at en betingelse blir oppfylt og programmet vil endre tilstand. Vi valgte dette fremfor *interrupts* først og fremst på grunn av enkelheten. Ulempen med polling er ineffektiviteten men vi forutsa at det tilpassede datasystemet ikke krevde en høyst effektiv kode.

For å best mulig endre variablene i tråd med de eksterne tilstandene brukte vi pekere. Å lage funksjoner som tar inn pekere tillater oss å endre variablene direkte hvor de er laget. Alternativet er pass-by-value, som forutsetter bedre kontroll på scope. I implementasjonen har vi brukt mange while-løkker, ofte nøstet. Det blir dermed kronglete å bruke pass-by-value. Forøvrig har pass-by-pointer den fordel at man ikke hele tiden lager kopier som sparer minne.

Designet vårt tar hensyn til oppstartskravene gitt av heisspesifikasjonen. Helt i begynnelsen av koden kaller vi på funksjonen GoUpToClosest. Som navnet tilsier setter funksjonen motorretningen til oppover og setter den ikke til idle før floor_sensor viser noe annet en -1. Dette er tilstrekkelig siden vi ikke trenger å ta høyde for at heisen starter over 4. etasje. Og mens GoUpToClosest kjører blir ikke noe lagt til i bestillingslisten: *OrderArray*.

OrderArray er en char array. Posisjonen til tegnene i arrayen bestemmer hvilken etasje det er snakk om. Definisjonene på tegnene og et eksempel på en bestillingsarray er vist under. I eksempelet har vi ingen bestilling til 1. etasje, en cab bestilling til 2. etasje osv...

Bestillingsarrayen legger opp til at vi kan oppfylle H2 punktet i heisspesifikasjonen. Dersom heisen er på vei i en retning og kommer over en etasje som ikke er en endestasjon vil den den boolske variabelen floor_stop bli oppdatert av funksjonen UpdateFloorStop. Se Figur 5c og 3. Heisen vil stoppe på etasjen hvis det tilhørende tegnet fra *OrderArray* er 'C'. Ellers hvis vi har 'U' eller 'N' vil den sjekke om dette stemmer overens med retningen.

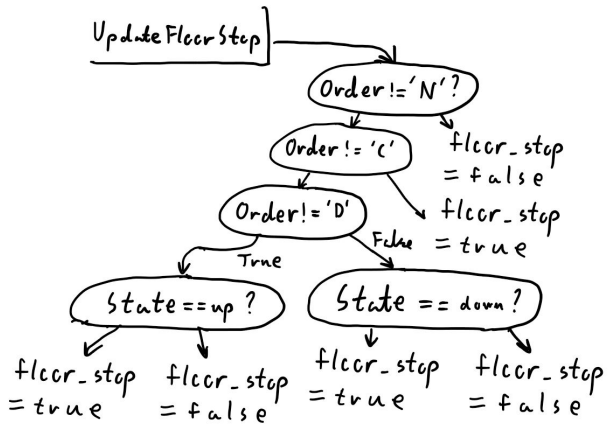
Under normale tilstander sjekker systemet svært ofte etter bestillinger. Funksjonen *AddOrders* sørger for å oppdatere *OrderArray* med bestillinger fra etasje- og heispanelet. *AddOrders* blir svært hyppig kalt på (normalt hvert 10 ms) . Dette forsikrer oss om at alle bestillinger blir lagt til i *OrderArray*.

Implementasjonen håndterer altså godt mottaket av bestillinger, så hvis den også håndterer sletting rett blir punkt H1 og H3 oppfylt. Vi designet det slik at under normal drift er *DeleteOrderWithSensor* den eneste som kan fjerne en bestilling fra *OrderArray*. Funksjonen *DeleteOrderWithSensor* blir kalt på i begynnelsen av den indre tilstanden *StopAndLight*, se Figur 5b. *StopAndLight* er den delen av koden som kjører når man kommer til en etasje. Da slettes bestillingen til etasjen som er indikert av *floor_sensor*.

