



## Revisjonshistorie

År	Forfatter
2014	Øyvind Stavdahl Anders Rønning Petersen
2016	Øyvind Stavdahl Konstanze Kölle
2017	Ragnar Ranøyen Homb Bjørn-Olav Holtung Eriksen
2020	Kolbjørn Austreng
2021 - 2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr
2025	Kristian Blom

## I Introduksjon - Praktisk rundt laben

I heisprosjektet skal dere bruke det dere har lært i de tidligere fem øvingene til å samarbeide om et større prosjekt, altså til utviklingen av programvare for logikkstyring av et fysisk system, i dette tilfellet en heismodell.

Som dere vet, er programmeringsspråket **C** et kraftig verktøy som benyttes i et bredt spekter av industrielle sammenhenger, særlig i sanntidsapplikasjoner og maskinnær programvare. Denne laben går ut på å benytte **C** til å implementere et styresystem for en fysisk heismodell på Sanntidssalen (se introduksjon [III](#)), som styres gjennom en Arduino. I tillegg til selve implementasjonen, skal systemet beskrives og dokumenteres med UML før dere starter med selve implementasjonen.

For å strukturere arbeidet, og sikre verifikasjon av akseptkriterier, skal den pragmatiske V-modellen benyttes (se appendiks [A](#)), og den tilhørende rapporten skal speile dette. Flittig bruk av **git** vil gjøre utførelsen av prosjektet enklere, og selv om dette ikke er obligatorisk, så er det anbefalt å bruke **git** sammen med en kodevertsplattform slik som GitHub. Ettersom dere skal følge den pragmatiske V-modellen i dette prosjektet, er det derfor lurt at **git**-historikken speiler dette.

## Godkjenning

Heislaben teller ikke på sluttkarakteren, 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. Dersom dere har brukt generative KI-verktøy slik som ChatGPT i arbeidet med heisprosjektet, så skal dette også dokumenteres. Se appendiks D for kommentar på de viktigste punktene.

God bruk av `git` og dokumentering av kode med `doxygen` er ikke obligatorisk, men begge deler vil gjøre arbeidet med heisprosjektet enklere for dere, samtidig som dette er nyttige verktøy å mestre til senere arbeid. I tillegg vil begge disse delene telle positivt for prosjektet, og kunne trekke dere opp dersom det finnes andre mangler ved prosjektet deres.

UML- og refleksjonsdelen inngår i en rapport som dere skal skrive. Rapporten skal være på maks ti sider, inkludert alt av figurer, og kan skrives på norsk (eller andre skandinaviske språk), engelsk eller tysk. Denne skal inneholde alle de tidligere nevnte kravene, samt alt annet som trengs for å beskrive prosjektet deres. Denne rapporten leveres på BlackBoard sammen med all kildekoden deres, altså alt som trengs for å kunne gjenskape resultatene fra FATen.

I tillegg er det viktig at gruppenummeret og navnene på alle gruppemedlemmer tydeliggjøres i rapporten. Det holder med at ett gruppemedlem leverer på Blackboard.

## Viktige datoer

Som man kan se fra tabell 1, så blir FAT-en utført i uke 11. 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. Rapporten og koden deres leveres inn på Blackboard som en zip-fil uken etter.

Viktige dato	Beskrivelse
Labtid, uke 11	FAT
Fredag 20. mars	Rapport + kode som zip-fil

Table 1: Viktige datoer å merke

## II Introduksjon - Praktisk rundt de utleverte filene

I denne laben får dere utlevert noen `.c` og `.h`-filer under `skeleton_project`-mappen. Tabellen under lister opp alle filene som blir utlevert til bruk i heis-prosjektet, samt litt informasjon om hvorvidt 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 komplikasjoner seinere i prosjektet. 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 filen(e) endres?
<code>skeleton_project/Makefile</code>	Ja
<code>skeleton_project/SimElevatorServer</code>	Nei
<code>skeleton_project/SimElevatorServer.exe</code>	Nei
<code>skeleton_project/simulator.con</code>	Nei
<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/elevio.con</code>	Nei
<code>skeleton_project/source/driver/con_load.h</code>	Nei
<code>skeleton_project/source/driver/elev_test.c</code>	Nei
<code>Main/*</code>	Nei
<code>.github/*</code>	Nei



## III .2 Betjeningsboks

Betjeningsboksen 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.

