TTK4235 - Tilpassede datasystemer
Vår 2021

Norges Teknisk-Naturvitenskapelige Universitet
Institutt for Teknisk Kybernetikk

Lab 2
Heis

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	Kiet Tuan Hoang

I Introduksjon - Praktisk rundt laben

Programmeringsspråket C er et kraftig verktøy som regelmessig benyttes i et bredt spekter industrisammenhenger, særlig i sanntidsapplikasjoner og maskinnær programvare. Denne laben går ut på å benytte C til å implementere et styresystem for en heis. I tillegg til selve implementasjonen, skal systemet beskrives- og dokumenteres i UML.

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 C). Disse verktøyene skal brukes på en fysisk heis-modell (se introduksjon II). Oppgavene finner dere i seksjon 1 .

Vurdering

Heis-labben vil telle på sluttkarakteren i faget. Prosjektet er konseptuelt delt i tre deler, som i utgangspunktet teller like mye:

- Dekningsgrad av kravspesifikasjonen; Factory Acceptance Test (FAT) (se appendiks B.1).
- Kvaliteten på koden dere produserer. Det vil si hvor godt ting er strukturert, og hvor vanskelig det vil være å vedlikeholde koden senere (se appendiks B.2).

- Dokumentasjon av arbeidsprosessen og de offentlige API-ene til hver modul man utvikler (se appendiks B.3).

II Introduksjon - Heisen på Sanntidslabben

Vi skal bruke en fysisk modell av en heis i løpet av prosjektet. Denne modellen består av tre hoveddeler; selve heismodellen, et betjeningspanel, og en motorstyringsboks. Det finnes en heis-modell på hver av Sanntidssalens arbeidsplasser.

II .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.

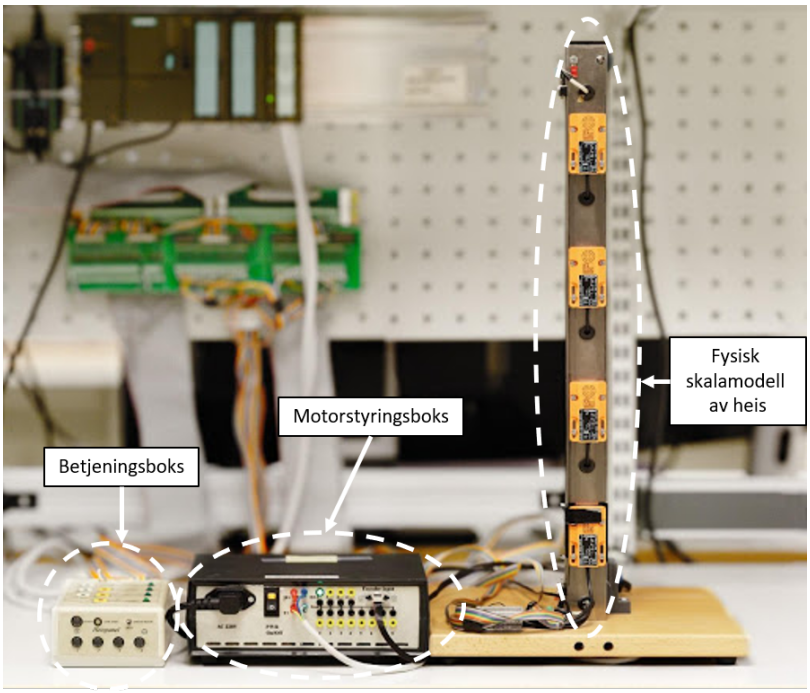

Figure 1: Heis-modellen på sanntidslaben.

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 lovlige 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.

II .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.