- 'D'- En bestilling fra etasjepanelet nedover
- 'U'- En bestilling fra etasjepanelet oppover
- 'C'- En bestilling fra heispanelet
- 'N'- Ingen bestilling til etasjen

```
Eksempel: {'N', 'C','U','D'}
```

Figur 2: Hvordan *OrderArray* er bygd opp



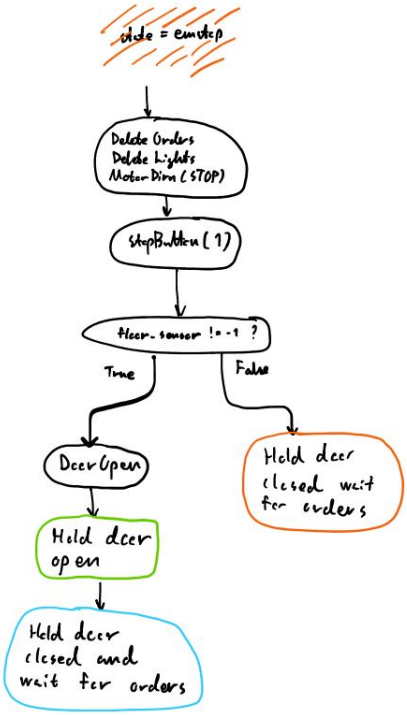
Figur 3: Diagram over funksjonen UpdateFloorStop

Vår definisjon av *OrderArray* muliggjør også at heisen står stille når det ikke er noen ubetjente bestillinger. Vi definerte den boolske variabelen *orders_empty*. Variabelen blir oppdatert etter hver gang *AddOrders* blir kallt på. Hvis *OrderArray* da bare inneholder 'N' vil *orders_empty* bli true, false ellers. Som konsekvens av at *orders_empty*=true vil heisen forbli i *idle* hvis den først er der. Det samme punktet gjelder for *emstop*. I tillegg bytter man til *idle* fra retningstilstandene dersom det ikke finnes bestillinger nedover eller oppover.

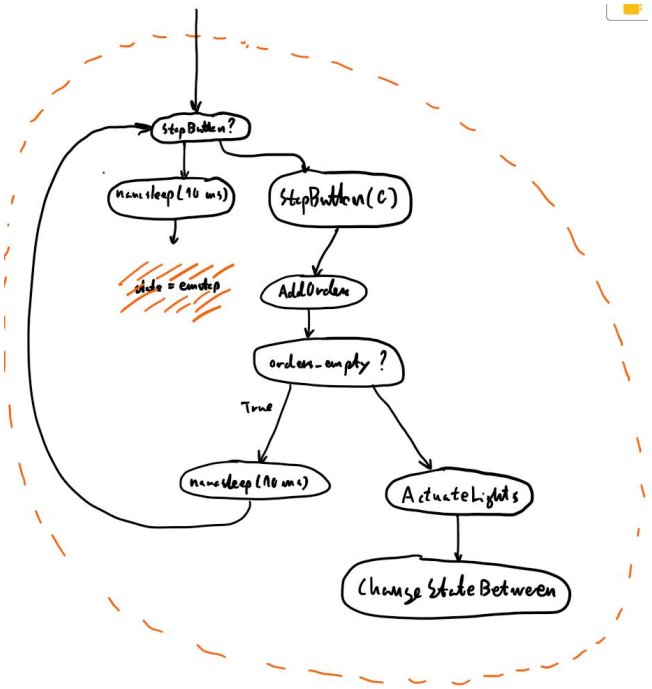
Implementasjonen tar også hensyn til sikkerhet. I koden blir stoppknappen sjekket hele tiden (cirka hvert 10 ms). Hvis den blir trykket vil man uansett havne i emstop eller tilbake til emstop, hvis man allerede er der. Ved inngangen til emstop slettes alt av bestillinger og tilhørende lys. Se Figur 4a. Så lenge knappen er trykket vil man kalle StopButton(1), men med en gang den blir sluppet kalles StopButton(0).

For åpning av døren lagde vi et timersystem. Vi initierte en double verdi kalt Time med startverdien 3.0, for 3 sekund. Deretter skrudde vi på lyset for door_open. Så begynnte timeren: hvis man ikke oppdaget en obstruksjon skulle man trekke fra 0.01 fra Time og nanosleepe i 10 ms. Hvis ikke skulle Time konstant oppdateres til 3. Dette medfører at døren ikke kan lukkes så lenge det er en obstruksjon. Hvis obstruksjonen fjerner seg vil det ta ytterligere 3 sekund før man kan forlate løkken og lukke døren. Se Figur 5b og 4c.