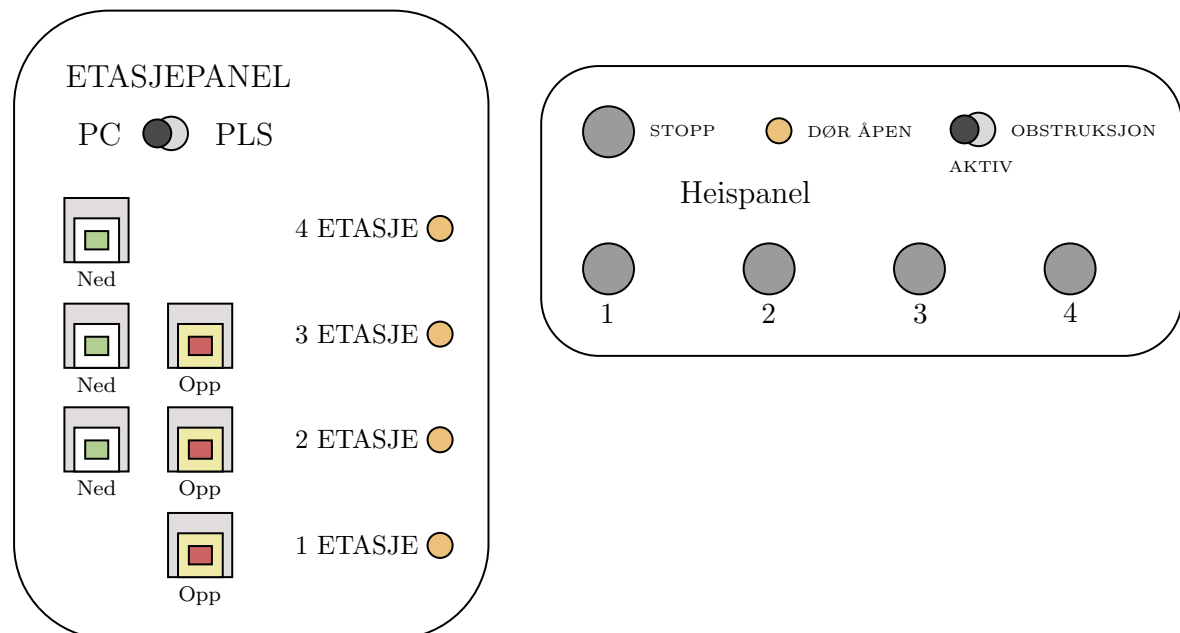


Figure 2: Etasje- og Heispanel i sanntidslaben

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 et 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

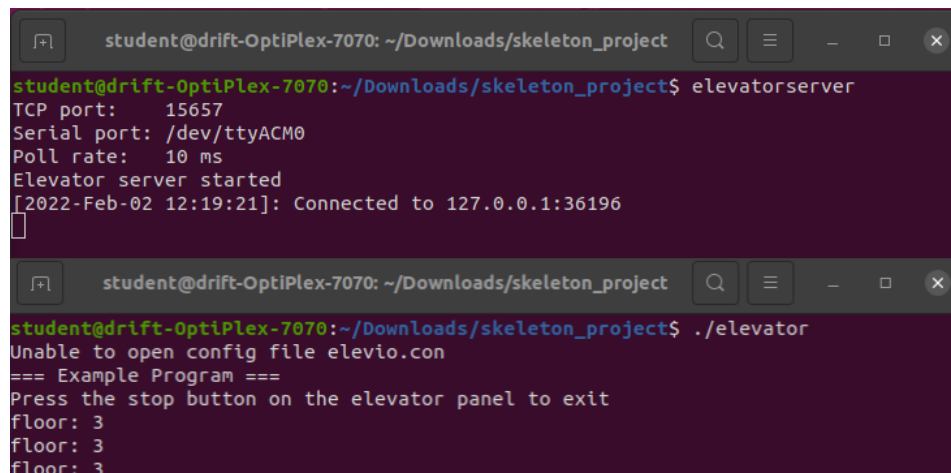
Styringsboksen er ansvarlig for å forsyne effekt til heismodellen, og for å forsterke pådraget som settes av datamaskinen (se figur 1). Motoren kan forsynes med mellom 0 og 5 V, som henholdsvis er minimalt og maksimalt pådrag. Veien motoren skal gå settes ved et ekstra retningsbit i styringsboksens grensesnitt.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

