



Revisjonshistorie

År	Forfatter
2014	Øyvind Stavdahl
2014	Anders Rønning Petersen
2016	Øyvind Stavdahl
2016	Konstanze Kölle
2017	Ragnar Ranøyen Homb
2017	Bjørn-Olav Holtung Eriksen
2020	Kolbjørn Austreng
2021 - 2023	Kiet Tuan Hoang

I Introduksjon - Praktisk rundt filene

I denne laben får dere utlevert noen `.c` og `.h`-filer under `skeleton_project`-mappen. Tabellen under lister opp alle filene som kommer med i `skeleton_project`-mappen samt litt informasjon om dere skal endre på filene eller om dere skal la dem bli i løpet av øvingen. I kontekst av tabellen under, så betyr *helst ikke* at dere kan endre på filene, men at dette ikke burde gjøres siden det kan føre til andre komplikasjoner videre på veien. For eksempel: om det trengs mer kompliserte funksjoner enn de som allerede er definert i `elevio`, så anbefales det å opprette filer som bruker de allerede definerte funksjonene i `elevio` istedenfor å endre direkte på `elevio`-funksjonene.

Filer	Skal denne filen endres?
<code>skeleton_project/source/main.c</code>	ja
<code>skeleton_project/source/driver/elevio.c</code>	helst ikke
<code>skeleton_project/source/driver/elevio.h</code>	helst ikke
<code>skeleton_project/source/driver/con_load.h</code>	nei
<code>skeleton_project/source/driver/elevio.con</code>	nei

II Introduksjon - Praktisk rundt laben

Programmeringsspråket C er et kraftig verktøy som regelmessig benyttes i et bredt spekter industri-sammenhenger, særlig i sanntidsapplikasjoner og maskinnær programvare. Denne laben går ut på å benytte C til å implementere et styresystem for en fysisk heis som styres gjennom en Arduino. I tillegg til selve implementasjonen, skal systemet beskrives og dokumenteres med UML før dere starter med selve implementasjonen.

Målet for laben er å gi praktisk erfaring med utvikling av et system med definerte krav til oppførsel, ved å benytte UML og C som verktøy. For å strukturere arbeidet, og sikre verifikasjon av akseptkriterier, skal den pragmatiske V-modellen benyttes (se appendiks A). Disse verktøyene skal brukes på en fysisk heis-modell på sal (se introduksjon III).

Godkjenning

Heis-laben vil ikke telle på sluttkarakteren i år, men må godkjennes for at dere skal kunne gå opp til eksamen. Prosjektet er konseptuelt delt i 3 deler (oppgaver), som i utgangspunktet må godkjennes:

- UML-del med klasse-, sekvens og tilstands-diagrammer. Dere skal her strukturere heis-prosjektet med ulike diagrammer for å beskrive implementasjonen før den blir skrevet.
- Implementasjons-del med en FAT (**F**actory **A**cceptance **T**est) (se appendiks B). Dette er for å se at implementasjonen faktisk oppfører seg som en heis.
- Refleksjon rundt egen implementasjon og hvordan det har vært å bruke UML og V-modellen, og hvordan bruken av disse har påvirket implementasjonen. Se appendiks C for kommentar på de viktigste punktene.

Viktige datoer

Som man kan se fra tabell 1, så må UML bli godkjent i eller før uke 8. Neste blir FAT-en som utføres i uke 10. Dette vil foregå på sanntidssalen hvor tidspunkt avhenger av når dere har saltid. Det eneste dere trenger å gjøre er å vente på arbeidsplassene deres, så vil studass eller vitass komme til arbeidsplassene deres og utføre FAT-en med styresystemet deres. Refleksjon skjer uken etter, og dere kan få godkjent ved å snakke med studass/vitass.

Viktige dato	Beskrivelse
Uke 8	UML
Uke 11	FAT
Uke 12	Refleksjon

Table 1: Viktige datoer å merke

III Introduksjon - Heisen på sanntidslaben

Vi skal bruke en fysisk modell av en heis i løpet av prosjektet som skal styres med et styringssystem ved hjelp av en Arduino. heis-modellen består av tre hoveddeler; selve heismodellen, et betjeningspanel, og en motorstyringsboks. Det finnes en heis-modell på nesten hver av sanntidssalens arbeidsplasser. Hensikten med denne labben er at dere skal programmere oppførselen til heisen med C uten å måtte ta hensyn til minne som allerede har blitt tatt på forhånd. Minnehåndtering i mikrokontrollere blir introdusert i mikrokontroller labben.

III .1 Heis-modell

Heis-modellen er illustrert i figur 1 og består av en heisstol som kan beveges opp og ned langs en stolpe. Dette tilsvarer henholdsvis heisrommet- og sjakten i en virkelig heis.