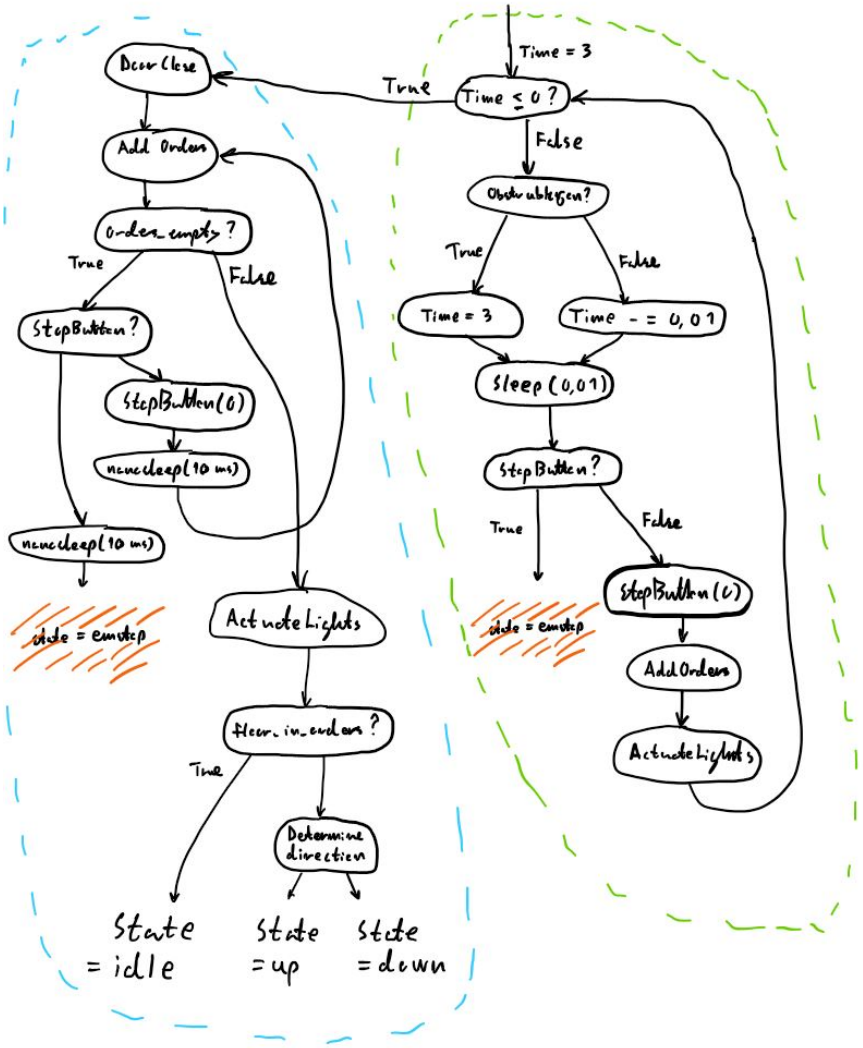
For at heisen skulle huske posisjonen sin ved nødstopp lagde vi *posArray* med tilhørende funksjoner. Dette var en heltalls array bestående av 7 elementer; 4 for etasjene og 3 for mellomrommene mellom dem. Vi utarbeidet en funksjon som oppdaterte denne arrayen. Den tok utgangspunkt i målingen fra floor_sensor, retningstilstanden og posisjonen. For eksempel hvis heisen er ferdig i 3.etasje og går nedover så vil man oppdatere posisjonsen dersom floor_sensor går over til -1. Da vet man at er mellom 3. og 2. etasje.