Det er også mulig å hente ut et analogt tacho-signal, samt en digital verdi for motorens enkoder, 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

Heismodellen er laget for å 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 E).



```
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.

## 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*.

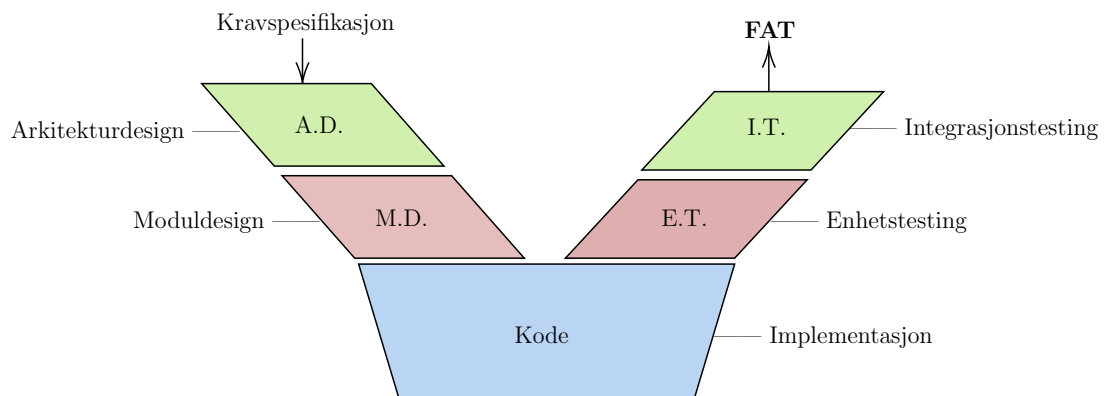


Figure 4: Illustrasjon av V-modellen.

### A.1 Arkitekturdesign

Det aller viktigste er at man faktisk forstår kravene i spesifikasjonen. Når man først er sikre på hva som kreves av sluttssystemet, 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. Dersom man mener at køsystem er en god løsning på dette kravet må man ikke tenke: *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.<sup>1</sup>

Resultatet av dette stadiet bør være ett eller flere klassediagrammer 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 kommunikasjonsdiagrammer for å gi en bedre oversikt over grensesnittet mellom hver modul. En *modul* kan i denne sammenheng forstås som en softwareenhet som er ansvarlig for å dekke funksjonalitet dere mener er nødvendig for å oppfylle ett eller flere punkter fra FAT'en.

<sup>1</sup>Det er ikke sikkert at et køsystem faktisk er nødvendig i dette prosjektet, dette er altså bare et eksempel

## 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 køsystem trenger åpenbart å lagre tilstand, mens en modul som utelukkende setter pådrag til motorstyringsboksen 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 holde hjelpemoduler så enkle som mulig.

Her bør man altså benytte seg av klassediagrammer og tilstandsdiagrammer. 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 designfasen 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 dere for mye hodebry. Man burde heller 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 primitiveene som språket støtter direkte. Om man derimot 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 i utgangspunktet ikke 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 primitiveene 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:** Sett at vi har en heis med 10 etasjer. Da kan det være fornuftig å lage en test som sjekker om en modul som håndterer motorstyring for 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 modulnivå 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 vedlikeholde enn moduler med svak kobling til hverandre. 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 *kraftpundsekunder* (lbfs), mens NASA sine moduler opererte med *newtonsekunder* (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 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.

