Hovedprosjekt:

KONTROLLSYSTEM FOR AUTOKLAV

Forfatter(e):

Audun Klundby
Asbjørn Konstad
Hans Einar Øverjordet

Dato:

Gjøvik 24.05.2006

Sammendrag av hovedprosjekt

Tittel:	Kontrollsystem for autoklav	Nr.:	4
		Dato:	22.05.06
Deltaker(e):	Audun Klundby Asbjørn Konstad Hans Einar Øverjordet		
Veileder(e):	Tom Røise, Høgskolen i Gjøvik		
Oppdragsgiver:	Automatikk & Hygiene Partner AS		
Kontaktperson:	Bård Marken		
Stikkord (4 stk)	Autoklav, Prosesstyring, C++, Multithreading		
Antall sider: 93	Antall bilag: 10	Tilgjengelighet: Åpen	

Kort beskrivelse av hovedprosjektet:

Autoklaver benyttes i medisinsk sammenheng for å sterilisere utstyr. Prosjektoppgaven har bestått i å lage et komplett styringssystem for en autoklav utviklet av Automatikk & Hygiene Partner AS.

Systemet benytter tredeparts kontroller for kommunikasjon med digitale og analoge inn/utganger for direkte styring av autoklavens ventiler, sensorer og pumper.

Utfordringen med prosjektet har vært å kombinere automasjons- og datateknologier – noe som det ikke finnes så mye kompetanse på fra før – og integrere disse i en løsning som drives av mye sanntidsdata i et multithreaded miljø.

Systemet kontrollerer, konfigurerer og styrer prosesser for sterilisering av utstyr og lagrer sanntidsdata for alle aktiviteter knyttet til autoklaven.

Systemet er utviklet i Microsoft Visual Studio .NET – managed C++ - for Windows XP embedded.

Forord:

Prosjektoppgaven ble presentert ved Høgskolen i Gjøvik november 2005. Det fascinerende ved oppgaven var kombinasjonen av data –og automasjonsteknologi. Dette er områder ingen i prosjektgruppa hadde erfaring med fra før og vi så det som en spennende utfordring rent faglig sett.

Etter kort møte med oppdragsgiver og en liten idéutveksling ble vi tildelt prosjektet. Arbeidet startet allerede i desember med studie av kontrollerenhet som skulle benyttes for styring av autoklaven. I januar 2006 startet arbeidet for fullt og alle deltagere i gruppa har jobbet målbevisst og jevnt helt frem til innlevering.

Prosjektet har bydd på mange spennende problemstillinger som har krevd mye tid og arbeid. Det har imidlertid vært meget lærerikt og alle gruppedeltagere har fått verdifullt faglig utbytte som vi tar med oss videre.

Veileder for prosjektet har - etter ønske - vært Tom Røise. Vi vil takke for et meget godt samarbeid og utmerket råd og veileding.

Kontaktperson hos oppdragsgiver har vært Bård Marken. Vi takker for godt samarbeide, støtte og oppfølging gjennom utviklingen av systemet.

Gjøvik 24.05.2006

Audun Klundby *Asbjørn Konstad* *Hans Einar Øverjordet*

____________________ ____________________ ____________________

Innholdsfortegnelse:

1. Innledning 6
   1.1 Målgruppe for rapporten 6
   1.2 Formål 6
   1.3 Prosjektgruppas bakgrunn 7
   1.4 Arbeidsmetoder 7
   1.5 Organisering av rapporten 9
2. Fagteori og utstyr 11
   2.1 Steriliseringsprosessen 11
   2.2 Autoklavens oppbygning 11
   2.3 Autoklavens funksjon 11
   2.4 Hardware grensesnitt 11
3. Kravspesifikasjon 12
   3.1 Introduksjon 12
   3.2 Overordnede funksjonelle krav 13
   3.3 Detaljerte funksjonelle krav 20
   3.4 Supplementær spesifikasjon 20
4. Design 23
   4.1 Introduksjon 23
   4.2 Logisk inndeling 24
   4.3 Tekniske notater 33
   4.4 Process view 40
   4.5 Implementation view 43
   4.6 Data view 46
5. Implementasjon 50
   5.1 AH::Pride::Domain 50
   5.2 AH::Pride::Domain::Processes 52
   5.3 AH::Pride::IO 56
   5.4 AH::Pride::Communication 58
   5.5 AH::Pride::Config 62
   5.6 AH::Pride::GUI 63
   5.7 GUI::RAW 67
6. Test 71
   6.1 Uførte tester 71
   6.2 Fremtidige tester 72
7. Avslutning 74
   7.1 Evaluering av oppgaven 74
   7.2 Gruppearbeidet 74
   7.3 Videre arbeid 75
   7.4 Konklusjon 76
Vedlegg A. Autoklavens oppbygning 77
Vedlegg B. Autoklavens Funksjon 78
Vedlegg C. Hardware grensesnitt 79
Vedlegg D. Detaljerte funksjonelle krav 80
Vedlegg E. Timelogg 81
Vedlegg F. Møtereferat (eksempel) 83
Vedlegg G. Skjermbilder 84
Vedlegg H. Forprosjekt (uten vedlegg) 85

H.1 Mål og rammer 86
H.2 Omfang 87
H.3 Prosjektorganisering 88
H.4 Planlegging, oppfølging og rapportering 89
H.5 Organisering av kvalitetssikring 90
H.6 Plan for gjennomføring 91
Vedlegg I. Planlagt prosjektplan 92
Vedlegg J. Faktisk prosjektplan 93

1. Innledning

En autoklav er en spesialkonstruert 'trykk-koker' laget for å sterilisere utstyr. Autoklaver benyttes i alle institusjoner og organisasjoner som har behov for å sterilisere utstyr og produkter. Næringsmiddelindustrien benytter autoklaver for å sterilisere mat som skal hermetiseres. Tatoveringsstudioer benytter autoklaver for å sterilisere nåler og utstyr for tatovering. Sykehus benytter autoklaver for destruering av smitteavfall og sterilisering av skalpeller, sakser, nåler og lignende for operasjonssaler. Laboratorier benytter autoklaver for å sterilisere for og smittefarlig avfall i forbindelse med forsøksdyr.

Alle bakterier, virus, mikrober og sporer destrueres ved at en kombinasjon av temperatur og trykk opprettholdes over tid.

Automatikk & Hygiene Partner AS leverer blant annet autoklaver, laboratoriemateriell og måleutstyr til laboratorier og helseinstitusjoner. Til tross for samarbeid med store aktører innen steriliseringsutstyr ser Automatikk & Hygiene Partner AS at det eksisterer et potensial for forbedringer av funksjonalitet på eksisterende produkter.

Automatikk & Hygiene Partner AS har derfor startet utviklingen av ny og forbedret teknologi i forbindelse med konstruksjon, operasjon og styring av autoklaver.

1.1 Målgruppe for rapporten

Målgruppe for denne prosjektrapporten prosjektets veileder og sensor, studenter og lærere ved Høgskolen i Gjøvik, oppdragsgiver og oppdragsgivers samarbeidspartnere.

1.2 Formål

AH Partners ønsker ved prosjektet er å oppnå en sterkere markedsposisjon ved å kunne levere en nyutviklet autoklav med et brukervennlig styringssystem. De ønsker også å senke kostnadene ved utvikling av autoklaven ved å benytte et egenutviklet styresystem.

Målet for prosjektet var å utvikle et komplett styringssystem for autoklaven. Systemet skulle erstatte eksisterende styringsmetoder som var basert på PLS (Programmerbar Logisk Styring). Systemet skulle innfri de krav og normer som er gitt for prosessforløpet ved sterilisering av utstyr.

Gruppens resultatmål var å etablere en stabil plattform for systemet samt å utvikle komplett og feilfri programvare for pc-basert styring av autoklaven med fokus på stabilitet og brukervennlighet. Systemet er utviklet for direkte konfigurering, styring og overvåking av autoklaven. Alle prosesser dokumenteres og systemet kan produsere detaljert informasjon til printere så vel som andre eksterne systemer. Løsningen er laget skalerbar med tanke på fremtidige muligheter til å variere valg av hardware samt mulighet for gjenbruk i andre maskinkonfigurasjoner.

1.3 Prosjektgruppas bakgrunn

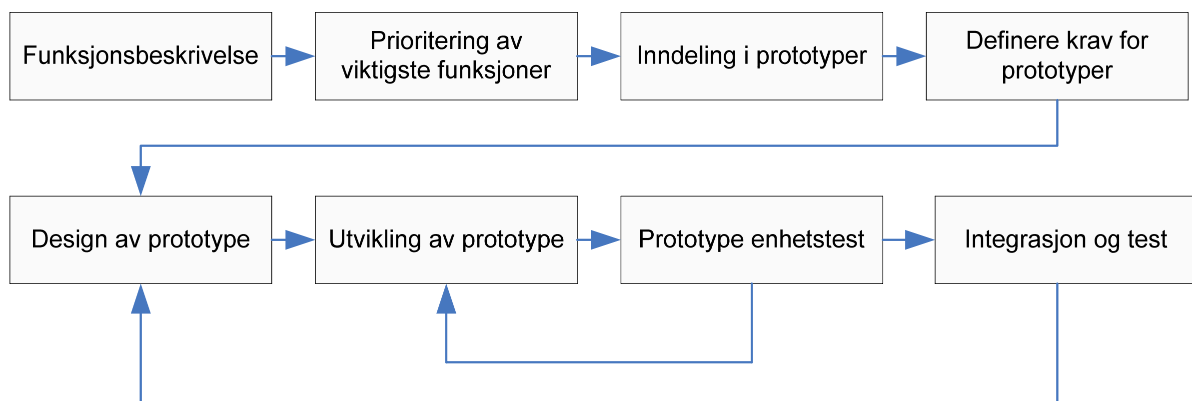
Prosjektgruppas bakgrunn er hovedsaklig fra datatekniske fag. Samtlige har imidlertid noe kunnskap innen elektronikk/automasjon fra videregående skole. Alle har kompetanse innen programmering og systemutvikling.
Behovet for fordypning og studie gjennom prosjektet har vært på automasjonsteknisk ustyr og det spesifikke utstyret som benyttes i autoklaven. I tillegg var det behov for kompetanseheving på valgt utviklingsplattform – Microsoft .NET 2.0 – for å kunne dra nytte av de spesielle teknikker og metoder som var tilgjengelige der.

Prosjektet har lagt stor vekt på å benytte tilegnet lærdom fra programutviklings studieretningen for å gjennomføre utviklingsprosessen på en best egnet måte. Dette har gitt oss en robust og skalerbar løsning som vil innfri kravet om stabilitet.

1.4 Arbeidsmetoder

Førende for prosjektet var at også autoklaven var under utvikling. Selv om det forelå klare krav for maskinens funksjon, så vi et behov for en fleksibel utviklingsmodell der vi kunne være åpne for endringer underveis samt ha muligheter for iterativ jobbing. Det var i tillegg behov for tidlige prototyper for å kunne starte testing og gjøre tilpassning av maskinvaren i autoklaven.

I innledende møter med oppdragsgiver satte vi opp en prioritering av hva som var viktig å få ferdig tidlig. Det er to forskjellige prosessforløp som skal kunne kjøres i autoklaven, og disse ble kategorisert som meget viktige for systemet. Det ble bestemt at første prototyp skulle være vakuumtest – test for å verifisere at autoklavens kammer er lufttett - og andre prototyp skulle være B&D Test – steriliseringsprosess. I tillegg var det viktig å kunne skape en visualisering av prosessen. Det ble derfor besluttet at en prototyp av GUI var nødvendig parallelt med de andre prototypene.



Figur 1-1: Utviklingsprosessen

Vi startet med å utarbeide en så fullstendig kravspesifikasjon som mulig for hver enkelt prototyp. Deretter gjennomførte vi en designfase der vi la ned mye arbeid i å

finne et så optimalt design som mulig. Dette ga oss et gjennomtenkt og fleksibelt rammeverk som vi kunne bygge videre funksjonalitet på.

Videre jobbet vi med utvikling av prototypene og de delene av systemet vi trengte for å lage disse. Det vi trengte i først omgang var kommunikasjonen mot busskobleren. Underveis i utviklingen av prototypene la vi til funksjonalitet når dette passet slik som dynamiske verdier, styring av komponentene rundt prosessen og administrative skjermbilder.

Mot slutten av prosjektperioden testet vi produktet på en ekte autoklav der vi tilpasset løsningen til diverse problemer og misforståelser rundt funksjon som dukket opp. Autoklaven som det ble testet på var ikke den nyutviklede autoklaven til AH Partner, men en eldre modell der vi kun byttet ut styresystemet. I utgangspunktet skulle testen kjøres på den nyutviklede autoklaven, men denne ble ikke ferdig i prosjektperioden. Vi fikk allikevel testet all funksjonalitet bortsett fra varmeveksleren siden denne ikke finnes på eldre modeller.

1.5 Organisering av rapporten

Rapporten er delt inn i 7 kapitler. Siden rapporten inneholder konfidensielt materiale om autoklavens funksjonalitet og oppbygning har det vært nødvendig å flytte viktig innhold ut i vedlegg. Det anbefales å lese disse vedleggene når det refereres til dem. Disse vedleggene vil imidlertid ikke være med i utgaven av rapporten som blir offentlig tilgjengelig via biblioteket ved Høgskolen i Gjøvik.

1. **Innledning og forord**
   Dette kapittelet omhandler prosjektet og praktiske opplysninger rundt dette.
2. **Fagteori og utstyr**
   Dette kapittelet forklarer om autoklavens oppbygning. Mye av denne informasjonen er tatt ut som vedlegg siden dette er av konfidensiell art.
3. **Kravspesifikasjon**
   Dette kapittelet Inneholder kravspesifikasjonen for applikasjonen.
4. **Design**
   Dette kapittelet omhandler designet av løsningen, det er av naturlig grunner også en del implementasjon beskrevet i dette kapitelet.
5. **Implementasjon**
   I dette kapittelet har vi tatt for oss de viktigste delene av løsningen og beskrevet implementasjonen av disse.
6. **Test**
   Dette kapittelet beskriver hva som er testet i løsningen, hvilke type tester som er utført samt hvilke tester som gjenstår.
7. **Avslutning**
   Dette kapittelet inneholder drøfting av oppgaven og løsningen. Hva som gjenstår og noen tanker rundt videre arbeid samt konklusjon.

Vedlegg A
Inneholder konfidensiell informasjon om autoklavens oppbygning.
Vedlegg B
Inneholder konfidensiell informasjon om autoklavens funksjon.
Vedlegg C
Inneholder konfidensiell informasjon om autoklavens hardware grensesnitt.
Vedlegg D
Vedlegget beskriver detaljerte funksjonelle krav i form av UseCase. Disse viser hvordan autoklaven og prosessene skal kontrolleres og er derfor å betrakte som konfidensielt.
Vedlegg E
Timelogg fra gruppas arbeid igjennom vinterhalvåret.
Vedlegg F
Eksempel på møtereferat.
Vedlegg G
Vedlegget viser et utvalg av skjermbilder fra det ferdige produktet.
Vedlegg H
Vedlegg H inneholder forprosjektrapport uten vedlegg.
Vedlegg I
Vedlegg I viser opprinnelig prosjektplan i form av gantt-skjema.
Vedlegg J
Vedlegg J viser faktisk prosjektplan i form av gantt-skjema

2. Fagteori og utstyr

2.1 Steriliseringsprosessen

Målet med en steriliseringsprosess er å drepe alle bakterier, virus, mikrober og sporer som kan befinne seg på det utstyret som skal steriliseres. Dette er - bortsett fra virus - levende organismer som ikke tåler høy varme. Dette kan vi finne eksempler på i husholdningen når vi for eksempel baker. I gjærbakst bruker vi gjær - som er en levende encellet soppkultur - til å gjøre deigen luftig og fin. Hvis gjæren blandes i for varmt vann dør de levende mikroorganismene og hevingen misslykkes. Denne soppkulturen dør ved ca 50 °C, som egentlig er en relativt lav temperatur. De fleste bakterier som vi har rundt oss i hverdagen dør når temperaturen kommer opp mot 100 °C, og vi kan dermed kvitte oss med en god del bakterier ved vanlig koking.

Koking er imidlertid ikke nok når noe skal karakteriseres som sterilt. Det vil si at det finnes mange skumle bakterier som tåler mye høyere temperaturer. Ved å opprettholde en temperatur 134 °C i noen minutter vil en drepe alle farlige bakterier som finnes rundt oss.

Grunnen til at denne temperaturen må opprettholdes over en periode er at bakteriene ikke dør så fort denne temperaturen oppnås. De har en viss motstandskraft, men ved 134 °C blir det så tøffe forhold at de ikke klarer å overleve over en lengre tidsperiode.

Det stilles strenge krav til en steriliseringsmaskin som skal brukes i medisinsk sammenheng for å garantere at det som er kjørt igjennom autoklaven er sterilt.

2.2 Autoklavens oppbygning

Denne seksjonen er flyttet til vedlegg A på grunn av teknisk informasjon som ikke skal være offentlig tilgjengelig.

2.3 Autoklavens funksjon

Denne seksjonen er flyttet til vedlegg B på grunn av teknisk informasjon som ikke skal være offentlig tilgjengelig.

2.4 Hardware grensesnitt

Denne seksjonen er flyttet til vedlegg C på grunn av teknisk informasjon som ikke skal være offentlig tilgjengelig.

