Tilpassede Datasystemer
Heisprosjekt

Institutt for teknisk kybernetikk
NTNU

Revisjonshistorie

2014	Øyvind Stavdahl
2014	Anders Rønning Petersen
2016	Øyvind Stavdahl
2016	Konstanze Kölle
2017	Ragnar Ranøyen Homb
2017	Bjørn-Olav Holtung Eriksen
2018	Kolbjørn Austreng
2019	Kolbjørn Austreng
2020	Kolbjørn Austreng

1 Introduksjon

1.1 Motivasjon

Programmeringsspråket C er et slagkraftig verktøy som regelmessig benyttes i et bredt spekter industrisammenhenger, særlig i sanntidsapplikasjoner og maskinnær programvare.

Dette prosjektet 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 prosjektet 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. Denne er beskrevet i seksjon 5.

1.2 Vurdering

Heisprosjektet vil telle på sluttkarakteren i faget. Prosjektet er konseptuelt delt i tre deler, som i utgangspunktet (før eventuelle sluttjusteringer) teller like mye:

- Dokumentasjon av arbeidsprosessen og de offentlige APIene til hver modul dere utvikler.
- Kvaliteten på koden dere produserer. Det vil si hvor godt ting er strukturert, og hvor vanskelig det vil være å vedlikeholde koden senere.
- Dekningsgrad av kravspesifikasjonen; FAT.

For noen kommentarer til vurderingen, se seksjon 4.

1.3 Sammenheng med øvingene i faget

Øvingene er ikke formelt en del av prosjektet, og de teller som sådan ikke på vurderingen dere får i heisprosjektet, i hvert fall ikke direkte. Når det er sagt, så vil det fortsatt uten tvil lønne seg å bruke øvingene flittig i løpet av prosjektet.

Om dere allikevel skulle miste sikt med den sanne vei, og vandre inn på stier med `printf` for debugging og mappenavn som "`elevator_03_works`" og "`elevator_04_queue_broken`", så ber jeg dere gjenta dette mantraet:

> "Git og GDB er sykt porno."
>
> - Kolbjørn

1.4 Utstyrsbeskrivelse

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 heismodell på hver av Sanntidssalens arbeidsplasser.

1.4.1 Heismodell

Heismodellen er illustrert til høyre i bildet på forsiden av dette dokumentet, og består av en heisstol som kan beveges opp og ned langs en stolpe. Dette tilsvarer henholdsvis heisrommet- og sjakten i en virkelig heis.

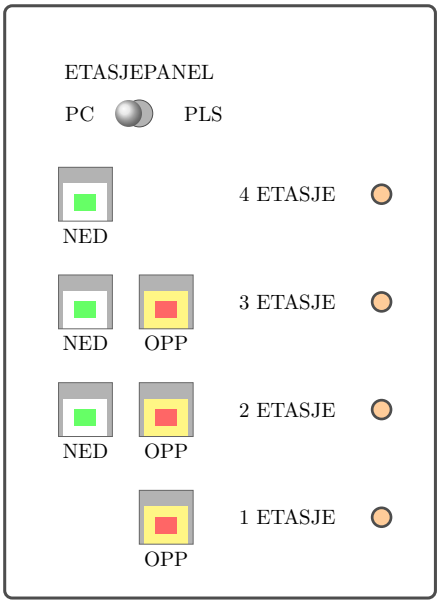
Langs heisbanen er det montert fire Hall-effektsensorer som fungerer som heisens etasjer. Over øverste etasje, og under nederste etasje er det også montert endestoppbrytere, som vil kutte motorpådraget dersom heisen kjører utenfor sitt lovlige område. Dette er for å beskytte heisens motor mot skade.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

1.4.2 Betjeningsboks

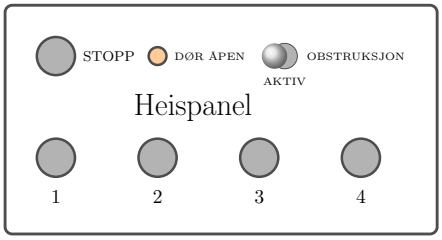
Betjeningsboksen er delt i to; *etasjepanel* og *heispanel*. Disse er illustrert i figur 1 og figur 2. Øverst på betjeningsboksen finnes en bryter som velger om datamaskinen eller PLSen skal styre heismodellen. Denne skal stå i "PC" gjennom hele oppgaven.

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 det inkludert et ekstra 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.



Figur 1: Etasjepanel.



Figur 2: Heispanel.