(a) Forenklet diagram over *emstop*

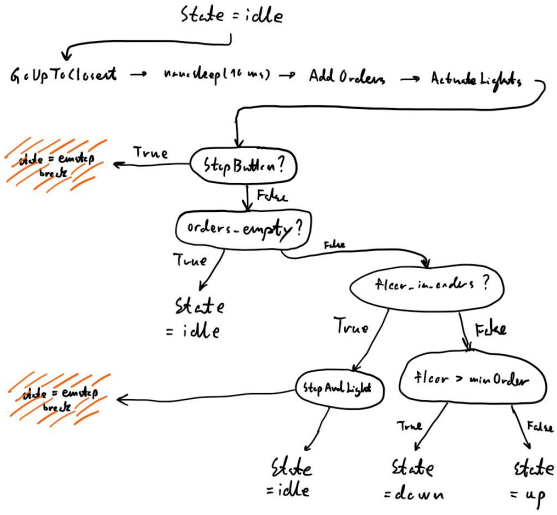


(b) Den aktive delen av *emstop* når heisen blir stoppet mellom etasjer

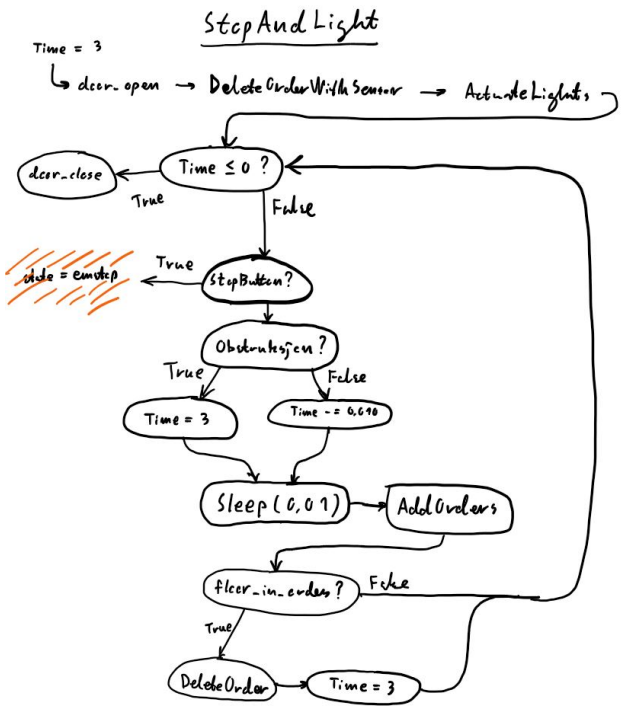


(c) Den aktive delen av *emstop* når heisen blir stoppet ved en etasje

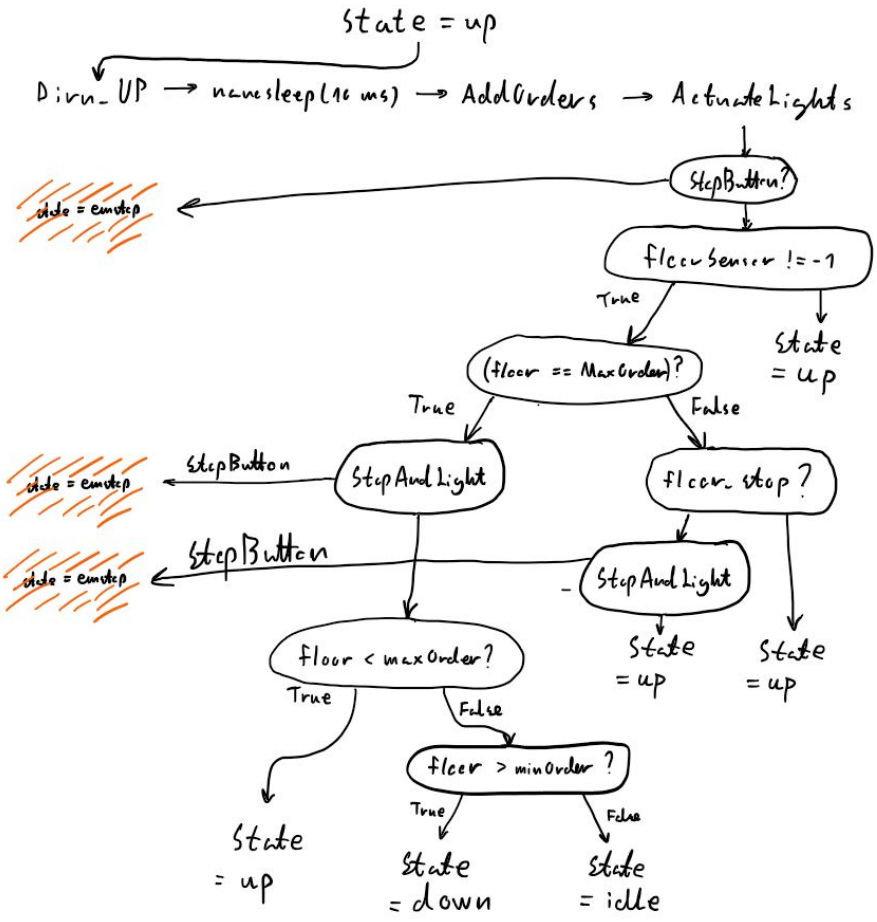
Figur 4: Emstop



(a) Diagram over tilstanden *idle*



(b) Diagram over den indre tilstanden *StopAndLight*



(c) Diagram over tilstanden *up*

Figur 5: Idle, up og StopAndLight

3 Resultater

Systemet vårt oppfylte alle punktene på Fat-testen utenom L3. Når en bestilling kom inn mens den delen av programmet som kjørte var i StopAndLight ble ikke bestillingslysene tent. Bestillingene ble derimot betjent som vanlig.