3. Kravspesifikasjon

3.1 Introduksjon

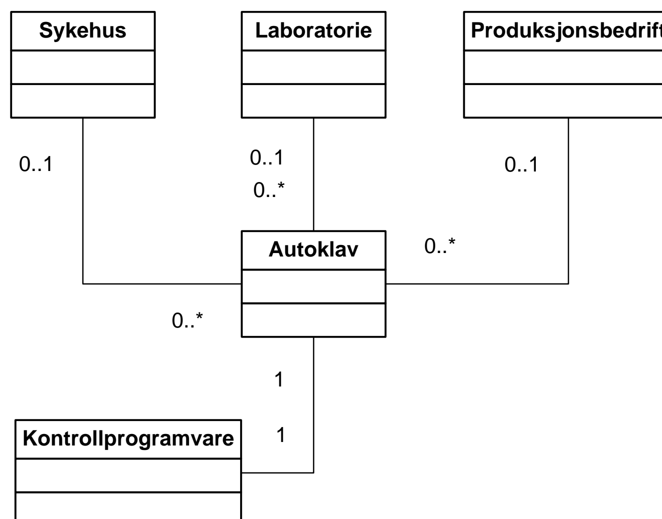
Denne seksjonen beskriver krav til systemet og omgivelsene rundt. Kravene har kommet fra en uformell kravspesifikasjon produsert av oppdragsgiver samt muntlig kommunikasjon.

3.1.1 Krav til systemet

Systemet skal kunne ivareta alle funksjoner knyttet til styring og konfigurering av autoklaven. Programvaren skal kunne kommunisere med I/O modul for kommunikasjon med automasjonsteknisk maskinvare i autoklaven som pumper, ventiler, temperatur- og trykksensorer og lignende. Det skal kunne opprettes og konfigureres egne programdefinisjoner for prosessforløpet ved sterilisering i tillegg til forskjellige tester for godkjenning og sertifisering av autoklaven.

3.1.2 Systemets omgivelser

Systemet skal kjøres på hardware fysisk montert på selve autoklaven. Autoklaven vil kunne benyttes i forskjellige miljøer som sykehus, laboratorier, produksjonsbedrifter og lignende.



Figur 3-1: Verden rundt autoklaven

3.1.3 Systemets brukere

Systemet vil i hovedsak ha 3 forskjellige brukergrupper. Det vil være varierende hvilke type brukere som vil betjene maskinen/systemet til daglig, men det er viktig at brukergrensesnittet er så enkelt og intuitivt som mulig for alle gruppene.

Operatører vil typisk være daglige brukere av systemet. Det settes ingen forutsetninger om bakgrunn eller datakyndighet hos denne gruppen. Personell som sykepleiere og laboratoriepersonell vil være typiske kandidater for operatørgruppen.

Administrator vil være ansvarlig for konfigurering av systemet og maskinen. Det stilles krav til at denne gruppen har kjennskap til autoklaverspesifikk funksjonalitet og kunnskap om prosessforløpet ved sterilisering. Denne gruppen vil også være ansvarlig for å vedlikeholde definisjoner for steriliseringsprogrammer og tester. Det vil være knyttet noe større krav til datakunnskap for denne gruppen da brukergrensesnitt for konfigurering vil være mer avansert.

Installatør/servicepersonell vil være gruppen som installerer maskinvaren og systemet generelt. Denne gruppen må ha kunnskap om datatekniske aspekter rundt operativsystem og generell maskinkonfigurasjon. Det vil stilles krav om datakunnskap for installasjon av systemet og editering av konfigurasjonsfiler i forbindelse med kommunikasjon med I/O-modul.

Felles krav til opplæring vil være i form av kurs og/eller systemets dokumentasjon.

3.2 Overordnede funksjonelle krav

Denne seksjonen gir en overordnet beskrivelse av de funksjonelle kravene i form av UseCase. Prioriterte deler er beskrevet i detalj under detaljerte funksjonelle krav. Skjermbildene som viser hvordan dette skal gjøres er tatt fra den ferdige løsningen, dette er gjort siden det gir en god illustrasjon i rapportsammenheng.

3.2.1 Funksjon

Figur 3-2: UseCase diagram

De funksjonelle kravene representert i Figur 3-2. Både Administrator og Servicepersonell kan ta på seg rollen som Operatør og dermed utføre hans UseCase.

3.2.2 Vise hjelp

Aktør: Operatør/Administrator/Installatør/Servicepersonell
Aktør ønsker hjelp om bruk av systemet. Hjelp skal være tilgjengelig fra knapp i hovedmenyen der øvre del av skjermbildet erstattes med en kortfattet dokumentasjon for systemet.

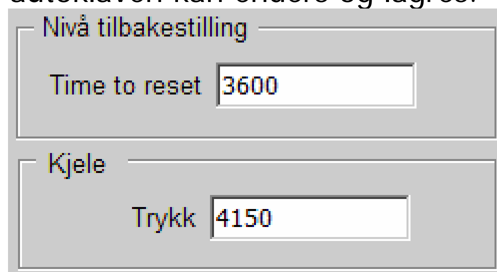
3.2.3 Vedlikeholde driftstider

Aktør: Administrator/Installatør/Servicepersonell
Aktør ønsker å administrere driftstider for autoklaven. Driftstider skal være tilgjengelig i eget skjermbilde der parametere for tidspunkter for drift og verdier for hvilemodus for autoklaven angis.

3.2.4 Konfigurere systemet

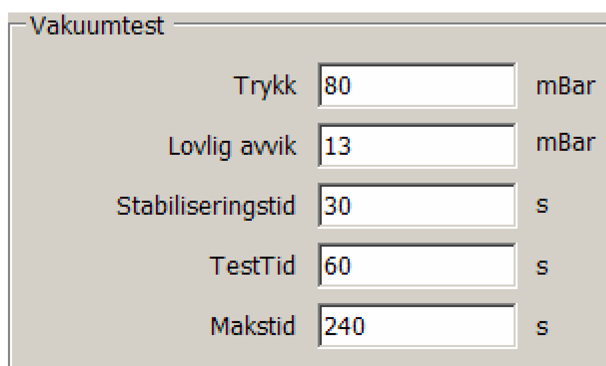
Aktør: Administrator/Installatør/Servicepersonell
Forutsetter at aktør har riktig brukernivå for tilgang til funksjonen.
Aktøren ønsker å konfigurere systemet med et av følgende oppsett:

- **Konfigurering av maskinen**
  Konfigurering av maskinen skal skje i egne skjermbilder der parametere for autoklaven kan enders og lagres.



Figur 3-3: Utdrag fra maskinkonfigurering

- **Oppsett av programmer/prosesser**
  Definering av programmer skal skje ved at aktør oppretter, endrer eller sletter programmer for kjøring av steriliseringsprosesser eller tester.



Figur 3-4: Oppsett av en vakuumtest

- **Konfigurere tilgangsnivåer**
  Konfigurering av tilgangsnivåer i systemet skal skje i eget skjermbilde der aktør kan legge til, endre eller slette brukernivåer og angi hvilken del av systemet den enkelte brukernivået skal ha tilgang til.

- **Språkoppsett**
  Konfigurering av systemets språkvariabler skal skje i eget skjermbilde. Aktør kan endre synlig tekst for GUI-komponenter og systemmeldinger for alle definerte språk i løsningen.

Tekster i systemelement

Beskrivelse	Synlig tekst
Text for displaying current user-level in all views	Brukernivå:
Text displayed on all the systems 'OK' buttons	Ok
Text displayed on all the systems 'Cancel' buttons	Avbryt

Figur 3-5: Oppsett av språk

3.2.5 Utføre service.

Aktør: Installatør/Servicepersonell
Forutsetter at aktør har riktig brukernivå for tilgang til funksjonen.
Aktør ønsker å uføre service på autoklaven. Følgende funksjoner inngår i servicerutine.

- **Konfigurere tellere**
  Konfigurasjon av tellere skal skje i eget skjermbilde. Aktør leser av driftstider og angir intervaller for luftfilter, kjele, vakuumpumpe og autoklaven totalt. Aktør kan nullstille tellere for de enkelte maskinkomponentene.

Intervall vakuumpumpe	4001	s	4402971	s	Tilbakestill
Antall prosesser totalt			143		Tilbakestill

Figur 3-6: Oppsett av timetellere

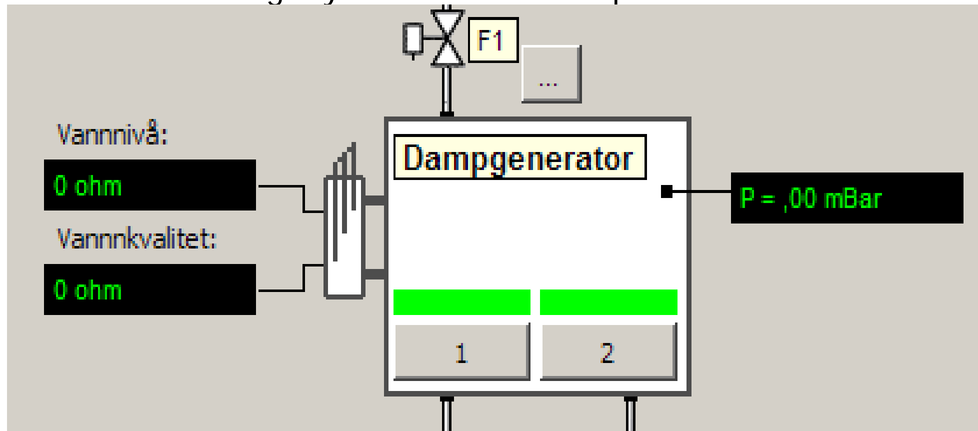
- **Kalibrere sensorer**
  Kalibrering av sensorer skal skje ved at aktør åpner skjermbildet for kalibrering og får en oversikt over alle kalibrerbare sensorer i autoklaven. Aktør leser av reelle minimum og maksimumverdier for aktuell sensor og taster inn målte verdier.

Kammer trykksensor 1

	Ekte	Målt		
Maks		0	Lås	Kalibrer
Min		0	Lås	

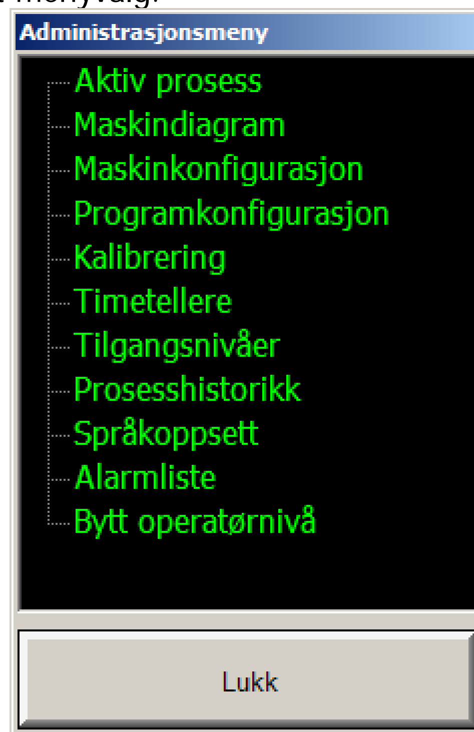
Figur 3-7: Kalibrering av sensorer

- **Teste komponenter og sjekke autoklavens funksjoner**
  Testing og sjekk av komponenter og funksjoner skal skje i eget skjermbilde der et diagram over autoklavens komponenter viser øyeblikksaktiviteter og der man kan teste/tvangsstyre den enkelte komponent.



Figur 3-8: Uttrekk fra maskindiagram

- **Åpne administrasjonsmeny**
  Aktør ønsker en administrasjonsmeny under utførelse av service på autoklaven. Menyen skal være 'flytende' over de andre skjermbildene. Menyen er tilgjengelig via eget menyvalg.



Figur 3-9: Administrasjonsmeny

3.2.6 Endre brukernivå

Aktør: Operatør/Administrator/Installatør/Servicepersonell
Aktør ønsker å endre gjeldende brukernivå i systemet. Endring av brukernivå skal skje via eget menyvalg og aktør blir presentert med dialog for inntasting av ønsket nivå samt gyldig passord.

3.2.7 Administrere alarmer

Aktør: Administrator/Installatør/Servicepersonell
Aktør ønsker å lese/kontrollere alarmer gitt fra systemet. Alarmliste skal være tilgjengelig i eget skjermbilde der alarmer som er kommet siden forrige start er representert som 'Nye alarmer' og historiske alarmer er representert som 'Alle alarmer'. Aktør kontrollerer alarmer og tømmer alarmlistene om ønskelig.

3.2.8 Se på prosesshistorikk

Aktør: Operatør/Administrator/Installatør/Servicepersonell
Aktør ønsker å lese eller skrive ut informasjon/data om prosesser som er kjørt på autoklaven. Prosesshistorikk skal være tilgjengelig i eget skjermbilde der en liste over historiske prosesser er representert. Ved å velge en prosess fra listen skal informasjon om denne prosessen vises med faseendringer og grafisk fremstilling av prosessforløpet.

3.2.9 Hente prosessdata

Aktør: Eksternt system
Aktør ønsker å hente data om prosesser og status på autoklaven. Aktør velger/angir type data og identifikasjon på prosess. Systemet skal returnere forespurt informasjon.

3.2.10 Håndtere alarm

Aktør: Autoklaven
Aktør ønsker å håndtere alarmene i systemet. Om det kommer en kritisk alarm under kjøring av en prosess skal prosessen avbrytes. Alle alarmer skal logges. Om alarmen er viktig skal den presenteres for brukeren.

3.2.11 Starte/Stoppe Autoklaven

Aktør: Administrator/Installatør/Servicepersonell
Forutsetter at aktør har riktig brukernivå for tilgang til funksjonen.
Aktør ønsker å starte eller stoppe Autoklaven. Start stopp av maskinen skal skje ved eget valg i hovedmenyen. Når man stopper autoklaven kjøres systemet ned og maskinen skrus av. Når man starter autoklaven startes maskinen på nytt og programmet for styring av autoklaven starter opp automatisk og gjør seg klar for kjøring av prosesser.

3.2.12 Åpne/Lukke dør

Aktør: Operatør, Installatør/Servicemann, Administrator

Aktør ønsker å åpne eller lukke dør på autoklaven. Åpning av dør skal skje ved eget valg i hovedmenyen eller i prosessbildet. Før dør åpnes vil autoklaven sjekke om alt er klart for at døren kan åpnes. Døren skal ikke kunne åpnes om en prosess kjører. Det skal ikke være mulig å åpne steril dør om forrige prosess ikke er kjørt med godkjent resultat, heller ikke usteril dør om steril dør ikke har vært oppe siden forrige godkjente prosess. På autoklaver med to dører skal det ikke være mulig å åpne en dør om den andre døren er åpen.

3.2.13 Kontrollere dampgenerator

Aktør: Autoklaven

Aktør ønsker å kontrollere dampgeneratorens vannivå og trykk.

- **Vannivå**
  Aktør ønsker å opprettholde vannivå i dampgeneratoren. Vannivået skal overvåkes og det skal fylles vann om nivået er for lavt. Vannfyllingen skal fortsette inntil det er nok vann i dampgeneratoren. Om vannivået er alt for lavt skal systemet generere en kritisk alarm og varmeelementene i dampgeneratoren skal ikke kunne skrus på vannivå er tilfredsstillende.

- **Trykk**
  Aktør ønsker å kontrollere trykket i dampgeneratoren. Om trykket er for lavt skal aktør skru på varmeelementene. Trykket skal styres inn mot settpunkt ved å variere hvilke elementer som er på til enhver tid. Ved feil i systemet skal varmeelementene skrus av.

3.2.14 Kontrollere kappen

Aktør: Autoklaven

Aktør ønsker å kontrollere temperatur på kappen. Om temperaturen er for lav skal kappen skrus på, om temperaturen er for høy skal kappen slåes av. Om det oppstår en feil skal kappen slåes av.

3.2.15 Kontrollere varmeveksler

Aktør: Autoklaven

Aktør ønsker å kontrollere varmeveksler på en slik måte at varmeenergi blir gjenbrukt i høyest mulig grad. Om temperaturen i varmeveksleren blir høyere en et gitt settpunkt skal temperaturen senkes slik at avløpvannets temperatur holdes innenfor lovlige verdier.

3.2.16 Kontrollere vanntank

Aktør: Autoklaven
Aktør ønsker å opprettholde vannivå og kontrollere temperatur i vanntank. Om vannivå er for lavt skal det fylles vann inntil vannet er på korrekt nivå. Om temperaturen er for høy skal det fylles vann inntil temperaturen er sunket til riktig nivå. Om temperaturen er alt for høy skal det genereres en kritisk alarm da dette tyder på at vanntilførsel til autoklaven ikke er i orden.

3.3 Detaljerte funksjonelle krav

Disse kravene er flyttet til vedlegg D på grunn av at de inneholder informasjon som ikke skal være offentlig tilgjengelig.

3.4 Supplementær spesifikasjon

Denne seksjonen beskriver alle krav til systemet som de funksjonelle kravene ikke dekker.

3.4.1 Funksjonalitet

3.4.1.1 Brukergrensesnitt

Systemet skal være enkelt å bruke. Det skal legges vekt på å designe GUI for å være intuitivt med riktig type informasjon.

Prosesser skal representeres grafisk og det vil være nødvendig å lage en egen komponent for denne type visning.

For bedre visualisering av menyer og navigasjonsknapper må det lages egne knapper for grensesnittet. Disse må visualiseres med en dypere 3D effekt enn standardskomponentene i utviklingsverktøyet.

Systemet skal kunne operere uten eksternt tastatur. Det skal lages et 'on-screen' tastatur som er enkelt og dekker de behov systemet har for inntasting av data. Dette skal være godt nok for all input i systemet, fra programoppsett til signering.

Systemet skal ha autentiseringsmekanisme – basert på brukernivå og passord – for å begrense tilgangen til gui-komponenter i henhold til hvilket brukernivå man befinner seg i. Når man går inn i et høyere tilgangsnivå enn standard (brukernivå 1), skal systemet automatisk tilbakestille brukernivået til standard nivå etter et konfigurert antall minutter.

3.4.1.2 Internasjonalisering

Systemet skal ha støtte for flere språk. Det vil i så måte være behov for en fleksibel løsning som tar høyde for endring i lengde på tekst. Språkdefinisjoner skal være på et enkelt tekstformat som gjør at man enkelt skal kunne sende filene til eksterne aktører for oversettelse.

3.4.1.3 Logging

Systemet skal kunne logge alle alarmer og alt som skjer under prosessens gang. Det skal lagres hvor mange prosesser som er kjørt siden hver komponents service samt autoklavens totale arbeidstimer.