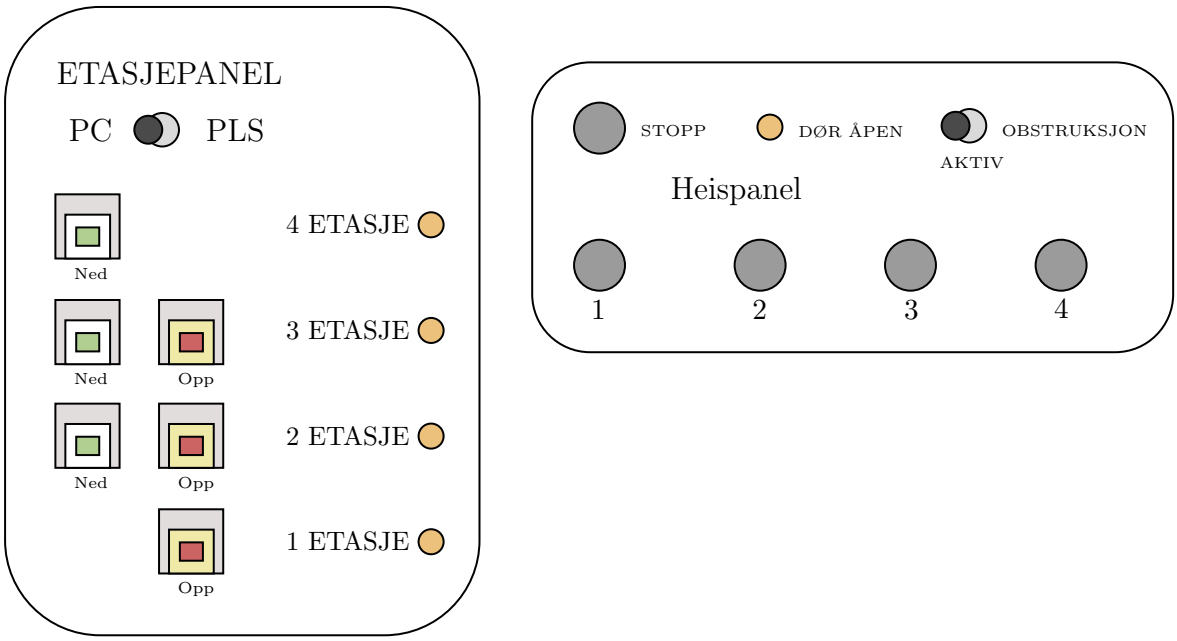

Figure 2: Etasje- og Heispanel i Sanntidslabben

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 nødstans. 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.

II .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 funksjonskall 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.

II .4 Virkemåte og oppkobling

Heis-modellen er laget for å konseptuelt oppføre seg som en virkelig heis. Det er et par punktet 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.

1 Oppgave (100%) - Heislab med dokumentasjon

I denne laben skal dere bruke V-modellen til å utvikle et styringssystem for en heis som følger spesifikasjonene gitt i appendiks A. Dere skal bruke UML til design og dokumentasjon av systemet, og selve systemet skal til slutt implementeres i C.

1 .1 Oversikt over hva som skal leveres inn

For å få full vurdering av heis-prosjektet, skal dere levere en `.zip`-fil til slutt, som skal ha følgende struktur:

```
levering.zip
├── Makefile
├── Doxygen config
├── source
│   └── .h-filer og .c-filer
└── rapport.pdf
```

hvor:

1 .1.1 Makefile

Makefilen dere leverer inn kan inneholde så mange regler dere vil, men må være i stand til å bygge prosjektet fra filene i mappen `source`.

1 .1.2 Doxygen config

Dette er konfigurasjonsfilen for Doxygen. Dere kan kalle den hva dere vil, men den må kunne lese gjennom filene i `source`, og bygge dokumentasjon ut fra disse.

Det kreves at alle offentlige APIer skal være dokumentert med kommentarer som Doxygen kan lese. Med offentlige APIer menes alle funksjoner og definerte datatyper som ligger tilgjengelig i headerfilene (`.h`).

1 .1.3 source

I mappen `source` putter dere all kode som er nødvendig for å bygge heisprogrammet. Dette inkluderer utleverte drivere.

1 .1.4 rapport.pdf

Det skal skrives en kort rapport (max 15 sider) som dokumenterer design-valg og arbeidet dere har gjort. Denne skal ikke være veldig omfattende, men det er hovedsaklig denne som bestemmer dokumentasjons-scoren i heisprosjektet.