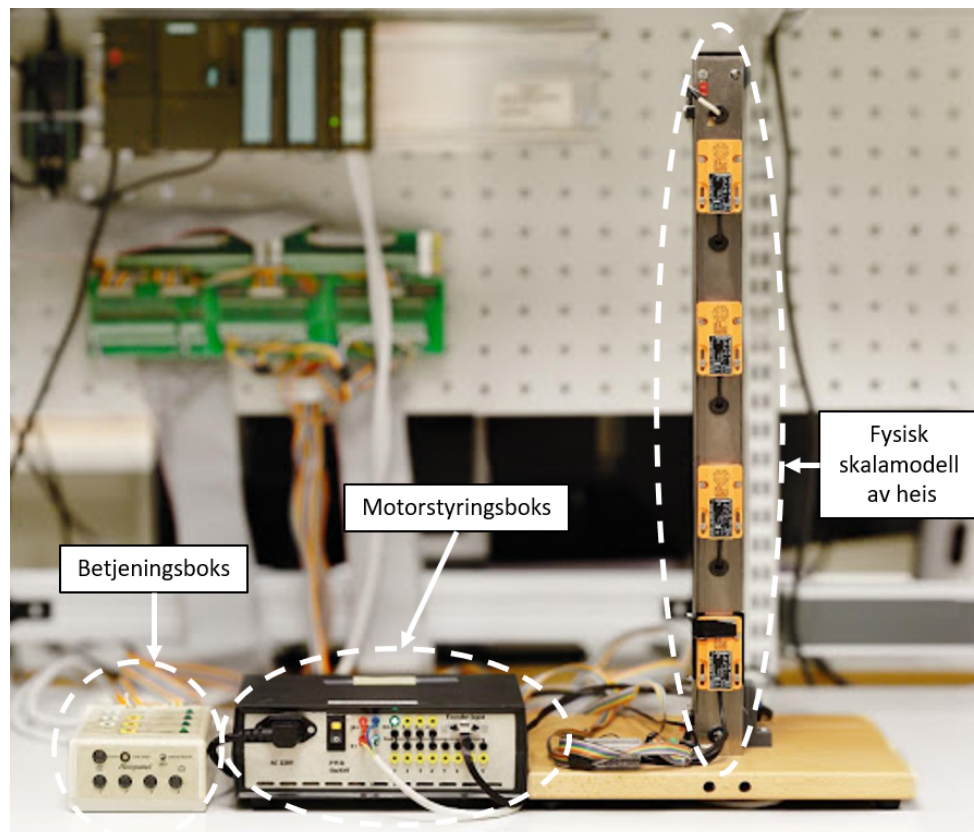


Figure 1: Heis-modellen på sanntidslaben.

Langs heisbanen er det montert fire hall-effektsensorer som fungerer som heisens etasjer. Over øverste etasje, og under nederste etasje er det også montert endestoppbrytere, som vil kutte motorpådraget dersom heisen kjører utenfor sitt lovlig område. Dette er for å beskytte heisens motor mot skade. Om heisen skulle treffe en av endestoppene, må heisstolen manuelt skyves bort fra bryterne før en kan be motoren om et nytt pådrag.

III .2 Betjeningsboks

Betjenings-boksen er delt i to; *etasjepanel* og *heispanel*. Disse kan man finne i figur 1 og 2. Øverst på betjeningsboksen finnes en bryter som velger om datamaskinen eller PLSen skal styre heismodellen. Denne skal stå i PC gjennom hele laben.

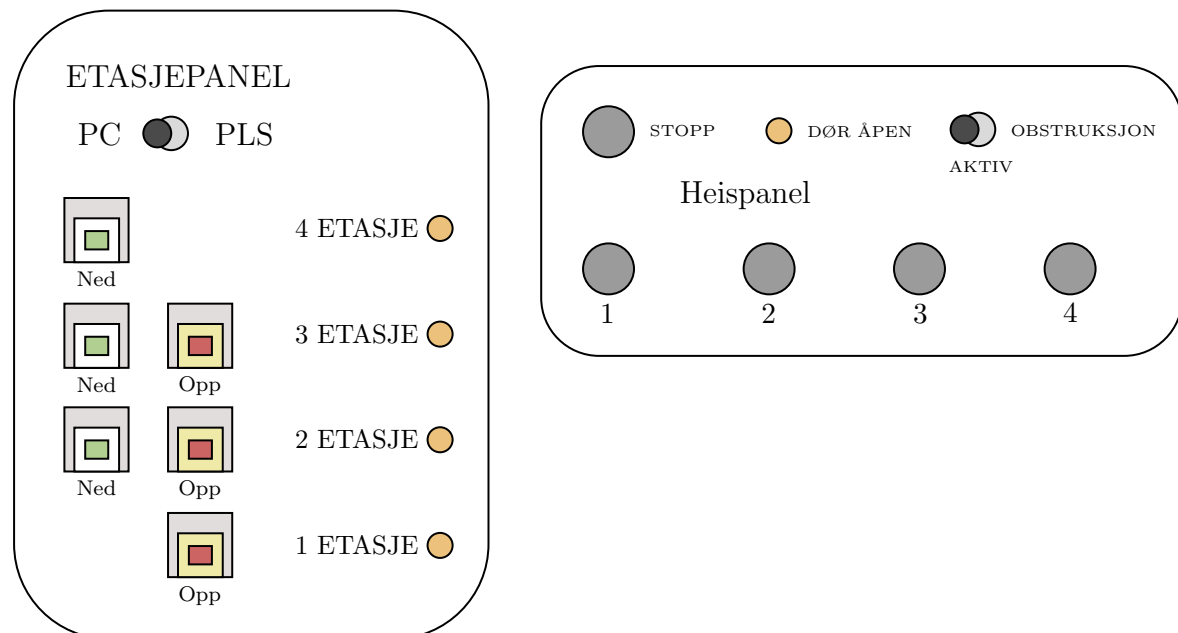


Figure 2: Etasje- og Heispanel i sanntidslabben

Etasjepanelet finnes på oversiden av betjeningsboksen. På dette panelet finner dere bestillingsknapper for opp- og nedretning fra hver etasje. Hver av knappene er utstyrt med lys som skal indikere om en bestilling er mottatt eller ei. Etasjepanelet har også ett lys for hver etasje for å indikere hvilken etasje heisen befinner seg i.

Heispanelet finnes på kortsiden av betjeningsboksen og representerer de knappene man forventer å finne inne i heisrommet til en vanlig heis. Her har man bestillingsknapper for hver etasje, samt en stoppknapp for nødstands. Alle knappene er utstyrt med lys som kan settes via styreprogrammet. I tillegg til knappene er panelet utstyrt med etasjeindikatorlys som kan settes via styreprogrammet og et lys, markert DØR ÅPEN, som indikerer om heisdøren er åpen. Heispanelet har også en obstruksjonsbryter, som kan brukes for å simulere at en person blokkerer døren når den er åpen.