Det skal lagres hvor mange timer motorene har gått siden forrige service samt autoklaven totalt.

Alle prosesser skal logges på et detaljnivå som gjør at regenerering av prosessens forløp er mulig i ettertid. Hver prosess skal tildeles en identifikasjon – manuelt eller autogenerert – og det skal være mulig å registrere en eller flere elementer (les utstyr) pr. prosess.

3.4.1.4 Feilhåndtering

Alle feil som oppstår i prosessforløpet skal logges som alarmer. Alarmene skal kategoriseres etter hvor kritisk den enkelte alarm er. Dersom en kritisk alarm oppstår skal systemet kunne håndtere denne på en slik måte at autoklaven tilbakestilles til en tilstand der den er trygg å håndtere. Dette innebærer hovedsakelig at kammeret trykkutjevnes.

3.4.2 Brukskrav

3.4.2.1 Kompatibilitet

Systemet skal være uavhengig av operativsystem, så lenge det støtter hele .NET standarden samt muligheten for Raw Input fra pekeenheter. Dette er for tiden Windows XP og Windows XP Embedded.

Systemet skal – med små forandringer – kunne brukes på AH Partners eksisterende autoklaver samt på den nyutviklede autoklaven.

3.4.2.2 Ytelse

Systemet skal respondere på sanntidsdata og er dermed avhengig av rask prosessering. Systemet skal også kunne kontrollere alle enhetene i autoklaven uten at ventetid resulterer i at kontrollen over enheten blir mistet.

3.4.2.3 Backup

Systemet skal ikke ha en egen backupløsning. Det skal kunne lese filer som er gjenopprettet via et eksternt backupsystem.

3.4.2.4 Andre krav til systemer

Systemet skal utvikles i .NET for enkel integrering med andre systemer. Det stilles ingen krav til oppbygning av softwaren med hensyn til andre løsninger bortsett fra dette.

3.4.3 Design begrensninger

3.4.3.1 Kommunikasjon med autoklaven

Systemet skal lages for å kunne benytte alternative løsninger for kommunikasjon med maskinvare i autoklaven.

3.4.3.2 Lagring av prosess -og alarmdata

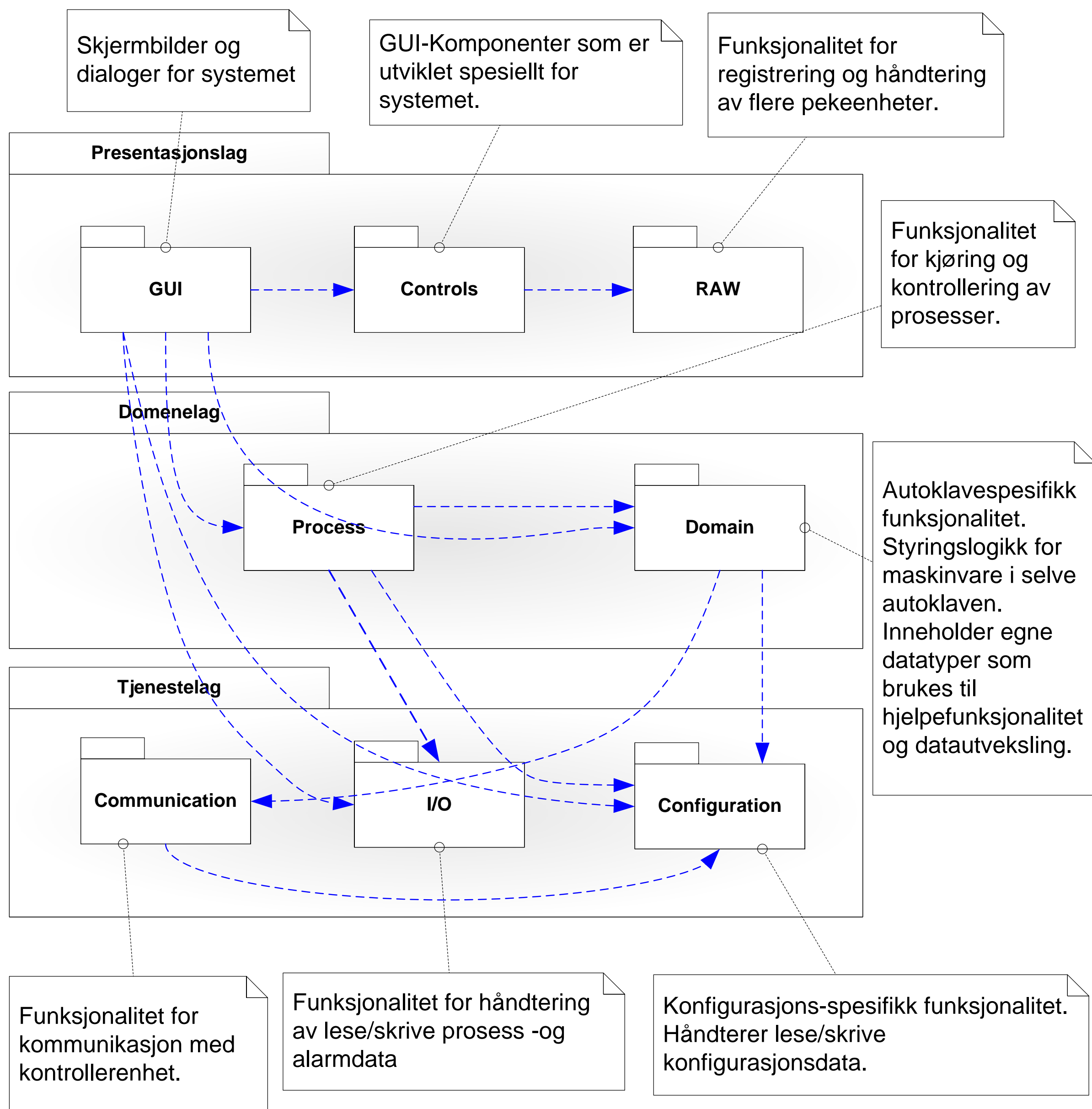
Systemet skal lages for å kunne benytte alternative teknologi for lagring av prosess – og alarmdata.

4. Design

4.1 Introduksjon

"Understanding responsibilities is key to good object-oriented design" - *Martin Fowler sitert i Craig Larman 2005 - Applying UML and Patterns.*

GRASP[1] er benyttet som generelle prinsipper og retningslinjer for utarbeidelse av systemets design. Avsnitt 4.3 gir en detaljert vurdering og beskrivelse av faktorene som i hovedsak har vært førende for systemets arkitektur og design.



Figur 4-1: Systemets hovedinndeling

[1] *Omtalt av Craig Larman 2005 – Applying UML and Patterns*

4.1.1 Overordnet Arkitektur

Arkitekturen i et system bør preges av oppgavene systemet skal utføre, og fokusere rundt en ideell segmentering som imøtekommer både funksjonelle og ikke-funksjonelle krav. I tillegg bidrar en riktig arkitekturmodell til en forenkling av det videre utviklingsarbeidet.

I hovedsak har faktorer fra avsnitt 3.4 Supplementær spesifikasjon vært førende ved vurdering og valg av arkitekturmodell og det er lagt vekt på systemets behov for sanntidsdata, krav til responstid og delegering av input samt systemets evne til endringer i enkeltmoduler.

En distribuert modell er ikke aktuell da systemet skal kjøres i et isolert embedded miljø. Repository modellen er heller ikke aktuell grunnet systemets behov for kontinuerlig sanntidsdata. Valget falt derfor naturlig på en lagdelingsmodell. Fordelene med denne inndelingen er at den støtter en inkrementell utvikling. Hvert lag representerer et gruppering av funksjonalitet og det etableres en kommunikasjonsstandard mellom dem. På den måten kan utvikling innad i de forskjellige lagene foregå uten at det påvirker de andre. I tillegg gjør denne inndelingen det enkelt å endre funksjonaliteten på et senere tidspunkt. Hvis man vil benytte en alternativ teknologi for lagring av data, vil denne tjenesten kunne byttes uten at de øvrige modulene/lagene påvirkes. Ulempene med en slik inndeling er at den kan bli kompleks i struktur, og det kan være vanskelig å opprettholde prinsippene for hvordan kall gjøres på tvers av lagene. Eksempler på dette er at det ofte vil være behov for fil I/O i alle lag. Hvis tjenester for I/O ligger i det nederste laget vil det for øverste laget medføre kall til underliggende lag som igjen må videreføre dette videre nedover i modellen.

4.2 Logisk inndeling

Dette avsnittet beskriver logisk inndeling og oppbygningen av programvaren. En mer detaljert beskrivelse er gitt i kapittel 5 Implementasjon. Det er imidlertid beskrivelser i dette avsnittet som grenser mot implementasjon i detaljeringsgrad. Dette er bevisst, og er gjort på grunn av at det er var vanskelig å skille mellom design og implementasjon for funksjonaliteten beskrevet og det ble vurdert som uhensiktsmessig å dele det opp.

4.2.1 Lag –og modulinndeling

Med utgangspunktet i trelagsarkitekturen beskrevet i avsnitt 4.1.1 er systemet delt opp i en struktur som gjenspeiler oppgaver de forskjellige delene av løsningen skal utføre.
Presentasjonslaget håndterer all interaksjon med brukeren av systemet som navigering, autentisering, konfigurering, oppsett av prosesser, starting av prosesser og overvåking av prosessen og selve autoklaven. Domenelaget har logikk for håndtering av autoklavespesifikk funksjonalitet som ventiler og pumper samt styring av prosesser. Tjenestelaget står for kommunikasjon med busskobler og lesing/skriving til disk/fil. Et viktig prinsipp i denne laginndelingen er at kall skal gå nedover i modellen. For kommunikasjon mellom lagene er det definert egne

grensesnitt/standarder som isolerer hvert enkelt lag og som gjør at endring i intern funksjonalitet ikke påvirker de andre lagene i modellen.

4.2.1.1 Presentasjonslaget

Presentasjonslaget er delt opp i tre pakker, en for skjermbilder, dialoger og generell GUI-funksjonalitet, en for spesielle komponenter som blir brukt i skjermbildene og en som tar seg av koordinering av hvilken pekeenhet som blir brukt. Pakkeinndelingen er resultat av en naturlig gruppering ut i fra funksjonalitet og oppbygning. I tillegg gir inndelingen fleksibilitet for individuell utvikling og test av pakkene.

4.2.1.2 Domenelaget

Domenelaget er delt i to pakker. I selve domenepakken er klassene som representerer de forskjellige enhetene i den fysiske autoklaven. I tillegg inneholder denne pakken objekter som benyttes i mange deler av systemet. Dette er objekter for datautveksling og tidtaking. Koden og funksjonalitet for prosessene er skilt ut i en egen pakke. Pakkeinndelingen er gjort på bakgrunn av knytninger og felles funksjonalitet i interne klasser. Inndelingen gir dermed en logisk gruppering av ansvarsområdene i domenelaget.

4.2.1.3 Tjenestelaget

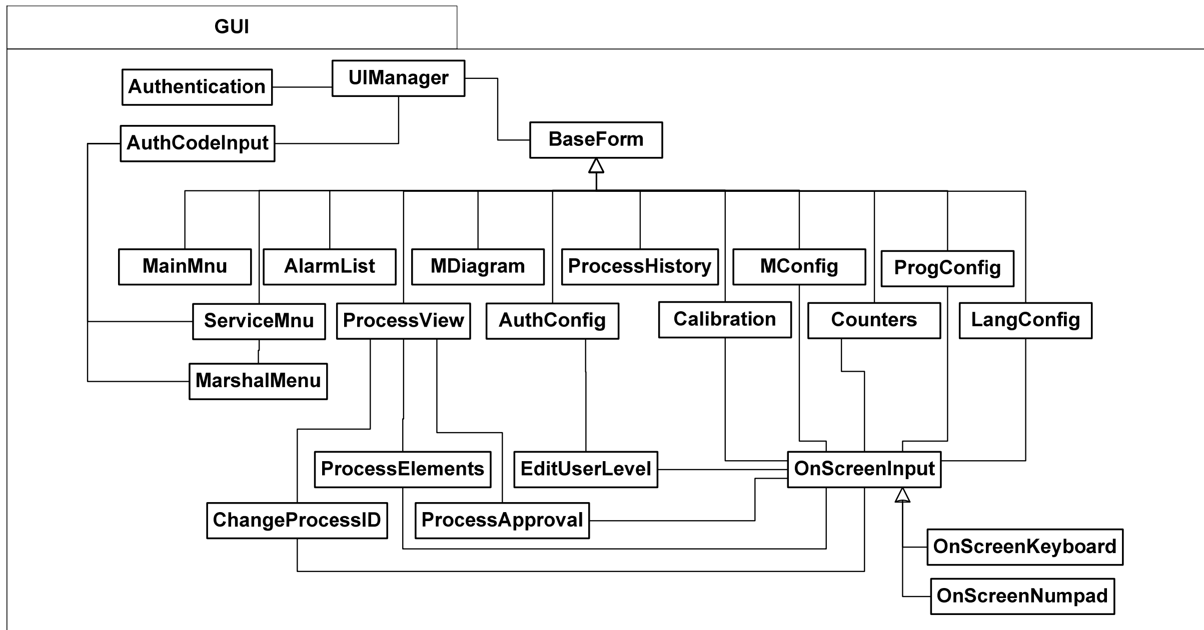
Tjenestelaget er delt opp i tre pakker. En pakke tar seg av kommunikasjonen mot busskobleren, en tar seg av lesing og skriving av konfigurasjonsdata til fil samt en som tar seg av lagring av prosessdata. Det var logisk å ta ut kommunikasjonen mot busskobleren som en egen pakke. På denne måten kan denne enkelt byttes med en annen busskobler. Skillet mellom *IO* og Configuration var ikke så enkelt, men siden lagring av prosessdata enkelt skal kunne bytte lagringsmedia valgte vi å ha dette som en egen pakke.

4.2.2 Pakkene

Dette avsnittet gir en mer detaljert beskrivelse av de enkelte pakkene og hvordan klassene internt er relatert.

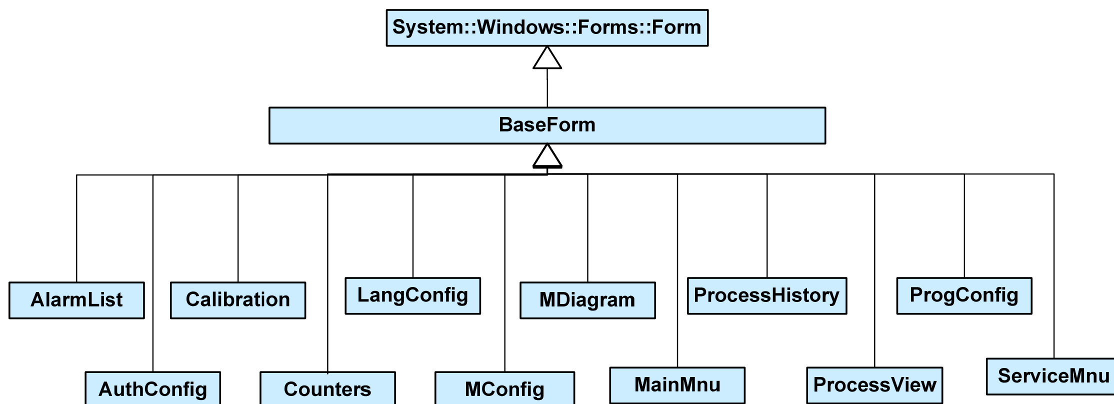
4.2.2.1 GUI

GUI-pakken består av alle skjermbilder og visuelle elementer i løsningen. Systemet skal kjøres på trykkfølsomme skjermer og må fungere uten tastatur og mus. Dette har lagt klare føringer for hvordan skjermbildene er bygd opp. Løsningen er designet for 15″ skjerm med en oppløsning på 1024x768 ppi.



Figur 4-2: AH::Pride::GUI

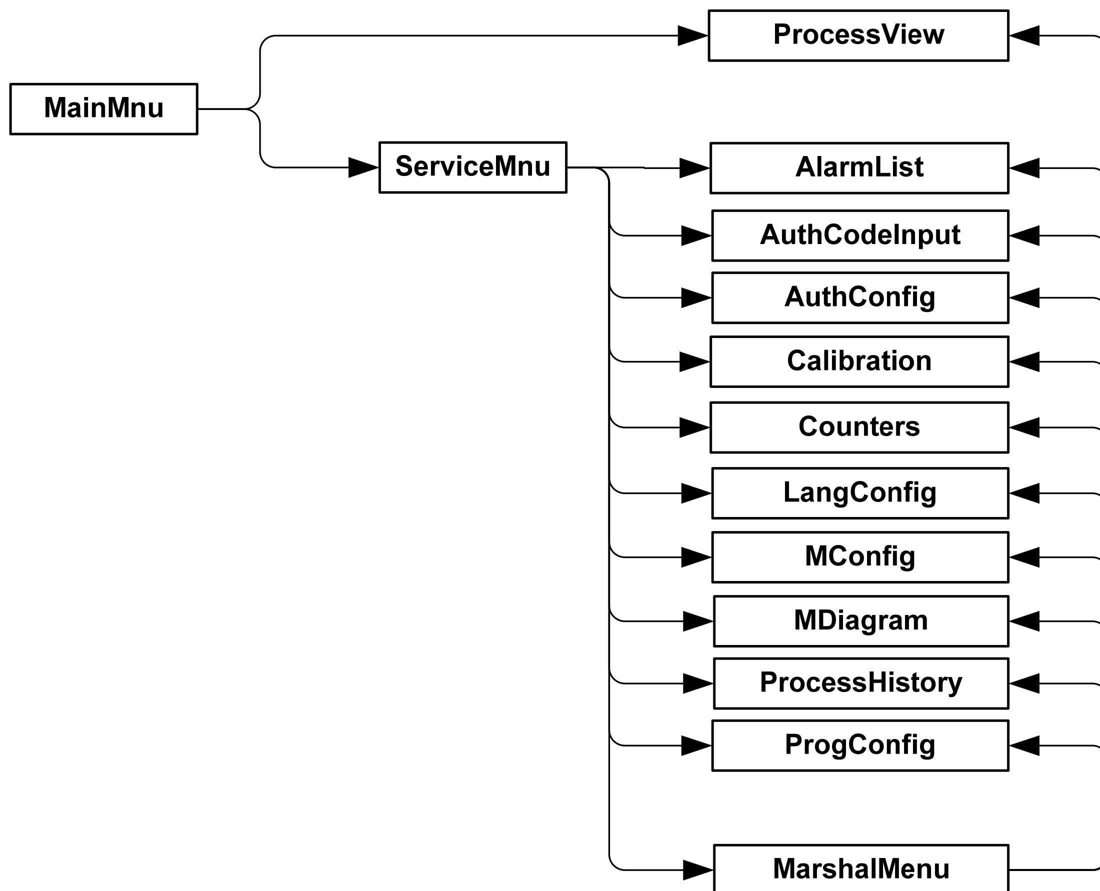
For å lage et felles rammeverk for alle skjermbilder i løsningen er en baseklasse valgt som mal. Denne klassen heter BaseForm og arver fra Forms-klassen i .NET rammeverket. Alle skjemaer i en managed Windows-applikasjon må arve fra denne klassen for å kunne tegne grafikk og håndtere window-callbacks.