1.4.3 Motorstyringsboks

Styringsboksen er ansvarlig for å forsyne effekt til heismodellen, og for å forsterke pådraget som settes av datamaskinen. 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 tachosignal, samt en digital verdi for motorens encoder, for å lese av hastighet- og posisjon fra styringsboksen. Disse målesignalene trenger dere ikke ta stilling til i dette prosjektet, men de nevnes for fullstendighetens skyld.

1.5 Virkemåte og oppkobling

Heismodellen 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-effektsensoren i hver etasje og tilhørende etasjeindikator.
- Om endestoppbryterne aktiveres vil pådrag til heisen kuttes. Om det skjer må heisstolen manuelt skyves vekk fra endestopp.
- Rød og blå ledning forsyner effekt til motoren. Henholdsvis kobles disse til $M+$ og M- som dere finner på motorstyringsboksen.

2 Kravspesifikasjon

Kravspesifikasjonen i denne seksjonen beskriver punktene styringskoden skal oppfylle. Ved uklarheter kan dere betrakte vitass eller foreleser som kunden av heissystemet, og spørre om oppklaring.

2.1 Oppstart

Punkt	Beskrivelse
O 1	Ved oppstart skal heisen alltid komme til en definert tilstand. En definert tilstand betyr at styresystemet vet hvilken etasje heisen står i.
O 2	Om heisen starter i en udefinert tilstand, skal heissystemet ignorere alle forsøk på å gjøre bestillinger, før systemet er kommet i en definert tilstand.
O 3	Heissystemet skal ikke ta i betraktning urealistiske start-betingelser, som at heisen er over fjerde etasje, eller under første etasje idet systemet skrus på.

2.2 Håndtering av bestillinger

Punkt	Beskrivelse
H 1	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.
H 2	Heisen skal ikke betjene bestillinger fra utenfor heisrommet om heisen er i bevegelse i motsatt retning av bestillingen.
H 3	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.
H 4	Om heissystemet ikke har noen ubetjente bestillinger, skal heisen stå stille.

2.3 Bestillingslys og etasjelys

Punkt	Beskrivelse
L 1	Når en bestilling gjøres, skal lyset i bestillingsknappen lyse helt til bestillingen er utført. Dette gjelder både bestillinger inne i heisen, og bestillinger utenfor.

L 2	Om en bestillingsknapp ikke har en tilhørende bestilling, skal lyset i knappen være slukket.
L 3	Når heisen er i en etasje skal korrekt etasjelys være tent.
L 4	Når heisen er i bevegelse mellom to etasjer, skal etasjelyset til etasjen heisen sist var i være tent.
L 5	Kun ett etasjelys skal være tent av gangen.
L 6	Stoppknappen skal lyse så lenge denne er trykket inne. Den skal slukkes straks knappen slippes.

2.4 Døren

Punkt	Beskrivelse
D 1	Når heisen ankommer en etasje det er gjort bestilling til, skal døren åpnes i 3 sekunder, for deretter å lukkes.
D 2	Om heisen ikke har ubetjente bestillinger, skal heisdøren være lukket.
D 3	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.
D 4	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.

2.5 Sikkerhet

Punkt	Beskrivelse
S 1	Heisen skal alltid stå stille når døren er åpen.
S 2	Heisdøren skal aldri åpne seg utenfor en etasje.
S 3	Heisen skal aldri kjøre utenfor området definert av første- og fjerde etasje.
S 4	Om stoppknappen trykkes, skal heisen stoppe momentant.
S 5	Om stoppknappen trykkes, skal alle heisens ubetjente bestillinger slettes.

S 6	Så lenge stoppknappen holdes inne, skal heisen ignorere alle forsøk på å gjøre bestillinger.
S 7	Etter at stoppknappen er blitt sluppet, skal heisen stå i ro til den får nye bestillinger.

2.6 Robusthet

Punkt	Beskrivelse
R 1	Obstruksjonsbryteren skal ikke påvirke systemet når døren ikke er åpen.
R 2	Det skal ikke være nødvendig å starte programmet på nytt som følger av eksempelvis udefinert oppførsel, at programmet krasjer, eller minnelekkasje.
R 3	Etter at heisen først er kommet i en definert tilstand ved oppstart, skal ikke heisen trenge flere kalibreringsrunder for å vite hvor den er.

2.7 Ymse

Punkt	Beskrivelse
Y 1	Oppførsel som ikke er "vanlig heisoppførsel" kan gi trekk på FATen. Ikke tenk for komplisert her. Dere har alle tatt en heis før, og vet hvordan de bør funke.