III .3 Motorstyringsboks

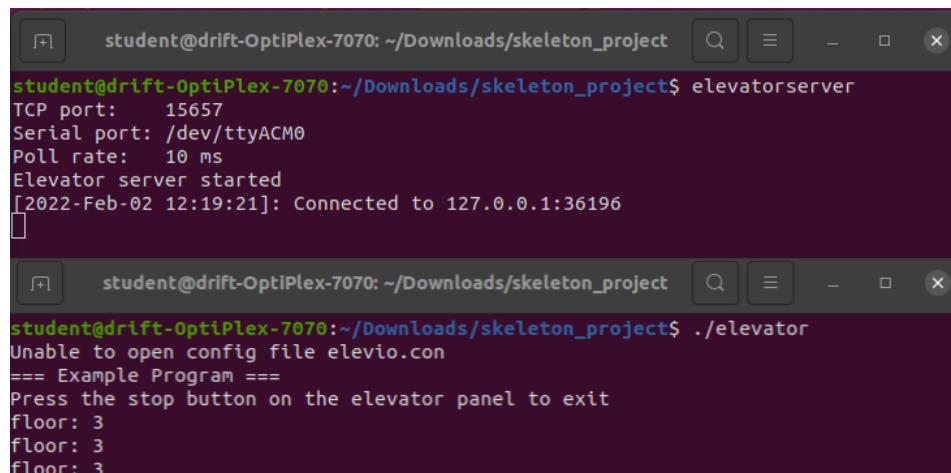
Styringsboksen er ansvarlig for å forsyne effekt til heis-modellen, og for å forsterke pådraget som settes av datamaskinen (se figur 1). Motoren kan forsynes med mellom 0- og 5 V, som henholdsvis er minimalt- og maksimalt pådrag. Veien motoren skal gå settes ved et ekstra retningsbit i styringsboksens grensesnitt. Alt dette gjøres via funksjonsskall i styringsprogrammet.

Det er også mulig å hente ut et analogt tacho-signal, samt en digital verdi for motorens encoder, for å lese av hastighet- og posisjon fra styringsboksen. Disse målesignalene trenger dere ikke ta stilling i dette prosjektet, men de nevnes for fullstendighetens skyld.

III .4 Virkemåte og oppkobling

Heis-modellen er laget for å konseptuelt oppføre seg som en virkelig heis. Det er et par punkter man bør merke seg for å få den til å fungere som ønsket:

- Alle lys må settes eksplisitt. Det er ingen automatikk mellom hall-sensoren i hver etasje og tilhørende etasje-indikator.
- Om endestopp-bryterne aktiveres vil pådrag til heisen kuttes. Om det skjer, må heisstolen manuelt skyves vekk fra endestopp.
- Rød og blå ledning forsyner effekt til motoren. Disse kobles henholdsvis til $M+$ og $M-$ som dere finner på motorstyringsboksen (se figur 1).
- Heisen kjøres ved at programmet kompiles med **make**, før styresystemet kjøres med **./elevator**. For at heisen skal ha noe å koble opp i mot, så må man først starte **elevatorserver** i et annet terminalvindu først (se figur 3).
- Dersom man jobber hjemmefra, startes heisen ved at man bruker simulatoren som server istedenfor. Likeså starter man simulatorserveren i et annet terminalvindu først med **./SimElevatorServer**, før man kompilerer og kjører **./elevator** i det opprinnelige terminalvinduet (se appendiks D).



```
student@drift-OptiPlex-7070: ~/Downloads/skeleton_project
student@drift-OptiPlex-7070:~/Downloads/skeleton_project$ elevatorserver
TCP port: 15657
Serial port: /dev/ttyACM0
Poll rate: 10 ms
Elevator server started
[2022-Feb-02 12:19:21]: Connected to 127.0.0.1:36196
█

student@drift-OptiPlex-7070: ~/Downloads/skeleton_project
student@drift-OptiPlex-7070:~/Downloads/skeleton_project$ ./elevator
Unable to open config file elevio.con
=== Example Program ===
Press the stop button on the elevator panel to exit
floor: 3
floor: 3
floor: 3
```

Figure 3: Skjerm bilde av terminalene og kommandoene som trengs for å kjøre heisen med utgitt kode.

III .5 Kommentar til kodekvalitet

Et lite appendiks om kodekvalitet har blitt lagt til i appendiks E. Det er ikke obligatorisk å følge, men kan gjøre videre utvikling lettere med hensyn på bugs.

A Appendiks - V-modellen

V-modellen er illustrert i figur 4. Det kan være fristende å hoppe direkte inn i implementasjonsfasen, men dere bør ikke ta for lett på hverken analyse og design, eller testing. Det er mye bedre med noen få linjer gjennomtenkt og veltestet kode, enn mange linjer med *spaghetti-kode*.

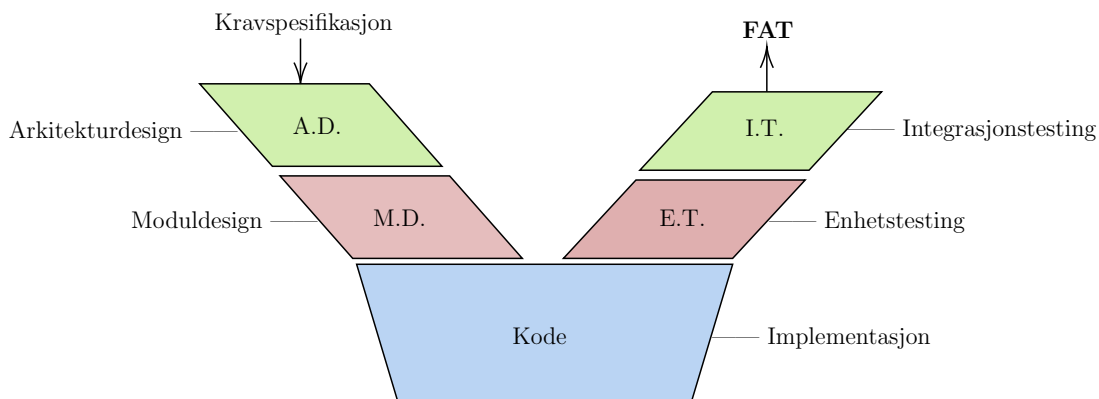


Figure 4: Illustrasjon av V-modellen.