Figur 4-3: Arvehierarki for skjermbilder

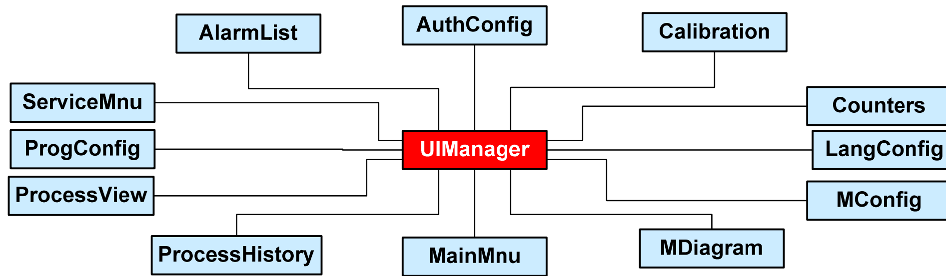
Alle skjermbilder i løsningen med unntak av dialoger arver fra BaseForm. På denne måten implementeres 'felles' funksjonalitet og layout via en felles klasse.

Utgangspunktet for navigering i løsningen er hovedmenyen MainMnu (se Figur 4-4). Navigeringen i systemet skjer fra MainMnu og ut i de forskjellige funksjonene. For vanlige brukere vil navigering foregå som en frem og tilbake navigering mellom hovedmenyen/servicemenyen og skjermbildene. Denne måten å navigere på tvinger brukeren til å returnere til utgangspunktet før videre navigering til nye deler av systemet kan skje.



Figur 4-4: Navigasjonskart

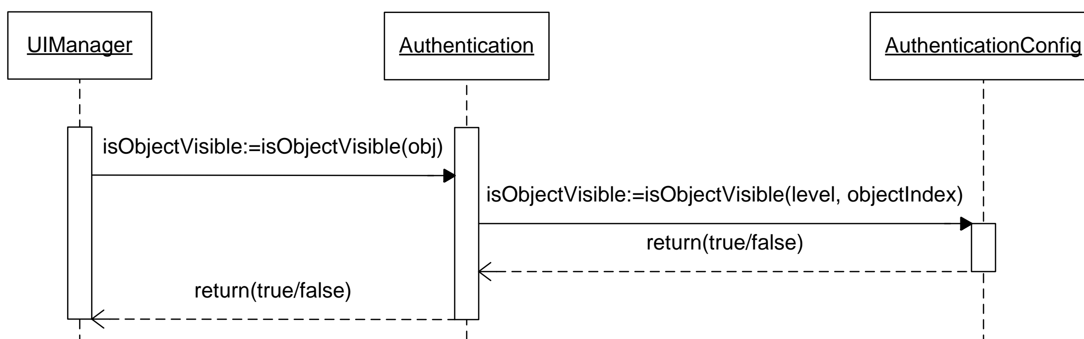
Det er imidlertid behov for en 'flytende' administrasjonsmeny i servicemodus som skal kunne bytte mellom alle skjermbilder uavhengig av hvor man er i systemet (MarshalMenu). Ettersom skjermbildene legger seg over hverandre etter hvert som de blir åpnet vil man med denne menyen opprette et nytt objekt for hver gang en skjermbildekommando blir kalt. Dette er ikke fordelaktig for noen av skjermbildene grunnet ressursbruk og callback-registrering. Et eksempel på dette er prosessbildet der det ville være katastrofalt om to instanser ble laget. Disse ville da jobbet mot samme maskinvare i autoklaven noe som kunne resultere i ukontrollerbare hendelser. Bruk av singleton-pattern er en alternativ løsning, men det ble vurdert som uegnet pga behovet for regenerering av objektene.
For å løse dette er det lages en managerklasse kalt UIManager. Denne klassen har et overordnet ansvar for å opprette og forkaste skjermbildeobjekter.



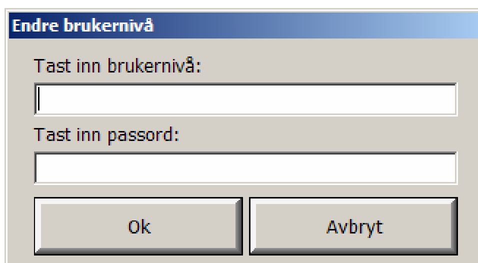
Figur 4-5: UIManager

Hver gang det er behov for å åpne et skjermbilde kalles UIManager som har en intern liste med referanser til åpne skjermbilder. Når et skjermbilde blir forespurt, sjekker UIManager om skjermbildet allerede er åpent. Hvis det er åpent, vil dette skjermbildet bli aktivert, hvis ikke vil det bli opprettet og vist.

Systemet har forskjellige tilgangsnivåer som representerer de forskjellige brukergruppene for systemet (Se avsnitt 3.1.3). Ved å assosiere definerte GUI-elementer med det enkelte brukernivået oppnåes tilgangskontroll på visuelt nivå. Det vil si at det begrenser tilgang til synligheten av informasjonen i løsningen. Kommandoen for GUI-elementet gjør kall til UIManager som autentiserer elementet mot gjeldende brukernivå. Autentisering skjer ved at dette kallet videreformidles til Authentication-objektet som igjen legger til gjeldende brukernivå og gjør et kall til AH::Pride::Configuration som gjør oppslag i konfigurasjonspakken for tilgangsdefinisjoner.



Figur 4-6: Sekvens ved autentisering



Figur 4-7: Dialog for endring av brukernivå

Hvis brukeren ikke er autentisert for aktuelt skjermbilde/GUI-element, blir dialog for endring av brukernivå presentert. Dialogen blir vist helt til gyldig brukernivå og passord blir angitt eller at brukeren avbryter.

4.2.2.2 GUI::Controls

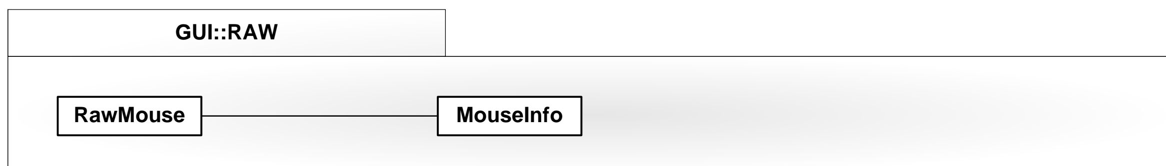
Denne pakken inneholder de spesielle komponentene vi trenger for å bygge opp vårt brukergrensesnitt.

Figur 4-8: GUI::Controls

Den inneholder en knapp som ser mer tredimensjonal ut en de vanlig Windows-knappene, en graf som skal vise prosessens forløp og et diagram som gir en grafisk fremstilling av komponentene i autoklaven. I tillegg er det en komponent som brukes under validering av input i skjermbilder. Den er avhengig av GUI::Raw på grunn av at ahButton bruker denne. Komponentene i denne pakken benyttes av andre elementer i presentasjonslaget, og det er derfor ingen definerte relasjoner innad i denne pakken.

4.2.2.3 GUI::Raw

Denne pakken inneholder funksjonaliteten for å finne ut hvilken pekeenhet som er brukt for å trykke på knapper.



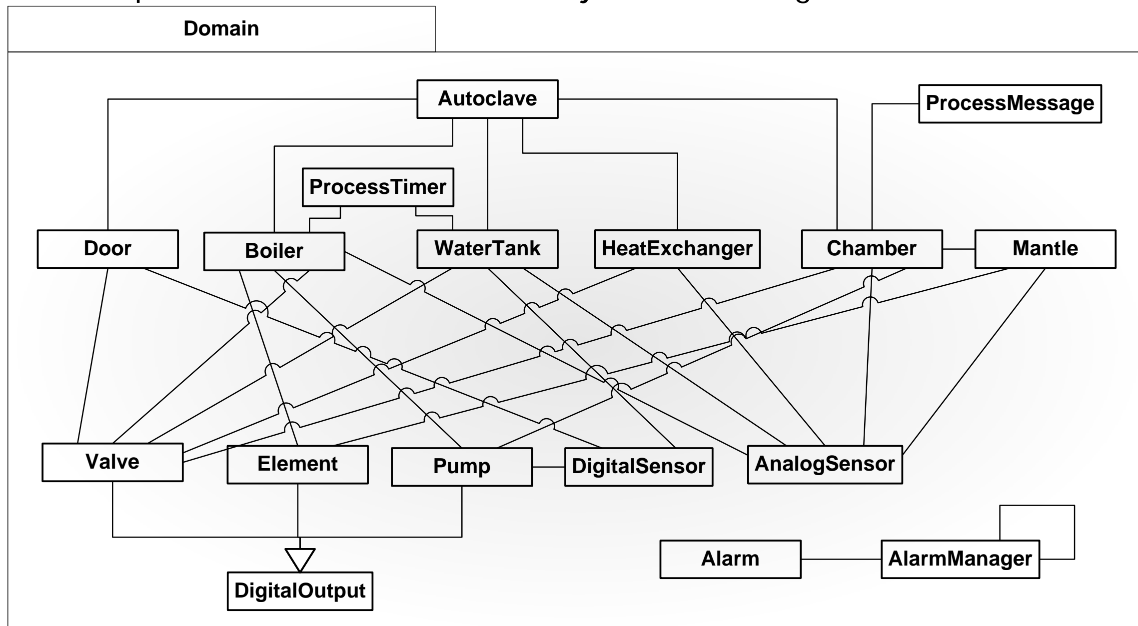
Figur 4-9: GUI::RAW

Dette er den eneste delen av løsningen vår som bruker native[2] kode, dette er på grunn av at .NET ikke har støtte for å finne hvilken pekeenhet som ble brukt. Den er ikke avhengig av noen andre pakker, men den benytter funksjonalitet som kun finnes i Windows XP kjernen og nyere.

[2] *Kode som ikke styres av rammeverket i .NET*

4.2.2.4 Domain

Denne pakken tar seg av alt som omhandler de forskjellige komponentene i autoklaven, den styrer ventiler, pumper og håndterer alarmer. Oppbygning og innhold i pakken er beskrevet i mer detalj i avsnitt 4.4 og 4.5.

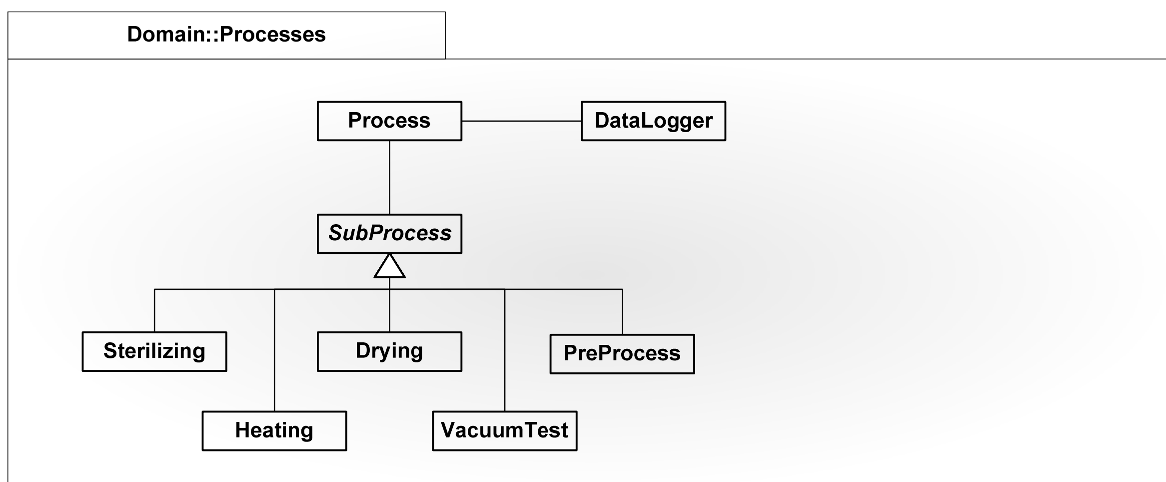


Figur 4-10: Domain

Den inneholder også ProcessMessage-klassen som benyttes for meldingsutveksling mellom klassene og lagene. ProcessTimer-klassen benyttes for tidsutmålinger i forbindelse med kjøring av prosesser. Den er avhengig av Communication, Configuration og IO. Igjen ser vi at det kun er koblinger nedover i laginndelingen.

4.2.2.5 Domain::Processes

Denne pakken kjører og logger prosesser. Oppbygning og innhold i pakken er beskrevet i avsnitt 4.5.

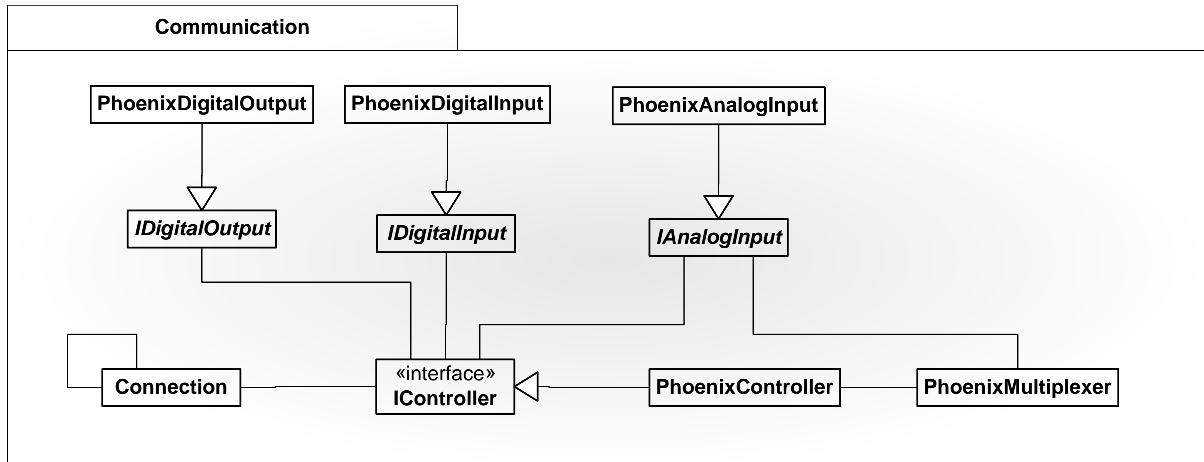


Figur 4-11: Domain::Processes

Pakken er avhengig av Domain, Configuration og IO.

4.2.2.6 Communication

Denne pakken tar seg av kommunikasjonen mot busskobleren. Den inneholder de abstrakte klassene som må implementeres for at busskobleren skal kunne byttes ut.

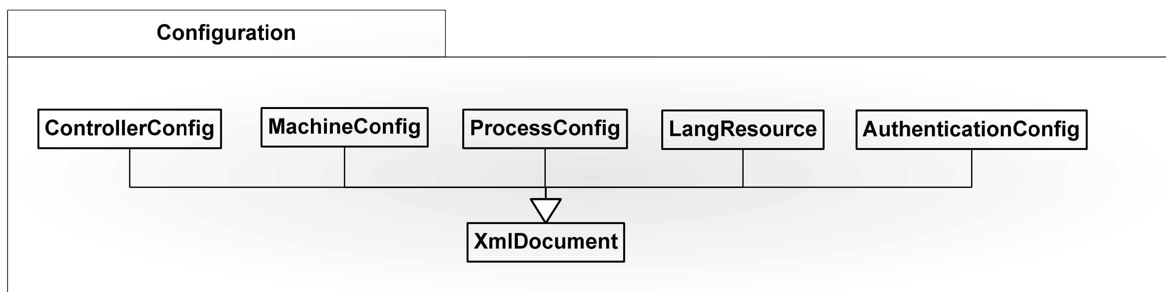


Figur 4-12: Communication

Bakgrunn for inndeling og oppbygning av pakken er beskrevet i avsnitt 4.3.1. Pakken er avhengig av Configuration.

4.2.2.7 Configuration

Denne pakken inneholder funksjonaliteten for å skrive og lese konfigurasjonsdata til og fra fil. Oppbygning og innhold i pakken er beskrevet i avsnitt 4.6.

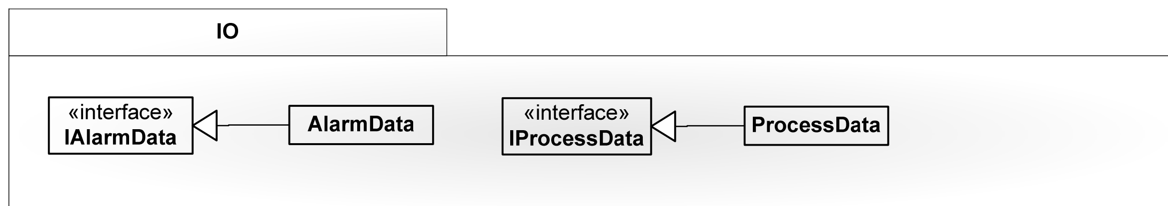


Figur 4-13: Configuration

Den er ikke avhengig av noen av de andre pakkene.