## B.1 FAT - Heisspesifikasjoner

### 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 startbetingelser, 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 dersom 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 dersom den ikke har noen ubetjente bestillinger.

### **Krav: Bestillings- og etasjelys**

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

### **Krav: Heisdør**

Punkt	Beskrivelse
<b>D1</b>	Når heisen ankommer en etasje det er gjort bestilling til, skal døren åpnes i 3 sekunder, for deretter å lukkes.
<b>D2</b>	Heisen skal være lukket når den ikke har ubetjente bestillinger.
<b>D3</b>	Hvis stoppknappen trykkes mens heisen er i en etasje, skal døren åpne seg. Døren skal forbli åpen så lenge stoppknappen er aktivert, og ytterligere 3 sekunder etter at stoppknappen er sluppet. Deretter skal døren lukke seg.
<b>D4</b>	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
<b>S1</b>	Heisen skal alltid stå stille når døren er åpen.
<b>S2</b>	Heisdøren skal aldri åpne seg utenfor en etasje.
<b>S3</b>	Heisen skal aldri kjøre utenfor området definert av 1. til 4. etasje.
<b>S4</b>	Om stoppknappen trykkes, skal heisen stoppe momentant.
<b>S5</b>	Om stoppknappen trykkes, skal alle heisens ubetjente bestillinger slettes.
<b>S6</b>	Så lenge stoppknappen holdes inne, skal heisen ignorere alle forsøk på å gjøre bestillinger.
<b>S7</b>	Etter at stoppknappen er blitt sluppet, skal heisen stå i ro til den får nye bestillinger.

### Krav: Robusthet

Punkt	Beskrivelse
<b>R1</b>	Obstruksjonsbryteren skal ikke påvirke systemet når døren ikke er åpen.
<b>R2</b>	Det skal ikke være nødvendig å starte programmet på nytt som følge av eksempelvis udefinert oppførsel som for eksempel at programmet krasjer, eller minnelekkasje.
<b>R3</b>	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
<b>Y1</b>	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
<b>O1</b>	Sørger systemet for at heisen kommer i en definert tilstand?
<b>O2</b>	Ignorerer bestillinger før heisen har kommet i en definert tilstand?
<b>O3</b>	Ignorerer stoppknappen under initialisering?

### FAT-test: Håndtering av bestillinger

Punkt	Beskrivelse
<b>H1</b>	Går heisen til riktig etasje når en bestilling mottas fra etasjepanelet?
<b>H2</b>	Går heisen til riktig etasje når en bestilling mottas fra heispanelet?
<b>H3</b>	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?
<b>H4</b>	Håndteres alle bestillingene hvis flere av bestillingsknappene trykkes samtidig?
<b>H5</b>	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
<b>L1</b>	Blir riktig etasjelys tent når heisen ankommer en etasje?
<b>L2</b>	Hvis heisen befinner seg mellom 2. og 3. etg og er på vei oppover, lyser etasjelyset i 2. etg?
<b>L3</b>	Blir lyset tent i bestillingsknappene når de blir trykket?
<b>L4</b>	Slukker lyset i bestillingsknappene når bestillingen er ekspedert, dvs. når heisen ankommer etasjen?

### **FAT-test:** Heis-dør

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