A.1 Arkitekturdesign

Det aller viktigste er at man faktisk forstår kravene i spesifikasjonen. Når man først er sikre på hva som kreves av slutt-systemet, burde man tenke gjennom hvilke implikasjoner dette har for koden som skal skrives senere.

På dette stadiet ønsker vi å bestemme en *arkitektur* som vil oppfylle kravene fra spesifikasjonen. Dette innebærer å legge abstraksjonsnivået forholdsvis høyt, og ignorere implementasjonsdetaljer enn så lenge.

Eksempel: Heisen skal kunne huske ordre helt til de blir ekspedert. Ikke tenk: *Dette skal jeg implementere som en lenket liste*, men heller: *På arkitekturnivå trenger vi et køsystem*. Detaljer om hvordan et eventuelt køsystem er implementert kommer ikke inn i bildet på dette stadiet.

Resultatet av dette stadiet bør være ett eller flere klasse-diagrammer som illustrerer hvilke moduler styringssystemet skal bestå av. For å få en ide om hvilken funksjonalitet hver modul må tilby, kan det også være lurt å sette opp et par sekvensdiagrammer som illustrerer hvordan forskjellige moduler samarbeider. I tillegg, kan man også benytte kommunikasjons-diagrammer for å gi en bedre oversikt over grensesnittet mellom hver modul.

A.2 Moduldesign

Når man har en overordnet tanke over hvilke moduler som kreves for å oppfylle kravspesifikasjonen, er det på tide å skissere hvordan hver modul skal se ut. Et spørsmål som er lurt å ta stilling til her er om hver modul trenger å lagre tilstander eller ikke.

Eksempel: Et kø-system trenger åpenbart å lagre tilstand, mens en modul som utelukkende setter pådrag til motor-styringsboksen kanskje kan skrives uten å ha hukommelse.

En modul som ikke trenger å lagre tilstander vil alltid ha færre måter den kan feile på enn en tilsvarende modul som lagrer tilstand. Det kan derfor være lurt å skille ut delene av systemet som har denne funksjonaliteten i en dedikert tilstandsmaskin, og beholde hjelpemoduler så enkle som mulig.

Her bør man altså benytte seg av klasse-diagrammer og tilstands-diagrammer. Motivasjonen for å gjøre dette er at det er mye lettere å eksperimentere og endre på designet på diagramnivå, enn med halvferdig kode.

A.3 Implementasjon

Det er på dette stadiet dere skriver kode. Hvis dere har lagt inn en grei innsats i design-fasen, vil dette stadiet stort sett koke ned til å oversette diagrammene til kode. Så feilfritt går det selvsagt aldri, men et godt forarbeid kan spare for mye hodebry. Man burde også ikke være redd for å gå tilbake for å endre på arkitekturen eller modulsammensetningen om man finner mer hensiktsmessige måter å gjøre noe på.

Det kan også være interessant å nevne forskjellen mellom å *programmere inn i et språk* og *programmere i et språk*. Om man programmerer *i et språk*, vil man begrense abstraksjonskonseptene og tankesettet sitt til de primitive som språket direkte støtter. Om man deretter programmerer *inn i et språk* vil man først bestemme seg for hvilke konsepter man ønsker å strukturere programmet inn i, og deretter finne måter å implementere konseptene på i språket man skriver.

Eksempel: C er ikke i utgangspunktet objektorientert. Allikevel kan man se på hver klasse i et klasse-diagram som en egen modul, hvor alle funksjonene som modulen gjør tilgjengelig svarer til offentlige medlemsfunksjoner i en klasse.

På den annen side er det selvsagt en fordel å benytte seg av de primitive et språk støtter direkte, fremfor å prøve å tvinge inn funksjonalitet som ikke gis av språket, men det er alltid greit å tenke gjennom et program som en abstrakt oppskrift på hvordan man løser et problem - før man tenker for eksempel: *dette kan implementeres som en klasse som arver fra en annen*.

A.4 Enhetstesting

Enhetstesting speiler moduldesignfasen. Her tester man for å forsikre om at hver modul oppfører seg som den skal. I første omgang er det greit å gjøre små, veldig veldefinerte tester, som tester ut en bestemt funksjon fra modulen dere prøver ut.

Antallet tester er ikke et bra mål på hvor godt testet en modul er, så sikt heller på å teste forskjellige ting. *Border cases* er stort sett en langt større kilde til feil

enn vanlige tilfeller, så det er mye mer verdifullt med *tester som tester forskjellige ting* enn med *forskjellige tester som tester samme ting*.

Eksempel: Gitt at vi har en heis med 10 etasjer. Det kan da være mer fornuftig å lage en test som sjekker om en modul for å håndtere motorstyring av heisen fungerer mellom etasjenummer 2-9, for å deretter sjekke hvordan modulen reagerer på bestillinger til etasjenummer 1 og 10 som er *border cases*. I tillegg er det hensiktsmessig å teste for etasjenummer 0 og 11 slik at man sikrer mot eventuelle feil som kan oppstå om hall-sensorene ikke fanger at heisen har passert etasjenummer 1 eller 10.