4.2.2.8 IO

Denne pakken tar seg av skriving og lesing av prosessdata og alarmdata til og fra fil. Den inneholder også de abstrakte klassene som må implementeres om man vil bytte ut hvordan disse dataene skal lagers.



Figur 4-14: IO

Den er avhengig av Domene på den måten at den må gjenkjenne data fra denne pakken. Da får vi et problem med sirkulær avhengighet der IO og Domene begge er avhengig av hverandre. Rent teknisk er dette løst ved å ta ut disse datatypene i en egen dll, men de hører fremdeles logisk til i domenepakken og ligger i dette namespacet.

4.3 Tekniske notater

Dette avsnittet inneholder tekniske notater som gir en redegjørelse for designvalg som er gjort i løsningen. De tekniske notatene benyttes som utgangspunkt for videre inndeling av løsningen i klasser og moduler som beskrevet i avsnitt 4.2.

4.3.1 Systemet skal ikke være avhengig av en bestemt busskobler

Faktor
Systemet skal ikke være avhengig av en bestemt busskobler. Systemet skal designes med høyde for at busskobleren fra PhoenixContact kan byttes ut med en annen busskobler.

Vurdering
Problemet med en annen busskobler er at den har andre initialiseringsrutiner, andre måter å sette porter og annen funksjonalitet. Dette kan føre til at vi må forandre kode i andre deler av løsningen, noe som er langt fra idealt. En løsing på dette er å bruke et design pattern kalt adapter.

Løsning
Vi lager et interface med metoder som andre moduler i løsningen bruker. Om vi bytter ut busskobleren må alle disse metodene implementeres i en klasse arvet fra dette interfacet. Den ene ulempen med dette er at under utviklingen må vi lage noen ekstra klasser, men vi ser ikke på dette som et problem.

4.3.2 Callback funksjonalitet

Faktor

Systemet trenger å få vite om forandringer i verdiene på busskobleren.

Vurdering

I driveren fra PhoenixContact ligger det innebygd funksjonalitet for callback. Dette vil si at når det skjer en endring på en inngang, vil det bli sent et signal til driveren. Vi kan implementere denne funksjonaliteten i vårt design slik at en hendelse kan oppstå i vårt program når en endring skjer på en inngang på kontrolleren. Dette er en funksjonalitet som vil gi klar føring for designet av applikasjonen siden det er disse callbackene som vil trigge forandringene i systemet som omhandler forandringer i inngangssignalene.

En annen løsning på dette kunne vært å gjøre en form for polling, der en del av systemet har ansvaret for å se på hver inngang innen for et gitt tidsintervall om noen har forandret seg. Dette vil være en meget tungvint løsning siden det ligger innebygd funksjonalitet for at inngangene selv kan si i fra når de har forandret verdi. I tillegg vil dette føre til at man kan gå glipp av signaler om man ikke poller ofte nok.

Løsning

Vi har valgt å benytte den innebygde funksjonaliteten i driveren. Hvis det senere blir benyttet en busskobler uten callback innebygd kan vi selv lage denne funksjonaliteten i kommunikasjonsmodulen. Dette blir gjort i vår løsning da vi koblet til en komponent på busskobleren som vi måtte multiplekse, det viste seg at dette fungerte utmerket.

4.3.3 Klasser skal ha mulighet for å overvåke andre klasser

Faktor

Med bakgrunn i avsnitt 4.3.2, er det behov for at klasser kan sende melding om endringer i verdier videre til andre klasser. Klasser skal derfor ha mulighet for å overvåke andre klasser.

Vurdering

Denne overvåkningen kan foregå på to måter. Den ene måten går ut på at den som overvåker poller den som blir overvåket med jevne mellomrom. Problemet med dette er at vi ikke får beskjed momentant da den som overvåkes skifter verdi, men vi får en fordel ved at de som overvåkes ikke trenger å vite om hvem som ser på dem. Den andre måten er å la de som overvåkes si i fra da de har en endring, dette fører til at de må ha en liste over hvem som skal ha informasjonen. Dette passer oss veldig godt på grunn av at designet er hendelsesbasert.

Løsning

I prototypen brukte vi alternativet med push, dette ble implementert via Observer / Observable patternet der vi lagde baseklassene selv. Det viste seg senere at dette patternet allerede var implementert i .NET via Events. Derfor skiftet vi til å bruke disse.

4.3.4 Systemet skal ha støtte for flere språk

Faktor

Systemet skal støtte flere språk i GUI og alle meldinger som blir presentert til brukeren. Språkoppsettet skal være enkelt å administrere og skal kunne distribueres til eksterne aktører for oversettelse.

Vurderinger

Microsoft Visual Studio har funksjonalitet for en meget fleksibel måte å internasjonalisere løsningen på. Dette gjøres ved at det genereres egne ressursfiler for hvert språk hvor man definerer språkvariabler. I tillegg kan man med disse ressursfilene variere selve layouten i skjermbildet som for eksempel størrelse og plassering av tekst og komponenter. Ulempene med denne løsningen er at man får en ressursfil for hvert GUI-element (skjermbilde). Dette gir mange filer man må administrere per språk. I tillegg er dette filer som må kompileres for å kunne benyttes av applikasjonen. Dette gir oss en løsning som er tung å vedlikeholde utenfor et utviklingsmiljø.
Alternativet er å lage egen funksjonalitet for å håndtere språkressurser i et enklere og mer vedlikeholdbart format.

Løsning

Vi valgte å implementere internasjonaliseringen som en egen løsning for å kunne innfri de krav som ble stilt til løsningen. Språkvariable blir definert i i XML-filer og det opprettes en slik språkfil per språk. En egen klasse tar seg av utlesing av tekststrengene på en enkel måte.

```
<element section="General" desc="Text displayed on all the systems 'OK'
buttons" key="GUI.App:_ok" text="Ok" />
<element section="General" desc="Text displayed on all the systems
'Cancel' buttons" key="GUI.App:_cancel" text="Avbryt" >
<element section="General" desc="Text displayed on all the systems 'Back'
buttons" key="GUI.App:_back" text="Tilbake" />
```

Ulempen med å benytte XML er at dette blir et relativt uleselig format selv om det er et rent tekstformat. For å løse dette er det laget funksjonalitet for vedlikehold av variablene i selve løsningen.

4.3.5 Systemet skal designes for å kunne benytte alternative teknologier for lagring av data.

Faktor
For lagring av data i systemet skal man kunne benytte forskjellige teknologier for dette. Mange institusjoner har egne løsninger for datalagring og systemet skal kunne tilpasses disse individuelle løsningene.

Vurdering
Som standardformat er XML diskutert. Dette er et format som i større og større grad benyttes som standard innen lagring og utveksling av data. XML er imidlertid et tekstfilformat, og man er i så måte avhengig av å lagre dataene som filer i et filsystem. Databaser er benyttet i større grad for lagring av programsspesifikk informasjon, og er brukt i de fleste organisasjoner. Databaser tilbyr fordeler med tanke på utveksling og rapportering via et standardisert verktøy, og vil i så måte ofte foretrekkes av større organisasjoner.

Løsning
Vi har valgt å designe systemet for å kunne være uavhengig av underliggende lagringsformat. Dette løses ved at modulene for lagring skilles fra selve systemet, og det opprettes standard grensesnitt for lesing og skriving av data. Dette gjør at lagringsmodulen kan tilpasses individuelle lagringsformater uten at selve løsningen trenger å ta hensyn til dette.

4.3.6 Rapportene som blir lagret skal ikke kunne forandres på

Faktor

Rapportene som blir generert av systemet er viktige dokumenter som kan bli brukt for å verifisere at utstyr faktisk har blitt sterilisert korrekt. På grunn av dette er det viktig at disse rapportene ikke kan forandres i ettertid. Det er i hovedsak prosessdata som kommer til å bli undersøkt da det skal sjekkes om noe har gått galt.

Vurdering

I utgangspunktet ville oppdragsgiver at vi skulle lagre skjermbildet som blir generert under en prosess som bildefil. Dette gjør det vanskelig å forandre på innholdet, men ikke umulig. I tillegg ville denne løsningen kreve en del plass i og med at bildefiler inneholder mye data.
Vi ønsker å lagre prosessrapportene i XML format. Dette gjør det ganske lett å forandre på verdier siden man bare trenger å åpne filen i en tekstbehandler og forandre det man vil. Vi har sett på måter å signere disse filene slik at vi merker om noen har rørt dem. En tanke var å sammenligne klokkeslettet den sist ble endret med klokkeslettet da den ble kjørt, men dette er ikke en god løsning siden filene da ikke kan flyttes. Det er i tillegg ganske lett å komme seg rundt denne sikkerheten.

Løsning

Løsningen vi valgte på dette problemet var å bruke en signatur nederst i filen, denne signaturen genererer vi ut i fra verdiene i filen. Vi bruker en enveis krypterings algoritme kalt MD5, denne genererer en signatur ut i fra en streng som input. Om noen prøver å forandre på noen av verdiene i filen kommer ikke denne summen til å stemme lenger og man merker at noe er galt. Selvfølgelig kan en person regne ut denne signaturen og også bytte den, men for å gjøre dette må han kjenne til hva vi bruker til å lage signaturen samt hvilken krypteringsalgoritme som blir brukt.

4.3.7 Systemet må kunne skille mellom flere pekeenheter

Faktor

Systemet må kunne vite hvilken skjerm på autoklaven man benytter for å kunne åpne riktig dør.

Vurdering

I utgangspunktet så vi på to mulige løsninger. Den første var å vise et vindu på hver skjerm, der alt var likt bortsett fra åpne dør knappen. Dette syntes vi virket som veldig tungvindt, i tillegg kunne det bli problemer med at begge skulle dele data.

Den andre løsningen var å finne ut hvilken skjerm som blir betjent gjennom funksjonaliteten i operativsystemet. Dette virket i utgangspunktet som den beste løsningen, siden vi da løste problemet der det er. Altså under funksjonaliteten til åpne dør knappen.

Løsning

I Windows XP la Microsoft til funksjonalitet for å lese rå data fra input enheter, blant annet kan vi da finne ut hvilken pekeenhet som sist ble brukt. De har enda ikke lagt til funksjonalitet for dette i .NET. Derfor må vi blande inn native kode for å bruke dette. Dette kan føre med seg minnelekkasjer så vi må være veldig nøye med at vi frigir minnet vi har allokert.

4.4 Process view

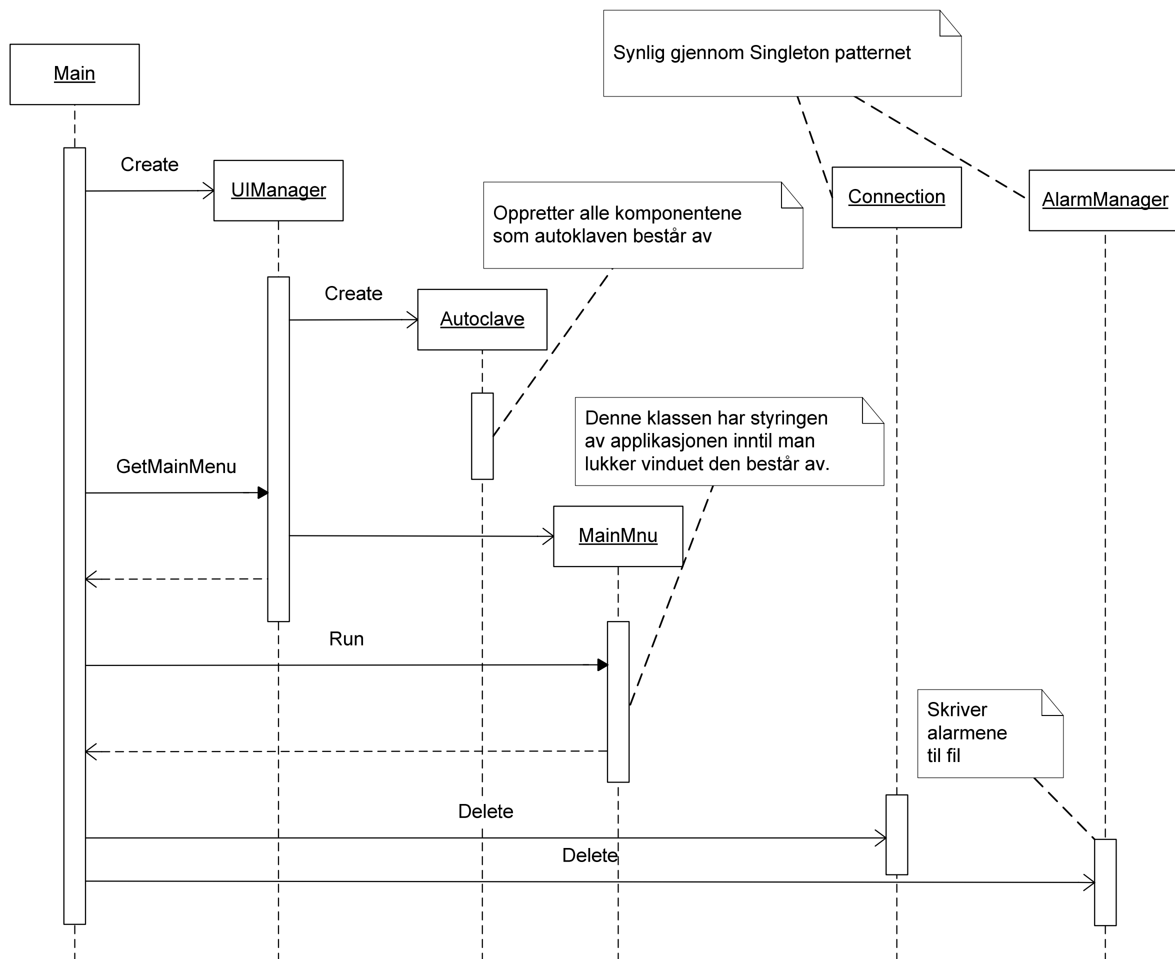
Dette avsnittet beskriver hvilke tråder og prosesser som finnes i løsningen. I tillegg beskriver den hvordan datautvekslingen foregår i løsningen.

4.4.1 Oppstart av applikasjonen

Det hjelper lite å ha massevis av klasser med mye funksjonalitet om det ikke finnes noe som setter i gang applikasjonen. En måte kunne være å opprette autoklav objektet først som igjen opprettet alle objekter i domenelaget og laget knyttingene til kommunikasjonspakken. Deretter kunne GUI starte opp og ta over kontrollen. Autoklav objektet er den klassen GUI bruker for å få kontakt ned i domenelaget. Om det ble gjort på denne måten ville vi måtte sende med autoklav objektet til GUI da denne ble opprettet.

Siden det kun er GUI som bruker autoklav objektet førte dette til en annen løsning der det er GUI om oppretter autoklav objektet. Dette er i samsvar med GRASP tankegangen om Creator der et objekt som bruker et annet burde opprette det.

Siden løsningen inneholder klasser som er basert på Singleton patternet må disse klassene få beskjed om når de skal destrueres. Dette må gjøres etter at brukeren har bedt om å få avslutte applikasjonen. Igjen finnes det to muligheter, å slette dem like før GUI-tråden terminerer eller slette dem etter at GUI-tråden er terminert. Å slette dem før GUI er avsluttet kan føre til at AlarmManageren, som er en av disse klassene, blir opprettet på nytt etter den er slettet om det oppstår software feil i løsningen. Derfor blir disse klassene slettet etter at GUI er avsluttet.



Figur 4-15: Oppstart og avslutning av løsningen

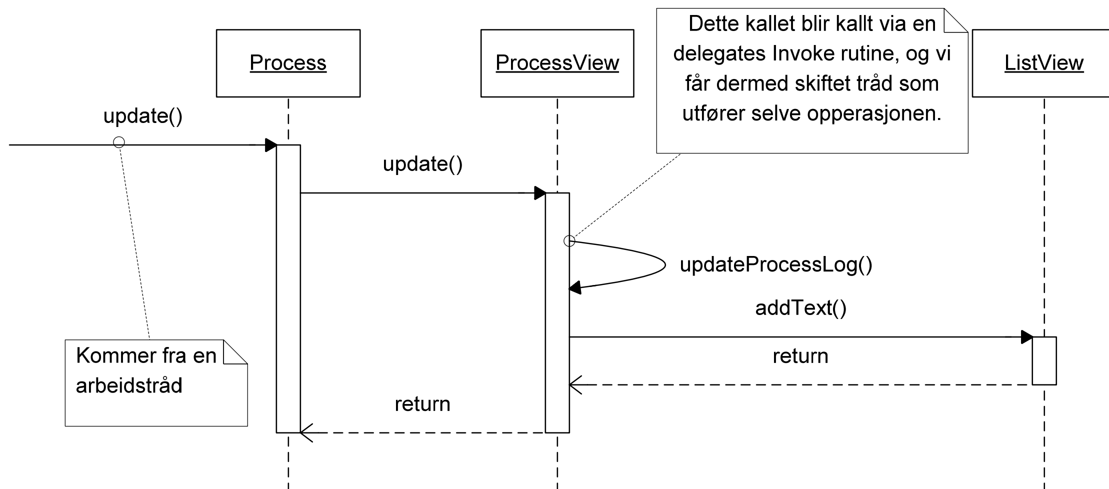
4.4.2 Tråder

Applikasjonen er veldig GUI fokusert og vi ønsker derfor at GUI skal svare på forespørsler under hele prosessen. Vi ønsker derfor å skille GUI og arbeidet i to tråder.