Rapporten skal "speile" V-modellen og skal inneholde følgende seksjoner:

- **Overordnet arkitektur**: Beskriver styresystemet sin arkitektur på et høyt nivå. Denne seksjonen skal inneholde et **klasse-diagram** som viser hver av modulene som inngår i designet, og hvilke relasjoner som finnes mellom dem.

  For å illustrere hvordan de forskjellige modulene fungerer sammen, skal denne seksjonen også inneholde et **sekvensdiagram** som viser denne sekvensen:

  1. Heisen står stille i 2. etasje med døren lukket.
  2. En person bestiller heisen fra 1. etasje.
  3. Når heisen ankommer går personen inn i heisen og bestiller 4. etasje.
  4. Heisen ankommer 4. etasje, og personen går av.
  5. Etter 3 sekunder lukker dørene til heisen seg.

  Utover dette skal denne seksjonen også ha et **tilstandsdiagram** som viser hvordan heisen oppfører seg basert på hvilke input som kommer inn, og hvilke output heisen selv setter.

  Til slutt skal dere argumentere for hvorfor deres arkitekturvalg har noe for seg. Prøv å ikke overkompliser argumentasjonen; dere trenger trenger ikke å komme på argumenter som ingen andre har tenkt på før, men dere må vise at dere har tenkt gjennom valgene dere har gjort på en fornuftig måte.

- **Moduldesign**: Her går dere kort over modulene deres i mer detalj. Dere skal argumentere for implementasjonsvalg av interesse. Om dere for eksempel lagde en kømodul som bruker lenkede lister, så er dette en interessant implementasjonsdetalj. Argumentasjonen bør fokusere på hva som gir "minst hodebry" for andre utviklere.

Eksempelvis er det ikke interessant med "Vi bruker lenkede lister fordi de teoretisk kan oppføre seg raskere enn dynamiske arrays, om man må gjøre mye innsettinger". Hastighet er *nesten* irrelevant ettersom vi skriver i C, hvor det meste går fort nok uansett.

Et **eksempel** på bedre argumentasjon er: "Tidsbiblioteket vårt lagrer verdien på tiden selv. I dette tilfellet er dette et lurt valg, fordi vi kun trenger en timer, så det er ingen fare for kollisjoner i tidsstempling. I tillegg til dette slipper moduler som kaller tidsbiblioteket å huske på tiden mellom hvert kall, som ellers ville ført til tettere kobling mellom modulene".

- **Testing**: Dere står fritt til å dele opp denne seksjonen i enhetstesting- og integrasjonstesting om dere har forskjellige fremgangsmåter for dem. Uansett skal denne seksjonen overbevise leseren om at heisen faktisk fungerer. Dere skal beskrive hvorfor dere har tro på at dere har oppfylt kravene i spesifikasjonen.

  Eksempel: "For å teste køsystemet kjørte vi det gjennom GDB, hvor vi satte- og fjernet bestillinger mens vi sammenlignet med forventet oppførsel".

  I tillegg bør dere også inkludere deres egen testprosedyre for systemet, for **eksempel**: "Vi starter heisen mellom to etasjer, og ser at den kjører ned til etasjen under. Dette sikerer at heisen er i en definert tilstand, og sikrer dermed at punkt 01 er tilfredsstilt".

- **Diskusjon**: Dere kommer sannsynligvis til å oppdage svakheter i deres egen implementasjon. Her skal dere prøve å identifisere slike aspekter, og forklare hvordan systemet kan omdesignes og forbedres.

  I tillegg er det også viktig å spekulere i andre valg dere kunne ha tatt i løpet av prosjektet, og hvilke betydninger det ville ha fått for arkitektur, implementasjon, testing, vedlikeholdbarhet, etc.

  God diskusjon kan gjøre opp for små ting som ellers ville gitt et negativt inntrykk fra arkitekturfasen, så lenge det finnes gode grunner for at dere endte opp med valgene dere gjorde.

  Eksempel: "Måten køsystemet husker bestillinger på viste seg å være mer komplisert enn nødvendig. Allikevel mener vi at køsystemet har et minimalt grensesnitt mot resten av programmet, og har lav grad av kobling til de andre delene programmet består av. I fremtiden betyr dette at innmaten til køsystemet kan skrives om uten at resten av programmet må endre seg nevneverdig. På denne måten argumenterer vi for at køsystemets kompleksitet er begrenset til kun seg selv, og derfor ikke reduserer den helhetlige kvaliteten til programmet på en måte som lett smitter andre moduler".