A.5 Integrasjonstesting

Enhetstesting foregår på modul-nivå og svarer på spørsmålet: *Fungerer denne modulen som den skal?*. Integrasjonstesting speiler arkitekturdesignfasen, og svarer på spørsmålet: *Fungerer denne modulen sammen med andre moduler?*. Her vil man typisk prøve ut hele- eller nesten hele programmet på en spesifikk funksjonalitet. Om man har tatt seg god tid til å lage gode seksvensdiagrammer, kommer disse godt med i denne fasen.

Integrasjonstesting kan enten være en særdeles enkel oppgave, eller veldig komplisert, avhengig av graden kobling man har mellom modulene i programmet. Moduler som avhenger sterkt av andre moduler blir nødvendigvis både vanskeligere å teste, og å vedlikehold. Derfor er det ønskelig at moduler kun vet om - og kommuniserer med akkurat de modulene den trenger.

Eksempel: 23. September 1999 ble *Mars Climate Orbiter* tapt etter at fartøyet enten gikk i stykker i Mars' atmosfære, eller spratt tilbake til en utilsiktet heliosentrisk bane. Styringssystemet til sonden bestod av software skrevet av Lockheed Martin og NASA: Begge hadde testet sine egne moduler, men Lockheed Martin sine moduler opererte med *pund per sekund* (lbs), mens NASA sine moduler opererte med *newton per sekund* (Ns). Manglende integrasjonstesting endte totalt opp med å koste NASA JPL omlag 330 millioner amerikanske dollar.

B FAT - Factory Acceptance Test

Kravspesifikasjonene som FAT-en er basert på finner dere i appendiks [B.1](#), mens selve FAT-testen som baserer seg på kravspesifikasjonene finner dere i appendiks [B.2](#). I praksis så er det vanlig at kunde og leverandør avtaler disse på forhånd. FAT-en bestemmer i hvilken grad dere har implementert et korrekt system og er en direkte gjenspeiling av kravspesifikasjonen fra appendiks [B.1](#), så om dere oppfyller alle kravene som er satt av den, har dere implementert et fullverdig system.

Krav: Oppstart

Punkt	Beskrivelse
O1	Ved oppstart skal heisen alltid komme til en definert tilstand. En definert tilstand betyr at styresystemet vet hvilken etasje heisen står i.
O2	Om heisen starter i en udefinert tilstand, skal heissystemet ignorere alle forsøk på å gjøre bestillinger, før systemet er kommet i en definert tilstand.
O3	Heissystemet skal ikke ta i betraktning urealistiske start-betingelser, som at heisen er over 4 etasje, eller under 1 etasje idet systemet skrur på.

Krav: Håndtering av bestillinger

Punkt	Beskrivelse
H1	Det skal ikke være mulig å komme i en situasjon hvor en bestilling ikke blir tatt. Alle bestillinger skal betjenes selv om nye bestillinger opprettes.
H2	Heisen skal ikke betjene bestillinger fra utenfor heisrommet om heisen er i bevegelse i motsatt retning av bestillingen.
H3	Når heisen først stopper i en etasje, skal det antas at alle som venter i etasjen går på, og at alle som skal av i etasjen går av. Dermed skal alle ordre i etasjen være regnet som ekspedert.
H4	Heisen skal stå stille om den ikke har noen ubetjente bestillinger.

B.1 FAT - Heisspesifikasjoner

Krav: Bestillingslys- og etasjelys

Punkt	Beskrivelse
L1	Når en bestilling gjøres, skal lyset i bestillingsknappen lyse helt til bestillingen er utført. Dette gjelder både bestillinger inne i heisen, og bestillinger utenfor.
L2	Om en bestillingsknapp ikke har en tilhørende bestilling, skal lyset i knappen være slukket.
L3	Når heisen er i en etasje skal korrekt etasjelys være tent.
L4	Når heisen er i bevegelse mellom to etasjer, skal etasjelyset til etasjen heisen sist var i være tent.
L5	Kun ett etasjelys skal være tent av gangen.
L6	Stoppknappen skal lyse så lenge denne er trykket inne. Den skal slukkes straks knappen slippes.

Krav: Heis-dør

Punkt	Beskrivelse
D1	Når heisen ankommer en etasje det er gjort bestilling til, skal døren åpnes i 3 sekunder, for deretter å lukkes.
D2	Heisen skal være lukket når den ikke har ubetjente bestillinger.
D3	Hvis stoppknappen trykkes mens heisen er i en etasje, skal døren åpne seg. Døren skal forholde seg åpen så lenge stoppknappen er aktivert, og ytterligere 3 sekunder etter at stoppknappen er sluppet. Deretter skal døren lukke seg.
D4	Om obstruksjonsbryteren er aktivert mens døren først er åpen, skal den forbli åpen så lenge bryteren er aktiv. Når obstruksjonssignalet går lavt, skal døren lukke seg etter 3 sekunder.

Krav: Sikkerhet

Punkt	Beskrivelse
S1	Heisen skal alltid stå stille når døren er åpen.
S2	Heisdøren skal aldri åpne seg utenfor en etasje.
S3	Heisen skal aldri kjøre utenfor området definert av 1 til 4 etasje.
S4	Om stoppknappen trykkes, skal heisen stoppe momentant.
S5	Om stoppknappen trykkes, skal alle heisens ubetjente bestillinger slettes.
S6	Så lenge stoppknappen holdes inne, skal heisen ignorere alle forsøk på å gjøre bestillinger.
S7	Etter at stoppknappen er blitt sluppet, skal heisen stå i ro til den får nye bestillinger.

Krav: Robusthet

Punkt	Beskrivelse
R1	Obstruksjonsbryteren skal ikke påvirke systemet når døren ikke er åpen.
R2	Det skal ikke være nødvendig å starte programmet på nytt som følge av eksempelvis udefinert oppførsel som for eksempel at programmet krasjer, eller minnelekkasje.
R3	Etter at heisen først er kommet i en definert tilstand ved oppstart, skal ikke heisen trenge flere kalibreringsrunder for å vite hvor den er.