### **FAT-test:** Sikkerhet

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

### **FAT-test:** Robusthet

Punkt	Beskrivelse
<b>R1</b>	Hvor stabilt er programmet? Må programmet startes på nytt under presentasjonen?

## **C    Appendiks - Bruk av KI**

Dere må oppgi all bruk av KI-baserte verktøy i arbeidet med heisprosjektet. Merk også at "feil" (dvs. åpen-sløyfe, se under) bruk av KI til å oppnå målene med

prosjektet kun vil medføre redusert læring for den som velger å gjøre dette.

Angående bruk av kunstig intelligens (KI) i programvareutvikling, kan man skille mellom to ulike bruksvarianter: åpen-sløyfe-bruk, og lukket-sløyfe-bruk. I kybernetikken introduseres vi for mange ulike redskap som vi kan benytte oss av til problemløsning, for eksempel Fourier-analyse og PID-regulatorer. KI kan også sees på som et slikt redskap, og i likhet med de andre redskapene, er vi nødt til å vite hvordan man faktisk skal bruke det. På samme måte som en kalkulator eller en datamaskin kan regne ut regnestykker for oss, kan KI hjelpe oss å løse problemer som ellers hadde vært tidskrevende. Det som skiller KI og en kalkulator er innsikt i hvordan redskapene fungerer. Vi stoler på en kalkulator siden vi kan reproducere og bekrefte informasjonen den gir oss via matematikk og algoritmer vi har lært gjennom utdanningen vår. På samme måte er det viktig at vi er klar over hvilken variant av KI vi bruker, og sørger for å bekrefte informasjonen KI-en gir oss, for å lukke sløyfen. Det spiller ingen rolle hvor rett kalkulatoren regner om vi skriver inn feil input, og det samme gjelder KI, spesielt av typen tekstgenererende språkmodeller som OpenAI sin `ChatGPT` og Meta sin `Llama`. Kort fortalt prøver språkmodeller å forutsi hva neste ord bør være gitt en tekststreng. Store språkmodeller er ikke-deterministiske, og har tendenser til å "hallusinere", altså at språkmodellen oppgir falsk informasjon som fakta. Hold derfor et kritisk blikk på hallusineringshyppigheten til språkmodellene dere bruker. Vi anbefaler at dere tar en titt på [denne siden](#) med hallusineringshyppigheter før dere eventuelt tar i bruk KI i dette faget eller andre fag under studiet.

Oppsummert forventer vi to ting fra studenter i TTK4235 når det gjelder KI: at dere ikke bruker KI i åpen sløyfe, og at dere alltid har et kritisk blikk på hva som genereres. I tillegg forventer vi at dere er ærlige om bruk av KI, og ikke angir arbeidet til KI som deres eget.

## D    **Appendiks - Kommentar til refleksjonsdelen**

I refleksjonsdelen, så er meningen at dere skal få litt tid til å reflektere rundt arbeidet deres, for å oppnå økt innsikt i hva dere faktisk har gjort. Dette er spesielt viktig når man bruker verktøy som UML og V-modellen, hvor det man planlegger ikke nødvendigvis 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.