Legg merke til punk Y 1 i spesifikasjonen. Hvis heisen på en eller annen måte "slanger seg innom" de andre kravene, men oppfører seg helt tullete, så *vil* dette gi trekk.

Når det er sagt, så er det ingen som er "ute for å ta dere". Bare bruk sunn fornuft, og eventuelt spør vitass eller foreleser om noe er uklart.

3 Oppgave

Kort oppsummert skal dere bruke V-modellen, som er beskrevet i seksjon 5, til å implementere et styresystem for en heis som følger spesifikasjonen i seksjon 2.

Seksjon 3.1 beskriver alt som skal leveres.

3.1 Hva skal leveres inn?

Alt i denne seksjonen skal leveres inn for å få full vurdering av heisprosjektet. Dere skal levere én `.zip`-fil til slutt, som skal ha følgende struktur:

```
levering.zip
  |-- Makefile
  |-- Doxygen config
  |
  |-- source
  |   |-- .h-filer og .c-filer
  |
  |-- rapport.pdf
```

Vær så snill, lever én `.zip`-fil til slutt.

3.1.1 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`".

3.1.2 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 fra disse.

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 headerfiler (`.h`-filer).

3.1.3 source

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

3.1.4 rapport.pdf

Det skal skrives en kort rapport som dokumenterer designvalg og arbeidet. Denne er ikke tenkt å være "ekstremt omfattende", men det er hovedsakelig denne som bestemmer dokumentasjonsscoren i heisprosjektet, så det er lurt å gjøre den litt forseggjort.

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

- **Overordnet arkitektur**: Beskriver styresystemet sin arkitektur på et høyt nivå. Denne seksjonen skal inneholde et **klassediagram** som viser hver av modulene som inngår i designet deres 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. Ikke overkompliser argumentasjonen; dere trenger ikke komme på argumenter som ingen andre har tenkt på før, men dere må vise at dere har tenkt gjennom valgene dere har gjort, og forhåpentligvis landet på noe fornuftig.

- **Moduldesign**: Her går dere kort over modulene deres i mer detalj. Her skal dere 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 er "minst hodebry" for andre utviklere.

Eksempelvis er det ikke interessant med "Vi bruker lenkede lister fordi de teoretisk kan oppføre seg raskere enn dynamiske arrayer om man må gjøre mye innsettinger". Hastighet er *nesten* irrelevant. Dere skriver i C; 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 én 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.

  Et eksempel på noe som godt kan stå i denne seksjonen er "For å teste køsystemet kjørte vi det gjennom GDB, hvor vi satte- og fjernet bestillinger mens vi sammenlignet med forventet oppførsel".

  Dere bør 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 sikrer at heisen er i en definert tilstand, og sikrer dermed at punkt **O 1** er tilfredsstilt".

- **Diskusjon**: Dere kommer sannsynligvis til å oppdage svakheter eller små smertepunkter i deres egen implementasjon. Her skal dere prøve å identifisere slike aspekter ved prosjektet, og forklare hvordan systemet kan omdesignes for å buktes med problemet.

  Dere kan også 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. Et eksempel er "Måten køsystemet husker bestillinger på viste seg å være mer komplisert enn nødvendig. Allikevel mener vi 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".

Legg merke til at *til og med* tittelen på denne seksjonen forteller hvilket format rapporten skal ha; den skal være en PDF.

Dere kan skrive i hva dere vil så lenge dere får en PDF ut av det til slutt, men LaTeX anbefales.

Ved tvil om hva dere skal levere, og hvilket format dere skal levere i, vær vennlig å les denne seksjonen igjen, eller spør vitass eller foreleser.

4 Kommentarer til vurdering

4.1 Factory Acceptance Test

Factory Acceptance Test er sluttprøven som bestemmer i hvilken grad dere har implementert et korrekt system. FATen er en direkte gjenspeiling av kravspesifikasjonen fra seksjon 2, så om dere oppfyller alle kravene som er satt av den, har dere implementert et fullverdig system.

4.2 Kodekvalitet

Kort fortalt er kodekvalitet hovedsakelig et mål på hvor "åpenbar" koden deres er. Koden er "åpenbar" om dere 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 én bokstav (med unntak av indeksvariabler) er et dårlig

tegn. Det er også "generiske" navn som "`handler`" eller "`supervisor`".