Driveren vi bruker til å kommunisere med busskobleren har mulighet for å gi oss beskjed om endringer og feil. Disse beskjedene kommer som kall til delegater i egne tråder. Hver enhet vi kobler oss til gjennom driveren får en egen tråd, det er disse som kaller delegatene våre. Vi har ikke mulighet for å få beskjed om nye endringer før vi har returnert fra vår kode til driveren. Vi har valgt å bruke denne funksjonaliteten, og dermed bygge opp løsningen veldig meldingsstyrt (hendelsesstyrt).

I noen subbprosesser skal vi ikke gjøre noe på en stund, altså bare vente. For å vite når vi skal gi oss med dette trenger vi å måle tiden. Dette kunne bli gjort via den tråden som ga beskjed om at vi skulle starte å vente. Dette viste seg å ikke fungere siden vi da ikke ville få returnert til driveren. Vi lager derfor en egen tråd som utfører nedtellingen i stedet slik at vi kan returnere til driveren med en gang.

En prosess kjører i utgangspunktet ikke i en egen tråd. Det opprettes et prosessobjekt som får meldinger om forandringer fra forskjellige tråder og utfører sin funksjonalitet ut i fra disse meldingene, altså meldingsstyrt.



Figur 4-16: Sekvens ved krysstrådkall

I .NET er det et skille mellom GUI tråder og arbeidstråder. Arbeidstråder får ikke lov til å gjøre forandringer i GUI komponenter. Vi har behov for å gjøre slike endringer for eksempel da vi skal legge til faseskifter eller lignende i et tekstfelt. Det er i utgangspunktet en av trådene fra driveren som gir GUI beskjed om at den skal forandre seg, men det er GUI tråden som må gjøre sleve forandringen. Dette løses ved hjelp av en delegat og krysstråkall som illustrert i Figur 4-16.

4.4.3 Synkronisering og deadlock

Siden det er en del tråder som kjører mot samme objekter og dermed samme minneområde er det viktig at vi har full oversikt over hvor det kan oppstå problemer om to tråder jobber mot minnet samtidig.

Da vi får beskjeder om forandringer på en eller annen sensor og skal opprette en ny melding til ovenstående systemer er det viktig at denne beskjeden blir ferdig prosessert før de kommer en ny melding. Vi får også samme problemet med alarmer, en alarm må være ferdig behandlet før vi får en ny alarm. Dette problemet har vi løst ved å angi områder der dette kan skje som kritiske seksjoner. Systemet lar derfor ikke flere tråder komme inn i denne rutinen samtidig.

Det kan også oppstå synkroniseringsproblemer når vi ber GUI tråden om å legge til tekst, her kan objektet den jobber mot bli skiftet ut før GUI tråden får lest den informasjonen den trenger. Dette er løst ved å bruke blokkerende kall som ::Invoke i stedet for ikkeblokkerende som ::BeginInvoke. Dette fører til at det er GUI tråden som utfører selve oppdateringen av komponenten, men tråden som ba om det venter til oppdateringen er ferdig før den fortsetter.

4.4.4 Datautveksling

I utgangspunktet henter vi den informasjonen vi trenger fra andre klasser, men i overføringen mellom kammerobjektet, prosessobjektene og GUI skal det sendes større mengde informasjon om gangen. Vi valgte å overføre denne informasjonen gjennom en egen klasse ProcessMessage. I tillegg har vi da mulighet for å flagge spesielle hendelser som fasebytte og prosesstatus. Dette objektet blir kun oppdatert da det finnes ny data. GUI bruker disse verdiene med jevne mellomrom og slipper da å hente dem nede i domenelaget. Andre steder i løsningen der vi henter en eller to verdier fra andre klasser, blir det mindre oversiktlig å bruke slike klasser enn å hente en og en verdi. Denne klassen blir også brukt til å sende data til dataloggeren. Vi bruker denne klassen fordi den inneholder all informasjon vi trenger for å logge en prosess.

Overføringen av informasjon fra konfigurasjonspakken foregår via en klasse som finnes i .NET kalt Dictionary, dette er et Hashmap som man kan spesifisere typen på både nøkkel og verdi. Konfigurasjonspakken ligger som et eget prosjekt og vil bli en egen dll. Ved å bruke innebygd funksjonalitet i .NET slipper vi at denne blir avhengig av selve applikasjonen. Ved å bruke en Dictionary trenger vi heller ikke gjøre forandringer i konfigurasjonspakken om man bytter ut de som bruker denne pakken. For eksempel kan man i kommunikasjonspakken skifte ut kontrollerkoden og gjøre forandring i konfigurasjonsfilene uten at dette krever forandringer i konfigurasjonskoden.

4.5 Implementation view

Det er jobbet for et godt design, noe som blant annet betyr høy styrke og lave koblinger. Ut i fra dette er det gjort en del valg rundt oppbygningen av løsningen. Dette avsnittet beskriver et utvalg av disse valgene.

4.5.1 Oppbygning av domenemodulen

Alternativ en var en klasse som vi kunne be om komponenter i autoklaven ved å kalle for eksempel getValve("F1") eller getSensor("BackupChamberSensor"). Denne var tenkt implementert via singleton patternet for å kunne være tilgjengelig fra alle klasser. Denne ville inneholde en instans av hver enhet i autoklaven. Fordelene med dette er at vi hadde fått et design som er lett å utvide, vi hadde også hatt alle enhetene vi styrte på samme sted. Vi måtte i tillegg skrive klasser for styring av disse komponentene.

Alternativ to var å bygge opp en logisk oppbygning der vi har en klasse kalt autoklav som under seg har alle de større enhetene i som vanntank og kammer. Disse består igjen av flere enheter som sensorer og ventiler. Fordelen med dette er at vi får en logisk oppbygning som ligner en del på hvordan det er i den virkelige verden. Vi får i tillegg et godt utgangspunkt for ansvarsfordeling. Hver enhet har ansvar for å styre de delene den består av. Denne løsningen fører til en del mer kode samt at den ikke er like utvidbar eller konfigurerbar.

Valget falt på alternativ to siden dette gir oss en mye mer robust løsning. Ansvarsfordelingen er også noe vi likte veldig godt med denne løsningen. Eierne av

komponentene er også de som oppretter dem. Dette er i samsvar med Creator tankegangen fra GRASP.

4.5.2 Ansvar for styring av prosessen

Det ble diskutert om hvem som skulle ha ansvar for å styre deler av prosessen. Måten å oppnå et bestemt trykk på kunne løses på flere måter. Den ene måten var å gi kammeret beskjed om å oppnå trykk 'X', og at kammeret tok seg av alt arbeidet med å oppnå trykket for så å gi beskjed da det er oppnådd. Dette virket i utgangspunktet som en utmerket løsning siden kammeret hadde kontroll over alle komponenter som blir styrt under en prosess. Men det viste seg at kammeret manglet en del prosesspesifikk informasjon som for eksempel hvor lang tid det skulle gå før den skulle gi opp å oppnå trykk. I tillegg ville dette gitt oss flere steder å vedlikeholde prosesskode.

En annen løsning kunne være å få tilgang direkte til sensorene og ventilene i prosessklassen og styre hele prosessen derifra. Dette virket heller ikke som en god løsning siden prosessen egentlig ikke trenger å vite noe om hvordan man for eksempel senker trykk, bare at den vil senke trykket. Det hadde ikke vært noen god ide å la både kammerklassen og prosessklassen styre de samme komponentene.

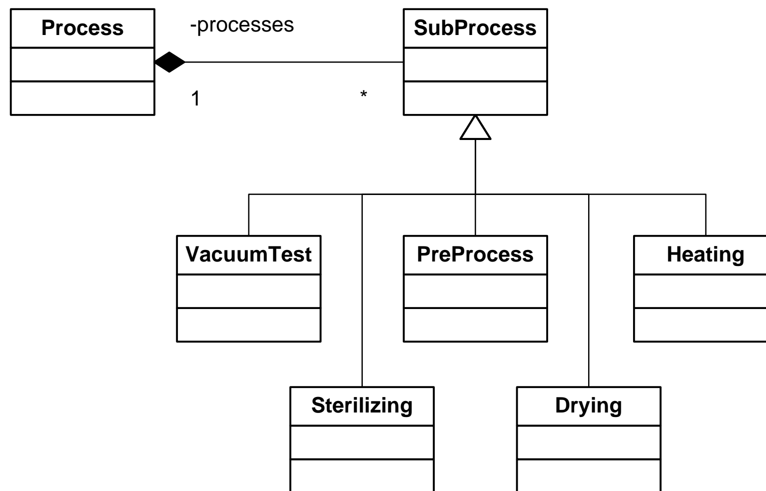
Dette førte oss til en tredje løsning der kammerklassen har funksjonalitet for å senke trykk, øke trykk, utligne trykk og lukke alle ventiler. Prosessen bruker disse metodene for å styre kammeret mot de trykk den vil oppnå. Med denne løsningen får vi klare skiller mellom ansvaret til de to klassene og vi får hele tiden beskjed i prosessen når kammeret forandrer tilstand.

I både løsning en og løsning tre løses avhengigheten mellom hvilke ventiler som skal brukes da trykket skal senkes i kammerklassen, mens i løsning to må dette styres fra prosessen.

Valget falt på løsning nummer tre grunnet den beste ansvarsfordelingen mellom de to klassene.

4.5.3 Oppbygning av prosess pakken

Vi kunne ha et prosessobjekt som styrte hele prosessen mot kammeret som beskrevet over. Men vi så at denne klassen kom til å bli veldig stor med all den funksjonaliteten som en prosess har. Vi undersøkte om vi kunne dele dette opp i flere klasser på en logisk måte. Det viste seg at vi kunne dele en vanlig prosess opp i fire deler med klare skiller. Vi la også merke til at vi kunne la vakuumtesten være av samme type som de delene vi delte prosessen opp i. Dette er en meget god løsning på grunn av at vi får delt opp ansvaret for de forskjellige delene som da kan testes individuelt. Inndelingen er vist i Figur 4-17.



Figur 4-17: Prosess og sub-prosesser

4.6 Data view

Denne seksjonen beskriver systemets behov for datalagring og hvordan dette er løst.

4.6.1 Konfigurasjonsinnstillinger

I løsningen er det behov for å lagre en del forskjellig data blant annet konfigurasjonsinnstillinger, prosesskonfigurasjon og språkfiler. Vi har valgt å lagre dette som XML filer siden dette er en åpen mye brukt standard med all den funksjonaliteten vi trenger. I tillegg er disse filene menneskelig leselige og har mulighet for hierarkiestisk oppbygning noe som også passet oss utmerket.

4.6.1.1 Prosesskonfigurasjon

Prosesskonfigurasjon lagres i en egen fil som vi bygger opp ut i fra samme tankegang som vi har brukt tidligere ved å dele opp en prosess i flere subprosesser. Altså en prosess består av en til mange subprosesser som igjen består av variablene og verdiene denne subprosessen trenger. Vi valgte å ha en fil for alle prosessene siden det aldri blir noen stor datamengde i denne filen. Sykehus kommer til å ha rundt 5 prosesser mens noen laboratorier kan ha opptil 15, men dette er ikke nok til at tolkingen av filene bruker for lang tid. Ellers lar vi alle verdiene som skal brukes i prosessen ligge i samme type tagger. Dette er gjort på denne måten for å ha en felles måte å lese alle disse verdiene på.

```xml
<?xml version="1.0"?>
<processes>
  <process name="BDTest">
    <subprocess type="preProcess"> ... </subprocess>
    <subprocess type="heating"> ... </subprocess>
    <subprocess type="sterilizing"> ... </subprocess>
    <subprocess type="drying">
      <attrib type="normal" />
      <attrib lowerPressure="150" />
      <attrib time="120" />
    </subprocess>
  </process>
  <process name="VacuumTest">
    <subprocess type="vacuumTest"> ... </subprocess>
  </process>
</processes>
```

4.6.1.2 Maskinkonfigurasjon

Maskinkonfigurasjonen er enklere bygd opp. Den inneholder kun de variablene vi ønsker å lagre med tilhørende verdier. Igjen har vi brukt attrib tagger for å få en felles måte å lese inn verdiene på. Denne filen har ikke en hierarkisk oppbygning siden informasjonen her ikke enkelt kan grupperes i blokker. Den er heller ikke så stor at mister oversikten ved en slik flat oppbygning.

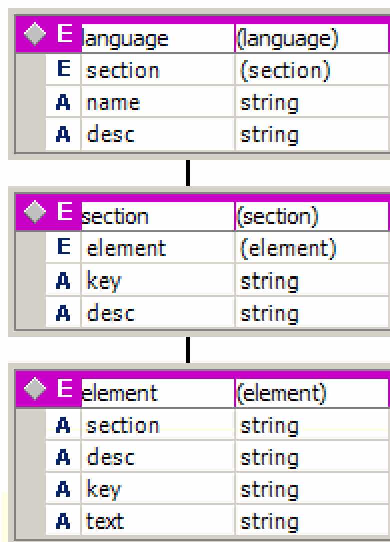
```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <attrib NumberOfDoors="1" />
  <attrib IpAddress="10.0.0.10" />
  <attrib Language="NO" />
  ...
</config>
```

4.6.1.3 Kontrollerkonfigurasjon

Kontrollerkonfigurasjonen er logisk bygd opp etter hva slags enheter en autoklav bruker. Her er det også tatt hensyn til at busskobleren kan byttes ut ved at det ikke er bestemt hva slags variabler og verdier som ligger i blokkene. I filen brukt i vår løsning lagrer vi de PhoenixContact spesifikke verdiene vi benytter. Disse kan være helt forskjellig om man benytter en annen busskobler. Men oppbygningen består alltid av en blokk med analoge innganger, en blokk med digitale utganger, en blokk med digitale innganger, en blokk med variabler for busskobleren og en blokk med Multiplexer enheter.

```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <analoginputs> ... </analoginputs>
  <digitaloutputs> ... </digitaloutputs>
  <digitalinputs> ... </digitalinputs>
  <busconnector> ... </busconnector>
  <pcpdevices> ... </pcpdevices>
  <muxdevices> ... </muxdevices>
</config>
```

4.6.1.4 Språkkonfigurasjon

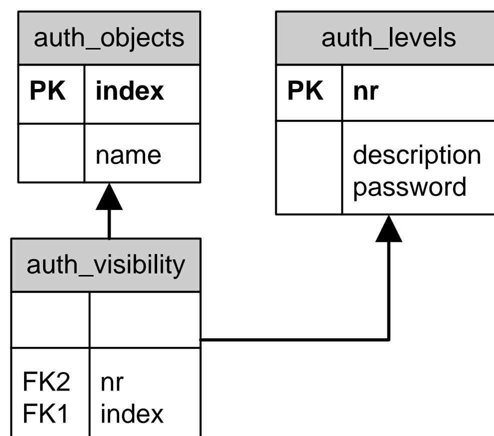


Figur 4-18: Relasjoner i språkfiler

Vi lagrer også språkfiler som XML dokumenter, her har vi valgt å ha et dokument for hvert språk. Dette er gjort siden disse filene fort blir store og dermed blir det mer oversiktlig med en fil per språk. I tillegg gjør dette jobben lettere for oversettere ved at de har et dokument å forholde seg til per språk. Filene får navn ut i fra hvilket språk de inneholder. resource.NO.xml inneholder for eksempel språkoppsett for norsk språk. Siden disse filene blir behandlet av tredjepart har vi her valgt å validere dokumentet før vi leser det inn. Dette er gjort ved hjelp av et XML Schema som vist i Figur 4-18. Vi bruker funksjonaliteten i .NET for valideringen. Som vi ser inneholder dette en del mer data en vi bruker i løsningen vår. Dette er data som kan bli brukt for å orientere seg i dokumentet da man oversetter. I tillegg kan dette brukes om man lager en applikasjon som man bruker under oversettelsen.

4.6.1.5 Autentisering

De siste konfigurasjonsfilene beskriver autentisering, altså hvem som har tilgang til hva i løsningen. Denne konfigurasjonen består av tre filer en for alle brukernivåene, en for alle komponentene som kan styres og en for å knytte disse sammen. Dette er løst på denne måten for å kunne dynamisk angi antall brukernivåer og for å gi full kontroll over hva som er tilgjengelig for hvem.



Figur 4-19: Relasjoner autentisering

4.6.2 Prosess- og alarmdata

Prosessloggen og alarmlisten kan vokse til en veldig stor datamengde. På grunn av dette blir de ikke håndtert på samme måte som konfigurasjonsinnstillinger. Som beskrevet tidligere er løsningen designet uavhengig av hvordan vi lagrer, men det trengs i alle fall en konkret implementasjon for at løsningen skal kunne kjøre.

Valget sto mellom å lagre i database og å lagre som filer på disken. Databaser er mer teknologisk komplekse og hadde ført til mer administrativt arbeid. Det gjør oss også avhengig av et tredjeparts program. Dermed falt valget på lagring som filer. Som filtype falt igjen valget ganske enkelt på XML av samme grunner som beskrevet i seksjonen over.

Filene for prosesslogg inneholder et utvalg av beskjedene fra kammeret til prosessen, dette utvalget er nok for å kunne reprodusere prosessen. I tillegg finner vi en del ekstra informasjon om hvem som signerte prosessen, hva som ble steriliser, en id, tidspunktet og et felt for sikkerhet.

```xml
<?xml version="1.0"?>
<Process>
  <Id>VacuumTest200652111021</Id>
  <Batch>
    <Item>Saks</Item>
  </Batch>
  <ApprovedBy>Audun</ApprovedBy>
  <Date>02.05.2006</Date>
  <ProcessEvents>
    <Event>
      <Time>11:10:37</Time>
      <Pressure1>1004.133</Pressure1>
      <Pressure2>999.4667</Pressure2>
      <Temperature1>15.51</Temperature1>
      <Temperature2>15.51171</Temperature2>
      <PhaseChange>true</PhaseChange>
      <Message>Reduserer trykk</Message>
    </Event>
    <Event>
      ...
    </Event>
    ...
  <Version>D5-89-6E-AD-32-22-F5-DB-31-DF-21-2E-15-CB-1D-36</Version>
</Process>
```

Lagringen av alarmene foregår på samme måte, bortsett fra at her lagrer vi alle alarmene. Informasjonen som blir lagret om alarmene er tidspunktet, hvor kritisk den er, feilkoden samt feilmeldingen. Under ser vi et eksempel på hvordan alarmene blir lagret som XML.

```xml
<alarms>
  <alarm>
    <time>02.05.2006 10:48:38</time>
    <level>1</level>
    <errorcode>2731</errorcode>
    <message>No water in boiler</message>
  </alarm>
  <alarm>
    <time>02.05.2006 12:07:02</time>
    <level>1</level>
    <errorcode>2731</errorcode>
    <message>No water in boiler</message>
  </alarm>
</alarms>
```

5. Implementasjon

I dette kapittelet blir et utvalg av selve implementeringen av løsningen beskrevet. Det blir beskrevet hvordan kravene og designet er blitt løst i koden. Kapittelet viser frem hovedtrekk og løsninger som krever mer beskrivelse enn kommentarene i koden. Underkapitlene er inndelt etter pakkene i systemet. Se avsnitt 4.2 for detaljer.