Generativ KI er et relativt nytt verktøy, og det kan derfor være nyttig å være kritisk, og stille spørsmål ved hva man får som svar, når man bruker det. Bruk av KI er ikke problematisk i seg selv, men det er viktig å være klar over at det kun er et verktøy, og ingen erstatning for sunn fornuft og vitenskapelige metoder.

Denne delen gir dere noen eksempler på spørsmål dere burde stille dere selv i arbeidet med heislåben, og som burde drøftes rundt i rapporten deres. Dere

behøver ikke besvare nøyaktig disse spørsmålene, men de gir dere et inntrykk av hva som kan være lurt å skrive om i refleksjonsdelen av rapporten.

## D.1 UML og V-modellen

- Hvordan har bruken av UML og V-modellen påvirket implementasjonen i start-, midt-, og slutfasen?
- Hvilke deler av det opprinnelige designet deres ble endret på underveis, og hvorfor?
- Hadde prosjektet blitt lettere/vanskeligere uten UML og V-modellen? Kunne prosjektet ha blitt lettere med en annen metode?

## D.2 KI

- Hva har KI blitt brukt til i prosjektet?
- Hvilke effekter har KI hatt på deres arbeid med prosjektet?

## D.3 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 så fall, hvilke?
- 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?

# E Appendiks - Heissimulator

I tillegg til den fysiske heisen, har vi en heissimulator<sup>2</sup> 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 sømløst på macOS eller Windows. Dersom dere ikke har Linux på personlig datamaskin, er det anbefalt å bruke Ubuntu i et virtuelt miljø som [Windows Subsystem For Linux \(WSL\)](#) eller alternativt som [Lima](#) eller [Parallels for MacOS](#). Det er utgitt ferdigkompileerte programmer for Windows og Linux, men dersom du ønsker å bruke simulatoren

---

<sup>2</sup>Skrevet av Torjus Bakkene og Erlend Blomseth.

på MacOS henviser vi til [dokumentasjonen på Github](#) for hvordan man kompilerer simulatoren.

## E.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`.

## E.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).



### E.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`

### E.4 Heissimulator for MacOS og Windows

Github-organisasjonen for TTK4235 inneholder en `fork` av *Simulator-V2*, skrevet av Torjus Bakkene og Erlend Blomseth. `elevator_simulator`, tilgjengelig via [denne lenken her](#), inneholder binærfiler for simulatoren for Linux og Windows. Repositoriet inneholder også dokumentasjon for hvordan man kompilerer heissimulatoren på MacOS.

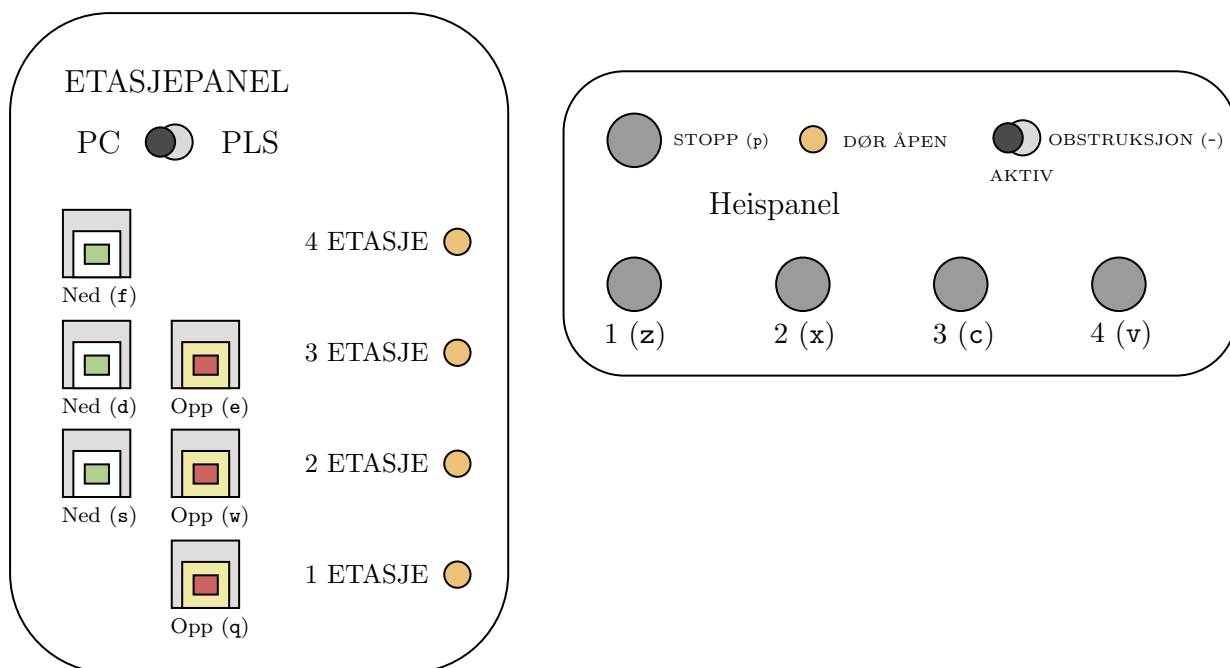


Figure 6: Etasje- og Heispanel i heissimulatoren.

## F 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.

### F.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.

## F.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.

### F.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.

### F.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.

### F.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.