4 Konklusjon

Vi tror at grunnen til at F3 ikke ble oppfylt ligger i en feil i den indre tilstanden *StopAndLight*, se Figur 5b. For at bestillingene i bestillingsarrayen skal bli synlige la vi opp til at vi eksplisitt måtte kalle på funksjonen *ActuateLights*. Bestillinger blir derimot lagt til i arrayen, som forklarer hvorfor heisen oppførte seg normalt selv om bestillingene ikke ble tatt. Den beste løsningen hadde vært å bare kalt på *ActuateLights*. Eventuelt kunne man ha innbakt *ActuateLights* i AddOrders, men må da passe på at det ikke fører til uønsket oppførsel andre steder i koden.

5 Refleksjon

5.1 UML og V-modellen

I startfasen har UML og V-modellen hjulpet med å gå fra en mer konseptuell fase til noe mer konkret. Problemstillingen og dens utfordringer har altså blitt konkretisert ved hjelp av UML og V-modellen. For midtfasen gir UML diagram bevissthet rundt hvilke funksjoner som er implementert og hvilke som mangler. Ved sluttfasen har bruk av UML diagram hjulpet med å følge logikk av det en har implementert.

Den delen av designet som ble endret på underveis var *OrderArray*. Årsaken for endringen var at den ikke tok hensyn til retning av bestillinger fra etasjepanelet.

Prosjektet kunne blitt lettere med tanke på tidsforbruk siden diagrammene kun er en gjenspeiling av den aktuelle koden. På den andre siden vil debugging være mer utfordrende som følge av mangelfull oversikt over implementasjonen. Det er mulig at prosjektet hadde blitt lettere med en annen metode, men bruk av UML og V-modellen fungerer godt for denne typen prosjekter. For andre typer prosjekter kan alternative modeller være foretrukket. Lærdommen er å benytte modeller som er godt egnet for det aktuelle prosjektet siden alle modeller har sine styrker og svakheter.

5.2 KI

I dette prosjektet har KI blitt brukt for å verifisere kodeimplementasjonens syntaks og få pekepinn på om det er noen yttertilfeller. [3]

Bruk av KI har hatt positive effekter ved at arbeidet er blitt effektivisert og hjulpet med å skape bedre forståelse av utfordrende temaer i C programmering. Noen av de utfordrende temaene er syntaks og virkemåte rundt pekere og mer.

5.3 Robusthet, skalerbarhet og vedlikehold

Bruk av UML og V-modellen har bidratt positivt for fremming av samtlige egenskaper. UML diagrammer gir god oversikt over hvilken del av programmet som kjører. Dette hjelper under implementasjon for å sikre robusthet.

Programmet vil ikke nødvendigvis være mer skalerbart, men UML diagrammet kan hjelpe med å gi et godt og oversiktlig utgangspunkt. Dermed vil en implementasjon med godt fundament potensielt gjøre at den er mer skalerbar.

Når vedlikehold knyttes opp mot UML diagrammer og V-modellen er det nyttig å ha UML diagrammer for å få god oversikt over hvilken del av programmet som trenger vedlikehold. Eksempelvis kan UML diagrammer illustrere ulike avhengigheter i implementasjonen på en mer oversiktlig måte

enn å måtte sette seg inn i funksjonene kun ved hjelp av kildekode. Her kan selvfølgelig gode kommentarer i kildekode fungere som substitutt til en viss grad, men for store programmer kan UML diagram definitivt forenkle vedlikeholdsarbeidet.

Referanser

[1] Martin Føre. *TTK4235 - Tilpassede datasystemer*. `https://www.ntnu.no/studier/emner/TTK4235`. 2024.

[2] Terje Haugland Jacobsson og Tord Natlandsmyr. *TTK4235 Lab 2 - Heisprosjektet*. `https://github.com/ITK-TTK4235/lab_2`. 2024.

[3] OpenAI. *ChatGPT*. `https://chat.openai.com/`. Accessed on March 19, 2024. 2022.

[4] Wikipedia. *Elevator algorithm*. `https://en.wikipedia.org/wiki/Elevator%23Elevator_algorithm`. Online; accessed 12-March-2024. 2024.