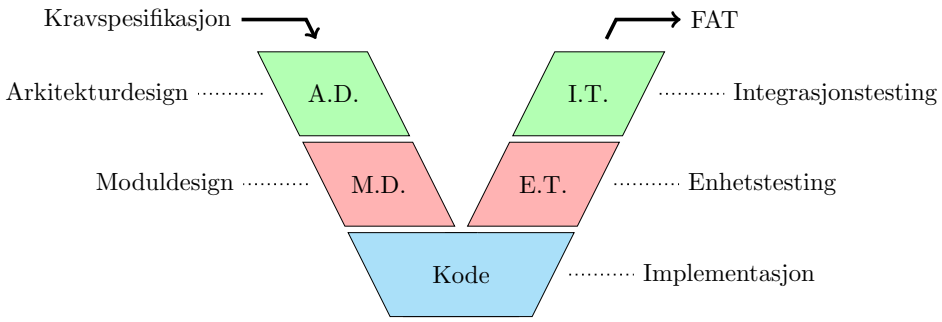
Seksjon 6 tar for seg hva vi anser som *god kode* i dette faget. Om dere følger anbefalingene der, eller kommer på noen bedre, så er dere godt plantet på trygg mark.

4.3 Dokumentasjon

Ikke gjør ting vanskeligere enn de strengt talt trenger å være. Størrelsen på sekvensdiagrammet ditt forteller ikke hvor bra det er. Det er mye viktigere hvordan du bruker sekvensdiagrammet ditt.

Når det kommer til alt av argumentasjon: Igjen, ikke gjør det vanskelig uten grunn. Bare forklar hvorfor dere tok valgene dere gjorde, og hvilke følger det fikk for resten av prosjektet.

5 V-modellen



Figur 3: Illustrasjon av 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 som "funket i går".

Hver av stegene i V-modellen er beskrevet under.

5.1 Arkitekturdesign

Dere bør først forsikre dere om at dere forstår kravene i spesifikasjonen. Når dere er sikre på at dere vet hva som kreves av sluttsystemet, tenker dere gjennom hvilke implikasjoner dette har for koden dere skal skrive senere.

På dette stadiet ønsker vi å bestemme en *arkitektur* som vil oppfylle kravene fra spesifikasjonen. Det 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". Detaljene om hvordan et eventuelt køsystemet er implementert kommer ikke inn i bildet på dette stadiet.

Resultatet av dette stadiet bør være ett eller flere klassediagrammer som illustrerer hvilke moduler styringssystemet skal bestå av. For å få en idé om hvilken funksjonalitet hver modul må tilby kan det også være lurt å sette opp et par sekvensdiagrammer som illustrerer hvordan forskjellige moduler samarbeider.

Om dere føler behovet, kan dere også benytte kommunikasjonsdiagrammer for å gi dere selv en bedre oversikt over grensesnittet mellom hver modul.

5.2 Moduldesign

Når dere har en overordnet tanke om 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 tilstand 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 husker ting mellom hvert funksjonskall 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 trenger hukommelse i en dedikert tilstandsmaskin, og beholde hjelpemoduler så enkle som mulig.

Her bør dere benytte dere 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.

5.3 Implementasjon

Det er på dette stadiet dere skriver kode. Hvis dere har lagt inn en grei innsats i designfasen, vil dette stadiet koke ned til å oversette diagrammene til kode. Så feilfritt går det selvsagt aldri, men et godt forarbeid kan spare dere for mye unødvendig hodebry. Vær heller ikke redd for å gå tilbake for å endre på arkitekturen eller modulsammensetningen om dere 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 klassediagram som en egen modul, hvor alle funksjonene som modulen gjør tilgjengelig svarer til offentlige medlemsfunksjoner i en klasse, og så videre.

På den annen side er det selvsagt en fordel å benytte seg av de primitivene et språk støtter direkte, fremfor å prøve å tvinge inn funksjonalitet som ikke

gis av språket[1], 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".

5.4 Enhetstesting

Enhetstesting speiler moduldesignfasen. Her gjør dere tester som forsikrer dere om at hver modul oppfører seg som den skal. I første omgang er det greit å gjøre små, veldig veldefinerte tester, som tester ut én 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. Randtilfeller ("Border cases") er stort sett en langt større kilde til feil enn vanlige tilfeller, så det er altså mye mer verdifullt med *tester som tester forskjellige ting* enn med *forskjellige tester som tester samme ting.*

Eksempel:

> "A test engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.
>
> First real customer walks in asks where the bathroom is. The bar bursts into flames, killing everyone."

5.5 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 denne andre modulen?". Her vil dere typisk prøve ut hele- eller nesten hele programmet på en spesifikk funksjonalitet. Om dere har tatt dere tid til å lage fornuftige sekvensdiagrammer, kommer disse godt med i denne fasen.