Krav: Tillegg

Punkt	Beskrivelse
Y1	Oppførsel som ikke er <i>vanlig heisoppførsel</i> kan gi trekk på FAT-testen. Når det er sagt så er det bare å bruke sunn fornuft og eventuelt spør vitass eller foreleser om noe er uklart.

B.2 FAT - Testspesifikasjoner

FAT-test: Oppstart

Punkt	Beskrivelse
O1	Sørger systemet for at heisen kommer i en definert tilstand?
O2	Ignorerer bestillinger før heisen har kommet i en definert tilstand?
O3	Ignorerer stoppknappen under initialisering?

FAT-test: Håndtering av bestillinger

Punkt	Beskrivelse
H1	Går heisen til riktig etasje når en bestilling mottas fra etasjepanelet?
H2	Går heisen til riktig etasje når en bestilling mottas fra heispanelet?
H3	Hvis heisen er på vei fra 4. etg til 1. etg og noen har bestilt OPP i 2. etg: kjører heisen til 1. etg før den kjører til 2. etg?
H4	Håndteres alle bestillingene hvis flere av bestillingsknappene trykkes samtidig?
H5	Vil alle bestillinger bli ekspedert, selv med vedvarende trykking av andre knapper (unntatt stopp), dvs. blir heisen aldri "fastlåst" mellom noen av etasjene?

FAT-test: Bestillingslys og etasjelys

Punkt	Beskrivelse
L1	Blir riktig etasjelys tent når heisen ankommer en etasje?
L2	Hvis heisen befinner seg mellom 2. og 3. etg og er på vei oppover, lyser etasjelyset i 2. etg?
L3	Blir lyset tent i bestillingsknappene når de blir trykket?
L4	Slukker lyset i bestillingsknappene når bestillingen er ekspedert, dvs. når heisen ankommer etasjen?

FAT-test: Heis-dør

Punkt	Beskrivelse
D1	Åpnes døren (lyser dørlyset) når heisen stopper i en etasje?
D2	Er døren åpen i 3 sekunder?
D3	Står heisen stille i de 3 sekundene døren er åpen?
D4	Lukkes døren før heisen kjøres videre?
D5	Lukkes døren og står heisen stille når det ikke er noen nye bestillinger?

FAT-test: Sikkerhet

Punkt	Beskrivelse
S1	Stopper heisen når stoppknappen trykkes?
S2	Blir bestillingene slettet (lysene på bestillingsknappene slukkes) når stoppknappen trykkes?
S3	Er lyset i stoppknappen tent mens stoppknappen er trykket?
S4	Ignorerer trykk på alle bestillingsknappene mens stoppknappen er trykket?
S5	Blir heisen stående i ro etter at stoppknappen er sluppet?
S6	Husker heisen hvor den er ved nødstop mellom etasjer (dvs. kreves ikke ny initialisering)?
S7	Åpnes døren hvis stoppknappen aktiveres i en etasje?

FAT-test: Robusthet

Punkt	Beskrivelse
R1	Hvor stabilt er programmet? Må programmet startes på nytt under presentasjonen?

C Appendiks - Kommentar til refleksjonsdelen

I refleksjonsdelen, så er meningen at dere skal få litt tid til å reflektere for å deretter få mer innsikt i hva dere faktisk har gjort. Dette er spesielt viktig når man bruker verktøy som UML og V-modellen, hvor hva man planlegger ikke alltid er det som faktisk blir implementert. Dette kan være på grunn av at man finner andre mer hensiktsmessige måter å designe/velge moduler på, eller rett og slett fordi man innser at de valgene man tok på starten ikke leder til et robust, skalerbart, eller vedlikeholdbart system som til syvende og sist er det viktigste for et slikt system.

Denne delen skal mest være for at dere skal få muligheten til å snakke med studass/vitass og få umiddelbar kommentar istedenfor at dere skal sende inn en rapport og få kommentarer uker senere.

C.1 UML og V-modellen

- Hvordan har bruken av UML og V-modellen påvirket implementasjonen i startfasen?
- Hvordan har bruken av UML og V-modellen påvirket implementasjonen i midtfasen?
- Hvordan har bruken av UML og V-modellen påvirket implementasjonen i slutfasen?

- Hadde prosjektet blitt lettere/vanskeligere uten UML og V-modellen? Kunne prosjektet ha blitt mye lettere med en annen metode?

C.2 Robusthet, skalarbarhet og vedlikehold

- Oppgaven har ikke eksplisitt sagt at robusthet, skalarbarhet, og vedlikehold skal prioriteres, men har bruken av UML og V-modellen bidratt til at noen av disse egenskapene har blitt fremmet? I såfall, hvilken?
- Alternativt, har bruken av UML og V-modellen gjort at robustheten, skalarbarheten eller vedlikeholden verre enn ved å ikke bruke UML eller V-modellen i det hele tatt?

D Appendiks - Heissimulator

I tillegg til den fysiske heisen, har vi en heissimulator¹ som kan brukes dersom man har lyst til å jobbe hjemme. Denne simulatoren har samme funksjonalitet som den fysiske heisen på sanntidssalen, og fungerer som et ypperlig verktøy dersom man vil teste heiskoden hjemme. Det som gjør heissimulatoren spesielt interessant, er at man kan velge hvor mange etasjer heisen eventuelt skal ha. Dere kan derfor teste ut programmet deres på en heis som har mange flere etasjer, enn det heisen på sanntidssalen har.