5.1 *AH::Pride::Domain*

Felles for domenelaget er at klassene har innebygd funksjonalitet for å kjøre seg selv uavhengig av brukeren av systemet. Chamber styres av en prosess som bruker Chamber sine metoder for å kontrollere det. Felles for klassene er at de har en update() funksjon som kalles av callback fra sensorene. Dette kan være analoge eller digitale sensorer. Når en sensor har forandret verdi vil den si i fra til objektet den ligger under ved å kalle update() funksjonen.

5.1.1 Update og faser

I update() er alle fasene som trengs for å kontrollere objektet ut i fra de verdiene som kommer fra sensorene. Når sensorenes verdi oppfyller satte kriterier, vil fasen byttes. Neste gang update() kalles kjøres den nye fasen. De forskjellige fasene lages ved hjelp av en `switch() case:` rutine som switcher på en variabel som kalles `currentPhase`. Fasene defineres med `static const int FASENAVN = 1;`
Den største ulempen med dette er at update() ikke blir kjørt hvis en sensor ikke har forandret sin verdi. Dette har vi tatt hensyn til og laget systemet slik at hendelser som skal skje ut i fra tidsutmålinger bruker en tråd til å gi et signal om når hendelsen skal utføres. For at ikke flere sensorer skal forandre verdier i klassen samtidig ved at mange har kalt update() på samme tid, låses objektet da en tråd kommer inn i update() metoden. Dette gjøres ved å låse objektet med en monitor som vist under.

```
void Mantle::update()
{
  // lock this object
  System::Threading::Monitor::Enter(this);
  // ... //
  System::Threading::Monitor::Exit(this);
}
```

5.1.2 Alarmer

En alarm består av en feilkode, et nivå, et tidspunkt og en melding. Alarmenes nivå og feilkode er definert i Alarm klassen slik at det ikke er behov for å slå opp disse noe sted. Det brukes for eksempel Alarm::CRITICAL for å angi at nivået til alarmen er kritisk. Alle styrene klasser i systemet skal ha mulighet for å generere alarmer. Klassen AlarmManager er derfor tilgjengelig for alle klasser i hele løsningen.
Dette er blitt løst ved hjelp av singleton patternet for å slippe å sende instansen til alle som trenger den. Alarmer genereres ved å opprette en alarm og sende denne til AlarmManager.

```
// Give an alarm
AlarmManager::getInstance()->addAlarm(
    gcnew Alarm(Alarm::CRITICAL,
    Alarm::ERROR_PRESSURE_CHANGED_TOO_MUCH,
    "Pressure outside of allowed area"));
```

AlarmManageren tar seg av å sende alarmen til alle som er interessert i å få vite om alarmer, samt å logge alle alarmene. Formidlingen av alarmer til alle interessenter foregår via en event på AlarmManageren som man kan legge seg til og dermed få beskjed hver gang en ny alarm blir lagt til.

```
AlarmManager::getInstance()->onChange +=
gcnew AlarmManager::ChangeCallback(this, &VacuumTest::alarmHandler);
```

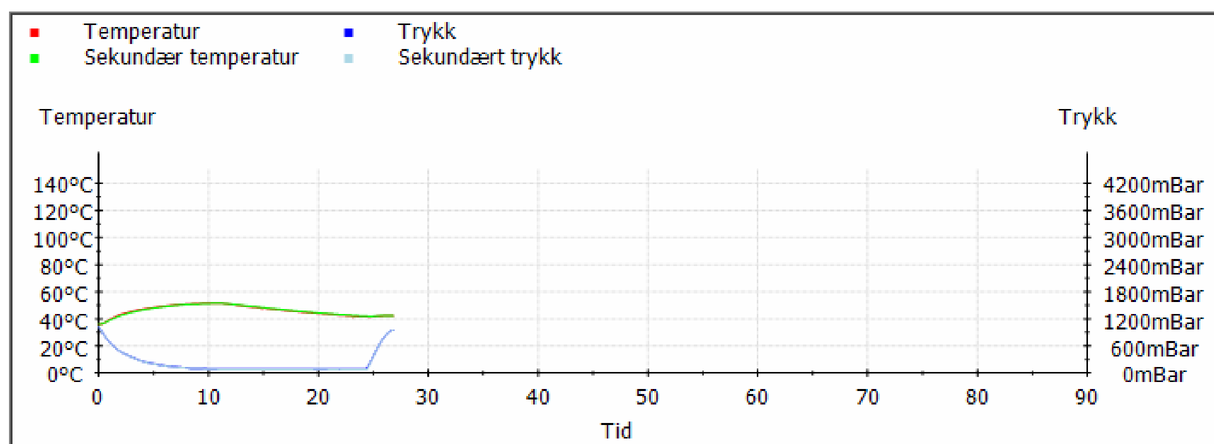
Lagringen og lesing av gamle alarmer foregår via IO pakken og AlarmManageren er dermed ikke avhengig av noe bestemt lagringsformat for alarmene

5.2 *AH::Pride::Domain::Processes*

Denne pakken tar seg av representasjon og kjøring av prosesser samt logging av prosessen.

5.2.1 Subprosesser og faser

Sub-prosessen VacuumTest er brukt som eksempel. Ved endt prosess ser prosessbildet ut som vist i Figur 5-1. Først pumpes trykket ned til vakuum. Så kommer en stabiliseringstid slik at trykk og temperatur oppnår en likevekt i tilfelle det er elementer som vann eller vanndamp i kammeret som kan forstyrre trykket. Etter stabiliseringstiden kjøres selve testen. Til slutt trykkutjevnes kammeret ved at det slippes luft inn i kammeret igjen.



Figur 5-1: Grafisk fremstilling av prosessforløpet

De fire handlingene deler vakuumtesten opp i tilsvarende fire faser. En fase der vi pumper ut luften fra kammeret, en fase for å stabilisere, en fase for å teste og til slutt en fase for å trykkutjevne kammeret igjen. I tillegg finnes det en avbrutt-fase som trykkutjevner kammeret om noe går galt. Faseinndelingen er gjort på grunn av at det er spesifikke ting som skal skje ut i fra hver tilstand.

5.2.2 Kjøring av prosessen

Prosessen blir kjørt ved å gi kammeret instruksjoner. Kammeret får for eksempel instruksjon om å senke trykket. Da bestemmer kammeret selv hva som skal gjøres for å senke trykket. Et eksempel fra vakuumtesten er vist under. Her får kammeret beskjed om å senke trykket.

```
void VacuumTest::start()
{
  // ... //
  // tell the Chamber to start decrementing the pressure
  chamber->decPressure();
}
```

Når subprosessen får en oppdatering fra kammeret leser subprosessen inn et ProcessMessage objekt fra kammeret der sensorenes verdier ligger. Hvordan fasen videre skal kjøres bestemmes ut fra verdiene i ProcessMessage. ProcessMessage

objektet blir i slutten av subprosessen oppdatert med verdier som skal sendes opp til Prosess objektet. Det er viktig at ikke to tråder kommer inn i denne oppdaterings rutinen samtidig siden det kan resultere i doble fasebytter. Løsningen på dette er som beskrevet i avsnitt 4.4 ved at det benyttes en Monitor for å låse objektet mens en tråd er i oppdaterings rutinen. Låsen blir frigjort ved å bruke en try, catch, finally blokk. Dette gjør at låsen *alltid* blir frigjort selv om det kommer en exception i programmet mens objektet er låst.

```
void VacuumTest::update()
{
  // lock this object
  System::Threading::Monitor::Enter(this);
  try{
    // read the pressure
    currentMessage = chamber->getMessage();
    // what phase are we in?
    switch(currentPhase){
    case PHASE_DOWN:
    //...//
    case PHASE_STABILIZE:
    //...//
    case PHASE_TEST:
    //...//
    case PHASE_UP:
    //...//
    case PHASE_ABORT:
    //...//
    // Tell everyone it has changed
    notify();
  }catch(Exception ^e){
    //...//
  }finally{
    // release lock
    System::Threading::Monitor::Exit(this);
  }
}
```

Dette er et utmerket eksempel på hvorfor et hendelsestyrt system er bedre en et sekvensiellt system for et autoklave kontroller system. Hvis subprosessen ikke hadde fått beskjed om endringer fra kammeret ville den måtte polle disse verdiene konstant og dermed brukt mye av prossessortiden i denne rutinen. Med hendelsesstyrt løsning brukers kun det som er nødvendig.

5.2.3 Fase- og subprosessbytter

Når det skjer et fasebytte settes currentPhase til den nye fasen slik at neste oppdatering blir håndtert i henhold til den nye fasen. Det kan også settes et flagg for fasebytte, og en beskjed om hva som skjedde i ProcessMessage.

Et flagg er implementert i ProcessMessage som et og et bit i et statusord. ProcessMessage er objektet som brukes for meldingsutveksling mellom lagene og da passer det utmerket å implementere denne informasjonsutvekslingen på denne måten.

Dette flagget blir lest i overforstående lag som dermed får de beskjed om at det har vært et fasebytte. Dette blir beskrevet nøyere i avsnitt 5.6.1. Under vises et slikt fasebytte. Dette er fra vacuumtesten i overgangen mellom fasen som senker trykket og fasen som stabiliserer kammeret. Her settes også flagget for fasebytte og en beskjed om hva som har skjedd.

```
case PHASE_DOWN:
  // check if the pressure is below the one we want
  if(currentMessage->getPressure() <= setpointPressure){
    // seal the chamber
    chamber->stopPressure();
    // start PHASE_2
    currentPhase = PHASE_STABILIZE;
    // set the message
    currentMessage->setMessage(
      resource->getString("Domain.ProcessMessage:_stabilizing"));

    // set the phase changed var
    currentMessage->setPhaseChange(true);
    //...//
break;
```

Skiftet mellom subprosesser tar Process klassen seg av. Når det får en oppdatering med SubProcessFinished flagget satt, sjekker den om det finnes flere subprosesser i denne prosessen. Om det finnes flere subprosesser i gjeldene prosess stopper vi den vi kjører og starter den neste. Om gjeldene subprosess er siste subprosess settes ProcessFinished flagget.

```
void Process::update()
{
  // read the message
  currentMessage = subProcesses[currentSubProcess]->getMessage();
  // check if we current subprocess is done
  if(currentMessage->getSubProcessChange()){
    // stop observing current subprocess and change to next
    SubProcesses[currentSubProcess]->onChange -=
    gcnew SubProcess::ChangeCallback(this, &Process::update);
    //...//
    currentSubProcess++;
    // are we done with all subprocesses?
    if(currentSubProcess < subProcesses.Count){
      // call onChange so this message are
      // handled before next subprocess starts
      onChange();
      // start observing
      subProcesses[currentSubProcess]->onChange +=
      gcnew SubProcess::ChangeCallback(this, &Process::update);
      // start the next subprocess
      subProcesses[currentSubProcess]->start();
    }else{
      // set the flag that means process done
      currentMessage->setProcessFinished(true);
      // running flag is now false
      running = false;
      // tell everyone we have a update ready
      onChange();
    }
  }
}
```

I tillegg til fasebytter på bakgrunn av oppdateringer fra kammeret kan fasebytter skje ut i fra intern timing i subprosessene. Vakuumtesten skal for eksempel under stabiliseringsfasen vente en gitt tid før selve testen starter. Om kammeret er fullstendig tett vil det ikke komme oppdateringer, derfor startes en tidtaking samtidig som stabiliseringsfasen startes. Når tiden er utløpt er stabiliseringsfasen over og fasen skal byttes til testfasen.

5.2.4 Logging av prosessen

Prosessen blir av styrt av ProcessMessage objekter som sendes fra kammeret og opp i aktiv subprosess. For å kunne reprodusere hva som skjedde under en prosess kan disse objektene lagres etter hvert. For å reprodusere grafen brukes et ProcessMessage objekt per bildepunkt i selve grafbildet. Vi valgte å droppe punkter ut i fra tid, altså vi lagrer til et punkt per tidsenhet. I tillegg er det viktig å få med de objektene der det skjer noe spesiellt, for eksempel om det kommer en tekst som skrives ut i loggen til venstre i prosessbildet. Disse beskjedene logges uavhengig av hvilket tidspunkt de kom på. Selve skrivingen til disk er det IO pakken som tar seg av, se avsnitt 4.2.2.8 IO.

5.3 AH::Pride::IO

Denne pakken har ansvaret for å lese og skrive prosessloggene og alarmlisten til ekstern lagring. Pakken inneholder også interfacene som må implementeres hvis funksjonaliteten til lagringen skal byttes ut. Dette er gjort i henhold til Adapter patternet i GangOfFour[3].

5.3.1 Prosesslogg

I denne løsningens konkrete implementasjon av disse interfacene foregår lagring av prosessen ved at en liste med ProcessMessage objekter lagres til en XML fil. I tillegg til ProcessMessage objektene lagres informasjon om hvem som signerte prosessen og hva som blir sterilisert.

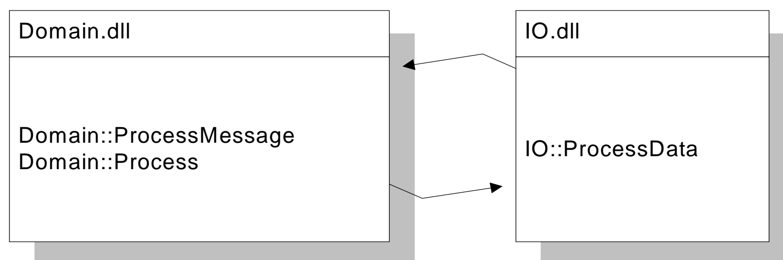
```
<?xml version="1.0"?>
<Process>
  <Id>VacuumTest200652111021</Id>
  <ApprovedBy>Audun</ApprovedBy>
  <Date>02.05.2006</Date>
  <ProcessEvents>
    <Event>
      <Time>11:10:37</Time>
      <Pressure1>1004.133</Pressure1>
      <Pressure2>999.4667</Pressure2>
      <Temperature1>15.51</Temperature1>
      <Temperature2>15.51171</Temperature2>
      <PhaseChange>true</PhaseChange>
      <Message>Reduserer trykk</Message>
    </Event>
    <Event>
      ...
    </Event>
    ...
  <Version>D5-89-6E-AD-32-22-F5-DB-31-DF-21-2E-15-CB-1D-36</Version>
</Process>
```

Over er et utsnitt av en lagret vakuumtest. En steriliseringsprosess der det er angitt hva som blir sterilisert vil i tillegg ha en seksjon som heter Batch. Testens ID og dato lagres også.
I version taggen lagres en MD5 sum av all data i dokumentet. Dette er for å signere dokumentet. Det vil gi en tilstrekelig sikkerhet for validering av dokumentet når det

[3] *Omtalt av Craig Larman 2005 – Applying UML and Patterns*

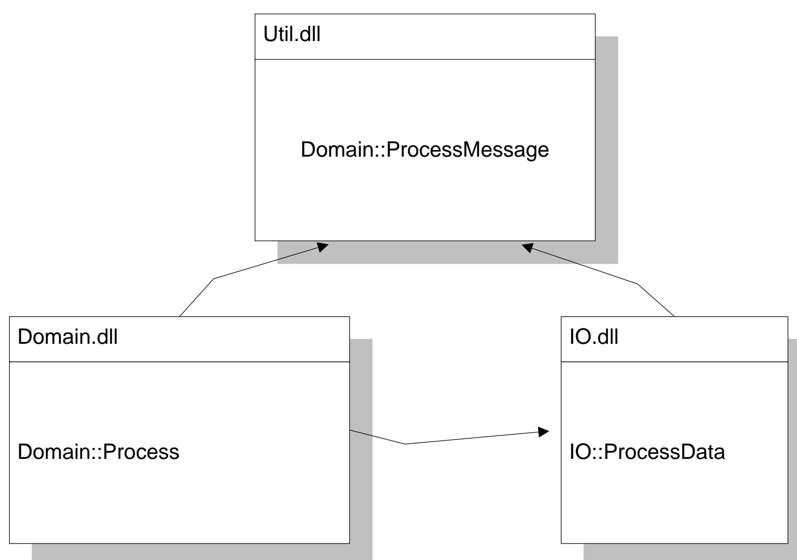
leses inn igjen. Dette det gjør det meget vanskelig å forandre verdier etter at dokumentet er lagret. Se avsnitt 4.3.6 for bakgrunn.

Når prosessene leses inn igjen fra fil genereres listen med ProcessMessage objekter ut i fra dataen i XML filen. Resten av informasjonen i dokumentet leses også inn. Etter dette er lest inn sjekkes det om MD5 summen stemmer. På denne måten merker systemet om det er gjort noe med dokumentet.



Figur 5-2: Kall mellom Domain.dll og IO.dll

Siden ProcessData - klassen som skriver prosessloggen til fil - skal kunne skiftes ut burde dette plasseres i en egen dll. Det burde gjøres slik for da er det enkelt å bytte den ut. Denne dll'en skal bruke ProcessMessage objekter og blir da avhengig av dllen til domenelaget. Dll'en til domenelaget er avhengig av klassene i IO siden de brukes derifra. Dette fører til sirkulær avhengighet som vist i Figur 5-2, og er ikke mulig å kompilere. Derfor er ProcessMessage flyttet ut i en egen dll som de to andre er avhengig av som vist i Figur 5-3.