A Appendiks - Heisspesifkasjoner

A.1 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 fjerde etasje, eller under første etasje idet systemet skrus på.

A.2 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.

A.3 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.

A.4 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.

A.5 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 første til fjerde 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.

A.6 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.

A.7 Tillegg

Punkt	Beskrivelse
Y1	Oppførsel som ikke er "vanlig heisoppførsel" 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 Appendiks - Kommentar til vurdering

B.1 FAT - Factory Acceptance Test

FAT er sluttprøven som bestemmer i hvilken grad dere har implementert et korrekt system. FATen er en direkte gjenspeiling av kravspesifikasjonen fra seksjon 1 , så om dere oppfyller alle kravene som er satt av den, har dere implementert et fullverdig system.

B.2 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.

Variabelnavn på en bokstav (med unnstak av indeks-variabler) er et dårlig tegn. Det er også "generiske" navn som "`handler`" eller "`supervisor`". Det aller viktigste er å være så presis som mulig.

B.3 Dokumentasjon

Når det gjelder dokumentasjon, er det oftest å følge **KISS** (**K**eep **I**t **S**imple, **Stupid**). Størrelsen på sekvensdiagrammet er ikke proporsjonalt med hvor bra det er. Det er mye viktigere hvordan man bruker sekvensdiagrammet for å formidle tanken bak systemet til en som ikke har vært involvert.

Når det kommer til alt av argumentasjon: Ikke gjør det vanskelig uten grunn. Bare forklar hvorfor dere tok valgene dere gjorde, og hvilke følger det fikk for resten av prosjektet. Et tips er at noen alltid noterer ned hva dere har gjort hver eneste dag, slik at det blir lettere å finne fram til de spesifikke valgene som dere gjorde opp til vurdering.

C Appendiks - V-modellen

V-modellen er illustrert i figur 3 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".

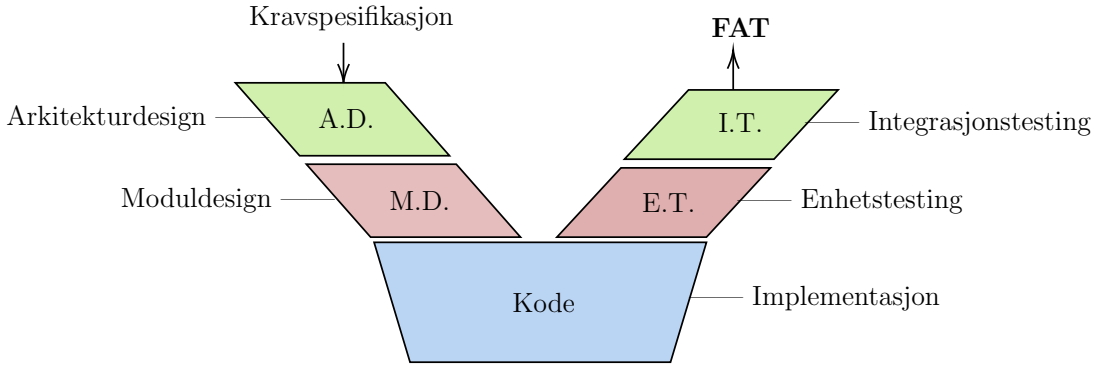

Figure 3: Illustrasjon av V-modellen.

C.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 implementerert 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.

C.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.

C.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 primitivene 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 primitivene et språk støtter direkete, 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 "dette kan implementeres som en klasse som arver fra en annen".

C.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: Det kan være mer fornuftig å lage en test som sjekker om en modul for å håndtere bestillinger bare legger inn bestillinger for etasjenummer 1-4 enn å lage en test som sjekker hvordan modulen responderer på negative tall, en test som sjekker hvordan den reagerer på bestillinger til etasjer høyere enn 4, en test som sjekker hvordan den responderer på bestilling til etasje 0 osv.