Liten advarsel til folk som bruker macOS eller Windows: Denne simulatoren er beregnet på Linux. Det er ingen garanti at den funker på macOS eller Windows. Dersom dere ikke har Linux på personlig datamaskin, er det anbefalt å laste ned Ubuntu enten gjennom *dual booting*, eller med en virtuell maskin (referer til øving 1 for hvordan dette gjøres).

D.1 Initialisering av heissimulator

For å initialisere heissimulatoren, må man gjøre følgende:

1. Åpne terminalen i `skeleton_project` mappen.
2. Kjør kommandoen `chmod +x SimElevatorServer` for å gjøre det mulig å kjøre simulatoren som et program. Dette trenger dere bare å gjøre én gang.
3. Kjør kommandoen `./SimElevatorServer` i terminalen for å starte simulatoren.
4. Åpne en annen terminal i `skeleton_project` mappen
5. Kompiler heisprogrammet i den nye terminal som vanlig med `make` og `./elevator`.

¹Skrevet av Torjus Bakkene og Erlend Blomseth.

D.2 Heissimulator grensesnitt

Dersom alt har blitt gjort riktig, burde dere få opp grensesnittet til heissimulatoren som vist i figur 5. I dette grensesnittet, symboliserer * at knappen, enten inne, eller ute har blitt aktivert (avhengig om det er Hall Up, Hall Down, eller Cab). Dette tilsvarer når man trykker på de fysiske knappene i etasje- og heispanelene i den fysiske heisen på sanntidssalen.

```

+-----+-----+
|          |          #>          |
| Floor    | 0  1*  2  3 |Connected
+-----+-----+
| Hall Up   | *  -  -          | Door:   -  |
| Hall Down |    -  -    *        | Stop:   -  |
| Cab       | -  -    *  -  | Obstr:  ^  |
+-----+-----+43+

```

Figure 5: Grensesnittet til heissimulatoren.

I tillegg, viser grensesnittet retningen heisen beveger seg i, med # (ligger rett over Floor). #> betyr at heisen er på vei oppover, mens <# betyr at heisen er på vei nedover. Tallet helt nede til høyre viser hvor mange ganger en ny tilstand har blitt printet.

Til slutt er det verdt å merke seg at hver Floor kan bli merket med * (i figur 5 er 1 merket). Dette tilsvarer etasjelysene i figur 6 (de gule sirklene ved siden av etasjeindikatorene).

D.3 Hvordan bruke simulatoren

For å bruke heissimulatoren, bruker man følgende tastetrykk for å styre den simulerte heisen (se figur 6 for hvilke tastetrykk som tilsvarer hva på simulatoren):

- Heisknapp (opp) : qwe
- Heisknapp (ned) : sdf
- Heisknapp (inne) : zxcv
- Obstruksjonsknapp : -
- Stoppknapp : p

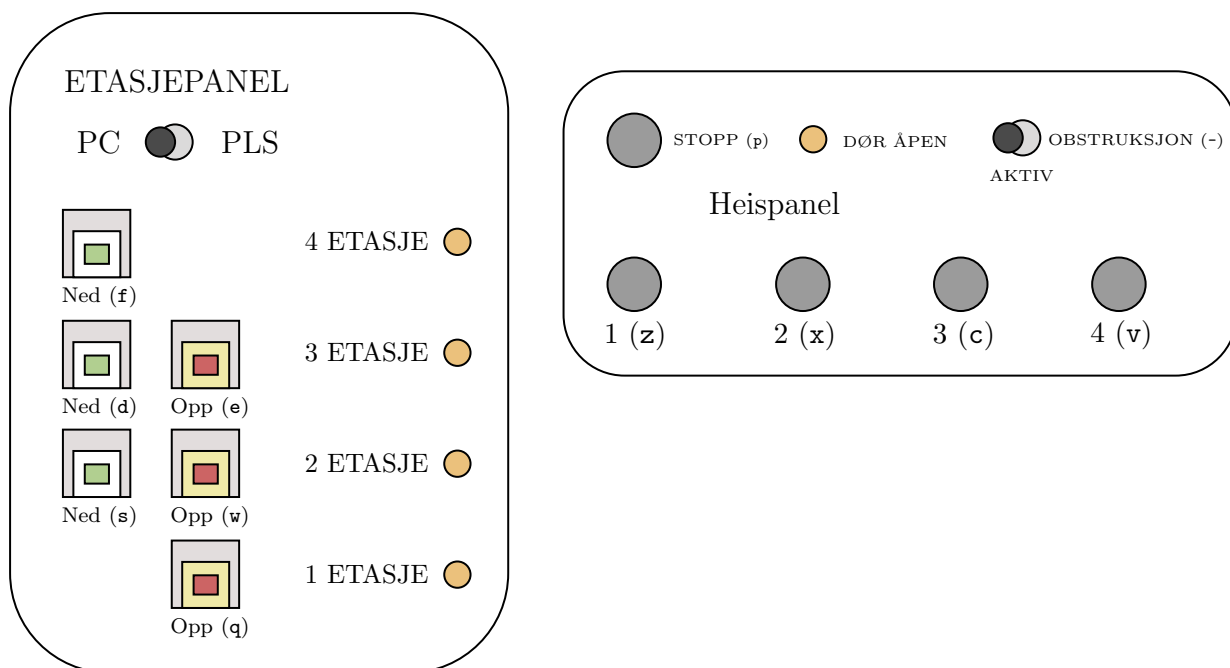


Figure 6: Etasje- og Heispanel i heissimulatoren.

E Appendiks - Kodekvalitet

Kodekvalitet er et mål på hvor lesbar og *åpenbar* koden er. Koden er lesbar om man er i stand til å ta en snutt og si nøyaktig hva den gjør, og hvorfor den gjør det, basert på koden alene. For deres egen del bør dere derfor alltid ha kodekvalitet i bakhodet hele tiden. Felles for god kode er at det resulterer i en lesbar kode som er lettere å vedlikeholde enn mindre god kode.