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

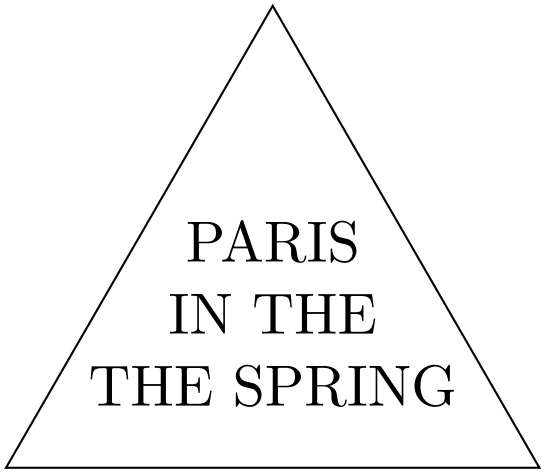
Eksempel: 23. September 1999 ble *Mars Climate Orbiter* tapt etter at fartøyet enten gikk i stykker i Mars' atmosfære, eller spratt tilbake til en

[1] "You can write FORTRAN in any language"

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* ($\mathrm{lb} \cdot \mathrm{s}$), mens NASA sine moduler opererte med *newton per sekund* ($\mathrm{N} \cdot \mathrm{s}$). Manglende integrasjonstesting[2] endte totalt opp med å koste NASA JPL omlag 330 millioner amerikanske dollar.

5.6 Kommentar til testing

Når dere tester er det lett å tro at man har testet alt det går an å teste, eller å tro at en gitt del av kodebasen *må* være feilfri. Som mennesker har vi også en ulempe når det kommer til testing - vi har en tendens til å oppfatte bare det vi ønsker å se, eller gjøre en subjektiv farging av input vi får. Ta en titt på figur 4 for en illustrasjon.



Figur 4: Hva er det som står her?

[2] Og enheter som kun brukes i U-land ¯_(ツ)_/¯

6 Kodekvalitet

For deres egen del bør dere ha kodekvalitet i bakhodet hele tiden. Hva som er god kode finnes det mange meninger om, men felles for dem alle er at de gir lesbar kode som er lettere å vedlikeholde enn annen kode.

Ute i virkelige settinger bruker man betraktelig 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 ligger 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 dere ha en bedre konvensjon selv, og dere bør i alle fall være helt konsekvente.

6.1 Moduler

- Alle funksjonene i en modul bør ha samme abstraksjonsnivå. Ikke bland lavnivå funksjonalitet med høynivå 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 dere kunne bruke modulen uten å ha den ringeste anelse om hvordan innmaten dens ser ut.
- 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 modulfunksjoner logisk.
- 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 én modul uten å måtte endre på alle andre som inngår i programmet.
- For klasser er det ønskelig at alle medlemsvariabler er definerte etter at konstruktøren har kjørt. Tilsvarende, for moduler er det ønskelig at all medlemsdata er definert etter en eventuell initialiseringsfunksjon.

6.2 Funksjoner

- Den viktigste grunnen til å opprette en funksjon er ikke kodegjenbruk, men å gi dere en måte å håndtere kompleksitet på. Det kan fint hende

at dere kommer over tilfeller hvor det 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 - én 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" som pesten.
- 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.
  - **Kommunikasjonskohesjon** er når en funksjon benytter samme data til å gjøre forskjellige ting, men hvor bruken av dataen 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 én og samme ting. Tingene funksjonen gjør er altså nødvendige for å utføre en bestemt oppgave.

  Av disse er funksjonell kohesjon den mest ønskelige.
- Motsatte verb bør være presise og opptre i veldefinerte par som "begin - end", "create - destroy", "open - close" eller "next - previous".
- Funksjoner har også en kobling mellom hverandre i den forstand at de avhenger av returverdien til andre funksjoner. Denne koblingen bør være så løs som mulig. Om dere endrer én funksjon skal det ikke være nødvendig å endre mange andre.

6.3 Variabler

- Unngå å bruke for mange "arbeidsvariabler" - 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 iterere en løkke kan fint hete "i", mens en global variabel bør ha et virkelig godt navn.

6.4 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 esoterisk 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.

6.5 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 modulvariabler begrenset med kodeordet `static`, og `g_` om de globale.

- Selv om C er forholdsvis lavnivå er det fullt mulig å ivareta god softwareutviklingspraksis. Allikevel kommer man alltid til å skrive noe "ondskapsfull" 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 heller vil følge, så lenge dere kan argumentere for at de bedre ivaretar lesbarhet og vedlikeholdbarhet.