C.5 Integrasjonstesting

Enhetstesting foregår på modul-nivå og svarer på spørsmålet: "Fungere 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* (lb s), mens NASA sine moduler opererte med *newton per sekund* (N s). Manglende integrasjonstesting endte totalt opp med å koste NASA JPL omlag 330 millioner amerikanske dollar.

D Appendiks - Kodekvalitet

For deres egen del bør dere 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 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 nano-sekunder 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.

D.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 bær 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 medlemsdata er definert etter en eventuell initialiseringsfunksjon.

D.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.

D.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.

D.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.

D.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.
- Selv om C er forholdsvis lavnivå er det fullt mulig å ivareta god softewareutviklingspraksis. 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.

E Appendiks - Heissimulator

I tillegg til den fysiske heisen, har vi en heissimulator[1] 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

[1] Skrevet av Torjus Bakkene og Erlend Blomseth

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).

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. Endre Makefilen slik at `SIM := true` (for å teste heisen på den fysiske modellen, må dere endre makefilen tilbake slik at `SIM := false`!).
5. Åpne en annen terminal i `skeleton_project` mappen
6. 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 4. I dette grensesnittet, symboliserer `*` at knappen, enten inne, eller ute har blitt aktivert. Dette tilsvarer når man trykker på de fysiske knappene i etasje- og heispanelene i den fysiske heisen på sanntidssalen.

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 5 (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 5 for hvilke tastetrykk som tilsvarer hva på simula-

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

Figure 4: Grensesnittet til heissimulatoren.

toren):

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

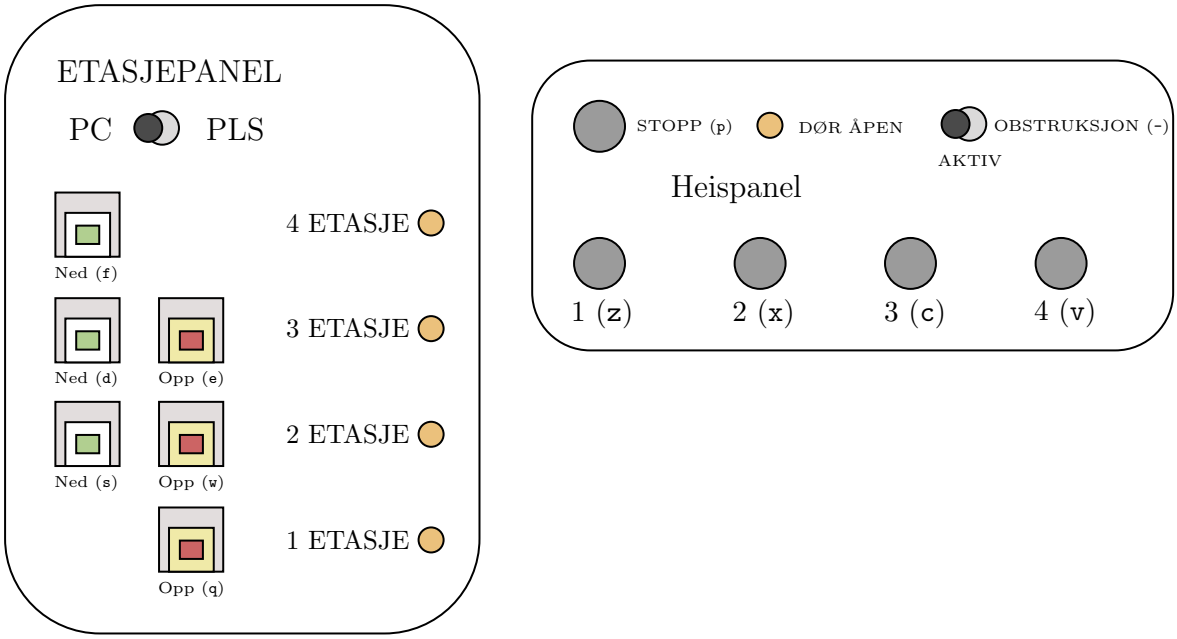

Figure 5: Etasje- og Heispanel i heissimulatoren.