Ute i virkelige settinger bruker man mye mer tid på å lese kode enn å skrive kode selv. Uavhengig om det er noen andre sin kode, eller deres egen kode en del frem i tid, er det mye greiere om den er skrevet for leserens skyld - enn at den er skrevet for å spare nanosekunder på ubetydelige steder.

Under er det oppgitt en liste av hva som regnes som god kodekvalitet i dette faget. Listen er i stor grad basert på *Code Complete 2* av Steve McConnell, men noen ekstra punkter som er nyttige for C er også lagt til. Det er helt greit å være uenig i deler av- eller hele listen, men da bør man ha en god konvensjon som man bruker konsekvent.

E.1 Moduler

- Alle funksjonene i en modul bør ha samme abstraksjonsnivå. Man burde aldri blande lav-nivå funksjonalitet med høy-nivå funksjonalitet.
- En modul har som formål å gjemme noe bak et grensesnitt. Det bør altså ikke lekke ut detaljer om modulens implementasjon til overflaten. Ideelt

sett skal man kunne bruke modulen uten å vite om hvordan modulen var implementert.

- Hver modul skal ha en sentral oppgave. En modul bør dermed ikke håndtere flere vidt forskjellige ansvarsområder.
- Grensesnittet til hver modul bør gjøre det helt åpenbart hvordan modulen skal brukes. I dette ligger det også å navngi modul-funksjoner logisk og presist.
- Moduler bør snakke med så få andre moduler som mulig, og samarbeidet mellom moduler bør være så lett koblet som mulig. Dere skal altså kunne fjerne en modul, uten å måtte endre på alle andre som inngår i programmet.
- For klasser er det ønskelig at alle medlems-variabler er definerte etter at konstruktøren har kjørt. Tilsvarende for moduler er det ønskelig at all medlems-data er definert etter en eventuell initialiseringsfunksjon.

E.2 Funksjoner

- Den viktigste grunnen til å opprette en funksjon er ikke kodegjenbruk, men å gi brukeren en måte å håndtere kompleksitet på. Det er tilfeller hvor det faktisk er mer lesbart å repetere dere selv et par ganger, fremfor å trekke noen få linjer ut i en egen funksjon.
- Alle funksjoner bør ha ett eneste ansvarsområde - en oppgave som funksjonen gjør bra.
- Navnet på en funksjon bør beskrive alt funksjonen gjør.
- Sterke verb foretrekkes fremfor svake- og vage verb. For eksempel bør dere sky ord som **handle** eller **manage**.
- Kohesjon er et viktig begrep for å klassifisere funksjoner:
 - **Sekvensiell kohesjon** beskriver funksjoner hvor stegene som tas innad i funksjonen må gjøres i den bestemte rekkefølgen de er satt opp i.
 - **Kommunikasjons-kohesjon** er når en funksjon benytter samme data til å gjøre forskjellige ting, men hvor bruken av data-en ellers er urelatert.
 - **Tidsavhengig kohesjon** har man hvis en funksjon inneholder mange forskjellige operasjoner som gjøres til samme tid, men som ellers ikke har noe med hverandre å gjøre.
 - **Funksjonell kohesjon** har man i funksjoner hvor instruksjonene som kalles samarbeider for å gjøre en og samme ting. Tingene funksjonen gjør er altså nødvendige for å utføre en bestemt oppgave.

Av disse er det mest ønskelig å ha funksjonell kohesjon.

- Motsatte verb, bør være presise og opptre i veldefinerte par som: **begin - end**, **create - destroy**, **open - close** eller **next - previous**
- Funksjoner bør være *self-contained*, slik at de til en liten grad avhenger av returverdien til andre funksjoner. Om dette ikke er mulig, bør denne koblingen være så løs som mulig slik at man ikke trenger å endre mange andre funksjoner når man skal modifisere en spesifikk funksjon.

E.3 Variabler

- Unngå å bruke for mange *arbeids-variabler* - variabler som opprettes i starten av en funksjon for så å muteres gjennom hele funksjonens levetid.
- Navnekvaliteten til en variabel bør speile variabelens levetid. Variabler som brukes til å itere en løkke kan hete *i*, mens en global variabel bør ha et virkelig godt navn som presist beskriver variabelen.

E.4 Kommentarer

- Når man kommenterer kode, er det en erkjennelse om at koden som kommentaren beskriver ikke er åpenbar. Åpenbar kode som forklarer seg selv trenger ikke kommentarer, og er bedre enn uklar kode med kommentarer.
- Alle kommentarer må være oppdatert. Det er fort gjort å endre kode, uten å endre kommentarene rundt. Kode med ukorrekte kommentarer har mindre verdi enn kode uten kommentarer.

E.5 Øvrig

- Funksjoner bør prefikses med navnet på modulen sin for å gjøre det åpenbart hvor funksjonen kommer fra, og for å gjøre samme jobb som et namespace.
- Variabler kan med fordel prefikses med **p_** om de er pekere, **pp_** om de er pekere til pekere, **m_** om de er modul-variabler begrenset med kodeordet **static**, og **g_** om de er globale.
- Forkortelser er greit å bruke såfremt de er veletablerte. Et eksempel på veletablert forkortelse er **FSM** (finite-state machine = tilstandsmaskin). For eksempel: forkortelser av typen **EHM** for "etasjehåndteringsmodul" bør unngås.
- Selv om C er forholdsvis lavnivå er det fullt mulig å ivareta god softwareutviklingspraksis. Allikevel kommer man alltid til å skrive noe *uggen* kode - det gjelder bare å velge det beste alternativet tilgjengelig.
- Det er helt greit å være uenig i denne listen, eller ha andre konvensjoner dere vil heller følge, så lenge dette kan argumenteres for at de gir god lesbarhet og vedlikeholdbarhet.