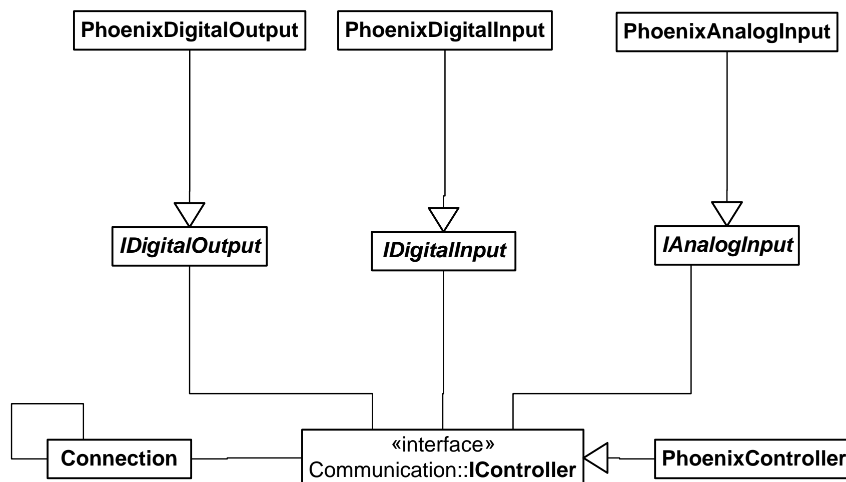
Figur 5-3: Kall mellom Domain.dll, Util.dll og IO.dll

5.4 AH::Pride::Communication

Denne pakken tar seg av kommunikasjonen med busskobleren. Den inneholder også de abstrakte klassene man må implementere for å kunne bytte ut busskobleren.

5.4.1 Adapter

Pakken er designet slik at busskobleren kan byttes ut uten å gjøre forandringer i de andre pakkene. Det er definert en del klasser som må implementeres i kontrollklassen til andre busskoblere. Dette er interface klassene IController, IAnalogInput, IDigitalOutput og IDigitalInput. Dette er i henhold til Adapter patternet til Gang of Four [4]. Disse klassene er abstrakte og det kan dermed ikke opprettes noen konkrete instanser av dem. De brukes som interface mellom resten av systemet og communication pakken. Se Figur 5-4 for hvordan dette er løst.



Figur 5-4: Adapter-pattern i Communication

5.4.2 Busskobleren og digitale inn- og utganger

Busskobleren som brukes i dette systemet er laget av PhoenixContact. Implementasjonen av IController i dette systemet heter dermed PhoenixController. IController krever at rutinen connect() er implementerert, samt en rutine for å lage hver av de tre andre abstrakte klassene.
Connect() ber driveren om å koble til, samt å initialiserer multiplexeren. Denne er beskrevet i avsnitt 5.4.5. Selve oppsettet av tilkoblingen til busskobleren blir gjort i constructoren til PhoenixController.

I klassen er det implementert get funksjoner som setter opp koblinger til de analoge og digitale innganger og utganger som finnes på busskobleren. getDigitalOutput oppretter en PhoenixDigitalOutput med parametere den henter fra Configuration pakken. Den registreres i controlleren og ber dermed om å få beskjed da verdien

[4] *Omtalt av Craig Larman 2005 – Applying UML and Patterns*

forandrer seg. Pekeren til det nyopprettede objektet returneres til den som kalte getDigitalOutput og utgangen er dermed initialisert i systemet.

```
IDigitalOutput ^PhoenixController::getDigitalOutput(String ^name){
 int baseAddress, length, offset;

 // get the values
 Dictionary <String^, String^> ^dic = config->getDigitalOutput(name);

 // store them
 baseAddress = int::Parse(dic["outputbaseaddress"]);
 length = int::Parse(dic["outputlength"]);
 offset = int::Parse(dic["outputoffset"]);

 // create the instance
 PhoenixDigitalOutput ^o = gcnew PhoenixDigitalOutput(baseAddress,
                                                      length,
                                                      offset);
  // add the object from the driver
  fl_bk->AddObject(o->getVarOutput());
    // add the callback
    o->getVarOutput()->OnChange +=
             gcnew VarChangeHandler(o, &IDigitalOutput::sendChange);
    return o;
}
```

Metoden for å be om digital input er tilsvarende, mens analog input har en del mer funksjonalitet. Det settes et kontrollord i tillegg til at verdien inn skal skaleres og korrigeres ut i fra kalibrering.

5.4.3 Analog inngang

Som beskrevet i forrige avsnitt skal en analog inngang både skaleres og korrigeres for kalibrering. Skaleringen brukes til å tilpasse de analoge verdiene som kommer fra kontrolleren slik at de stemmer med den målte verdien. Fra PT100 element, som måler temperatur, vil temperaturen komme ut som en heltallsverdi siden driveren ikke støtter flyttall ved lesing av verdier. Det settes en parameter i kontrollordet som sier hvor mange desimaler målingen skal gjøres med. Ved en nøyaktighet på 2 desimaler kan tallverdien 10000 leses fra sensoren når PT100 elementet måler 100grC. En skaleringsfaktor på 100 vil dele 10000 på 100 og målt verdi 100,00 vises i systemet.

Sensorene må også kalibreres. Siden sensorens feil vil være lineær vil kalibreringen av sensoren også gjøres lineært. Det lages en rett linje ut i fra avviket to steder i måleområdet. Ligningen for en rett linje er $y = a \cdot x + b$ og da blir formlene for å korrigere avviket som vist i Figur 5-5.

$$\Delta bunn = faktisk_bunn - målt_topp$$
$$\Delta topp = faktisk_topp - målt_topp$$
$$a = \frac{\Delta topp - \Delta bunn}{målt_topp - målt_bunn}$$
$$b = \Delta bunn - a \cdot målt_bunn$$

Figur 5-5: Beregning av kalibreringsfaktor

5.4.4 Callback

Alle de tre inn- og utgangene har events slik at systemet kan få beskjed når de forandrer verdi. Dette er løst ved hjelp av events i .NET. Denne kan settes til å peke på en metode i det objektet som er interessert i å få beskjed om forandringer i enheten.

```
/// the declaration of the callback method signature
delegate void Callback();
/// the actual callback method pointer
event Callback ^onChange;
```

Denne eventen blir kalt hver gang sensoren forandrer verdi og dermed får alle som er interessert beskjed om dette.

```
void PhoenixDigitalOutput::sendChange(Object ^sender){
  value = (int)output->Value;
  onChange();
}
```

5.4.5 Multiplekser

En type moduler fra PhoenixContact tilbyr muligheten for å kjøre multiplekset adressering på bussen. Dette er for å spare minneplasser som brukes til å adressere enhetene på bussen. I dette systemet brukes en slik enhet til å lese inn verdier fra 6 temperatur sensorer og 2 nivå sensorer. Systemet er designet for å lett kunne koble til flere slike enheter. Disse enhetene må behandles spesielt og denne funksjonaliteten ligger i klassen PhoenixMultiplexer.

PhoenixController klassen setter opp multiplekseren ut ifra konfigurasjoninstillinger. Her konfigureres enhetene som multiplekses. Det opprettes et PhoenixMultiplexer objekt for hver enhet som ligger under seksjonen <muxdevices>. Der leses konfigurasjonsdata inn og multiplekseren startes opp. I dette systemet er det enheten 4/8RTD som multiplekses og den har navnet RTD1 i konfigurasjonsfilen.

```
<muxdevices>
  <muxdevice name="RTD1">
    <attrib inputbaseadress="2"/>
    <attrib outputbaseadress="2"/>
    <attrib multiplexers="2"/>
    <attrib channelspermux="4"/>
    <attrib selectmultiplexers="0800;0900"/>
    <attrib setcontrolwordmask="4"/>
  </muxdevice>
</muxdevices>
```

I eksemplet over vises 4/8RTD konfigurert med navn RTD1. Den er koblet på inputadresse 2 og outputadresse 2. Den har 2 multipleksere med 4 innganger hver.

Når PhoenixController får forespørsel om å opprette en analog sensor sendes navnet på den ønskede sensoren med i funksjonskallet. Den ber da om verdiene fra konfigurasjonspakken med dette navnet.

```
<input name="ChamberTempSensor1">
  <attrib type="MUX"/>
  <attrib muxdevice="RTD1"/>
  <attrib channelnr="1"/>
  <attrib multiplexer="1"/>
  <attrib length="16"/>
  <attrib controlword="0040"/>
  <attrib scalingfactor="100"/>
  <attrib measuredValueMax="29695"/>
  <attrib realValueMax="300.0"/>
  <attrib measuredValueMin="700"/>
  <attrib realValueMin="10.0"/>
</input>
```

"Type" attributtet bestemmer den videre flyten i programmet. Er `attrib type="MUX"` legges sensoren til den multiplexer enheten som har tilsvarende navn som er konfigurert i `attrib muxdevice="RTD1"`. Objektet av typen PhoenixMultiplexer som er konfigurert for RTD1, tar da over og oppretter sensoren.

Sensoren "ChamberTempSensor1" blir lagt til i en array av sensorer som skal multiplekses på denne enheten. Multiplekseren genererer så callback for sensoren med gitte tidsintervall. Dette blir istedenfor å få callbacket fra PhoenixContact driveren for denne enheten. Callbacket genereres på denne måten for å være helt sikker på at hver sensor får gitt beskjed om forandret verdi.

Et forbedringspotensial i multiplexeren er å la sensoren selv utføre oppdateringen. Dette blir mer komplisert siden det da ikke er mulig å finne ut om en sensor har forandret seg mens en annen sensor bruker samme minneadresse. I verste tilfelle kan det risikeres at sensoren forandrer verdi bare mens den ikke er koblet til og dette anses som verre en å få noen oppdateringer for mye. Måten dette blir håndtert på fra PhoenixContact sin side kan bli forbedret i senere versjoner av driveren. Forhåpentligvis vil denne funksjonaliteten bli totalt integrert i driveren.

5.5 AH::Pride::Config

Denne pakken tar seg av å lese og skrive konfigurasjonsdata. All konfigurasjonsdata finnes i XML filer. Valget av XML er på grunn av at dette er en åpen standard med all den funksjonaliteten vi trenger. Alle klassene som brukes til å lese disse innstillingene arver fra .NET's XMLDocument som inneholder funksjonalitet for å tolke XML filer. Se avsnitt 4.2.2.7 for detaljer om oppbygning.

5.5.1 Finne riktig seksjon

For å finne frem til riktig sted i XMLdokumentene brukes XPath. Dette er en måte å beskrive hvor i dokumenttreet vi vil lese informasjon. Som et eksempel kan vises hvordan lesing av konfigurasjon om en digital utgang fungerer. Digitale utganger ligger under digitaloutputs i filen som vist under.

```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <analoginputs> ... </analoginputs>
  <digitaloutputs>
    <output name="F1">
      <attrib outputbaseaddress="24"/>
      <attrib outputoffset="0"/>
      <attrib outputlength="1"/>
    </output>
    ...
  </digitaloutputs>
  <digitalinputs> ... </digitalinputs>
  <busconnector> ... </busconnector>
  <pcpdevices> ... </pcpdevices>
  <muxdevices> ... </muxdevices>
</config>
```

Taggene som begynner på "attrib" vil være spesifikke for hvilken type busskobler som blir benyttet. På busskobleren fra PhoenixContact brukes disse tre verdiene for å identifisere en digital utgang. For å lese disse brukes XPath for å finne frem til riktig output blokk og velge alla taggene i denne. For en digital utgang som heter F1 blir XPath søkestrengen

```
"/config/digitaloutputs/output[@name='F1']/attrib"
```

Alle taggene i denne blokken lagres til en dictionary og returneres.

```
Dictionary <String^, String^>^
                         ControllerConfig::getDigitalOutput(String ^name){
   return getHashMap("digitaloutputs", "output", name);
 }

 Dictionary <String^, String^> ^ControllerConfig::getHashMap(String ^type,
                                                             String ^tag,
                                                             String ^name){
   // used to hold the information about the tags
   Dictionary <String^, String^> ^dic =
                                 gcnew Dictionary<String^, String^>();
   // pull things out of document using XPath
   String ^search = "/config/"+type+"/"+tag+"[@name='"+name+"']/attrib";
   XmlNodeList ^list = SelectNodes(search);
   // in the list we now have all the attribs for the correct name
   for(int i = 0; i < list->Count; i++){
     String ^key = list[i]->Attributes[0]->Name;
     String ^val = list[i]->Attributes[0]->Value;
     dic->Add(key, val);
   }
   return dic;
 }
```

getHashMap metoden brukes til selve innhentingen av data. Når en digital utgang opprettes får getHashMap informasjon om hvor den skal hente informasjonen og foretar selve søket. Dette er gjort på denne måten på grunn av at søket fungerer på akkurat samme måte når informasjon om digitale innganger og analoge innganger leses inn. Hentingen av data i de andre klassene foregår på samme måte, ved at den søkes frem ved hjelp av XPath og returneres til den som ba om det.

5.6 AH::Pride::GUI

En stor del av GUI er beskrevet i kapittel 4. Dette er gjort fordi det var vanskelig på skille implementasjon og design for denne pakken.

5.6.1 Kryssstrådkall

Som beskrevet i avsnitt4.4.2 er det forskjell på GUI-tråder og arbeidstråder (workerthreads). Da systemet er hendelsesstyrt betyr det at GUI får Callbacks fra slike arbeidstråder når det skjer endringer i systemets domenelag. Disse arbeidstrådene har ikke tilgang til komponenter og elementer som eies av GUI-tråden. For å løse dette må arbeidstråden delegere arbeidet over til GUI-tråden som selv utfører operasjoner og endringer på elementer og komponenter.

Tid:	Temp:	Trykk:	Status:
00:00:00	328,32°C	968mBar	Decreasing pressure
00:00:46	328,32°C	65,8mBar	Increase pressure
00:01:03	328,32°C	902,8mBar	Decrease pressure
00:02:07	328,32°C	66mBar	Preprocess ended
00:02:07	328,32°C	66mBar	Increasing Pressure

Figur 5-6: Prosessliste

Som eksempel vises oppdateringen av liste i prosessbildet. Når det skjer et fasebytte i prosesskjøringen gir prosessen en Callback til GUI om dette og en linje i prosesslisten skal skrives.

Når en Callback kommer til et skjermbilde skjer dette via funksjonen

```
// Handle callback
void ProcessView::update().
```

For at Callback skal få tilgang til prosesslisten, kalles delegaten og kontrollen overføres til GUI-tråden.

```
// Delegate to allow workerthreads access to the process-log
delegate void ProcessLogUpdater();
ProcessLogUpdater ^logUpdater;

// Create the delegate for updating the process-log.
logUpdater = gcnew ProcessLogUpdater(this,
                              &ProcessView::updateProcessLog);

// Invoke the processLogUpdater, we cant use asyncronious call here
// because of thread syncronisation
Invoke(logUpdater);
```

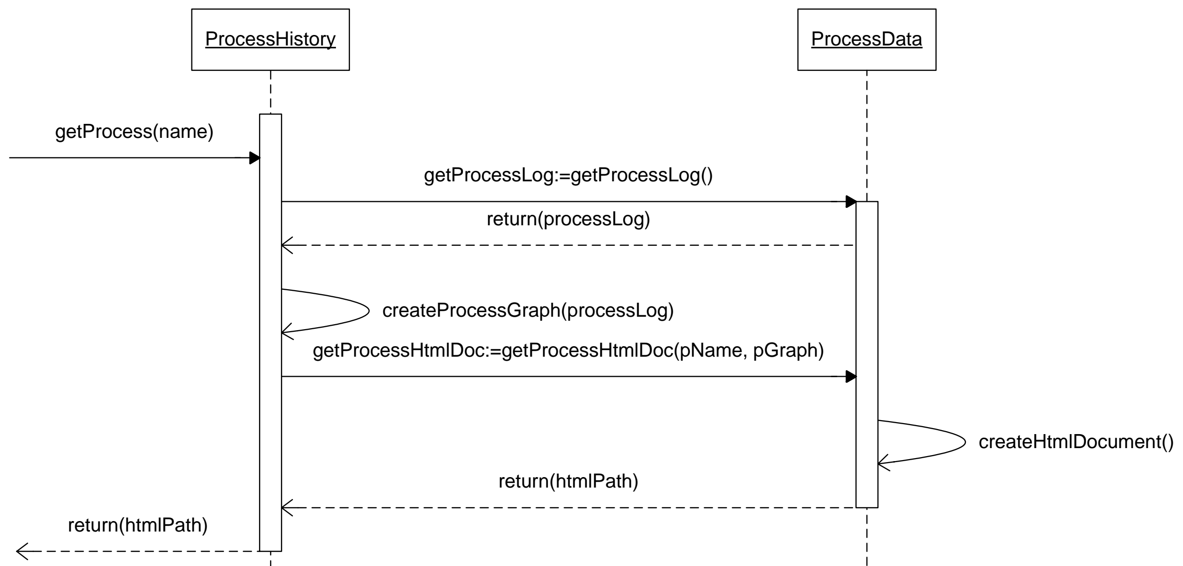
5.6.2 Prosesshistorikk

All prosessdata som genereres under kjøring av prosesser lagres med en detaljeringsgrad som gjør det mulig å regenerere prosessforløpet for rapportering. Denne funksjonaliteten er implementert ved at det benyttes et XML-stylesheet til å generere en HTML-fil for visning. Denne løsningen er fleksibel ved at det kan benyttes forskjellige stylesheets dersom man ønsker forskjellige typer rapporter. Dette er gjort fordi det vil være behov for forskjellig type prosessdokumentasjon for både A4-printere, resept/lable-printere og eksterne systemer.

For å generere HTML-fil starter kallet i ProcessHistory.

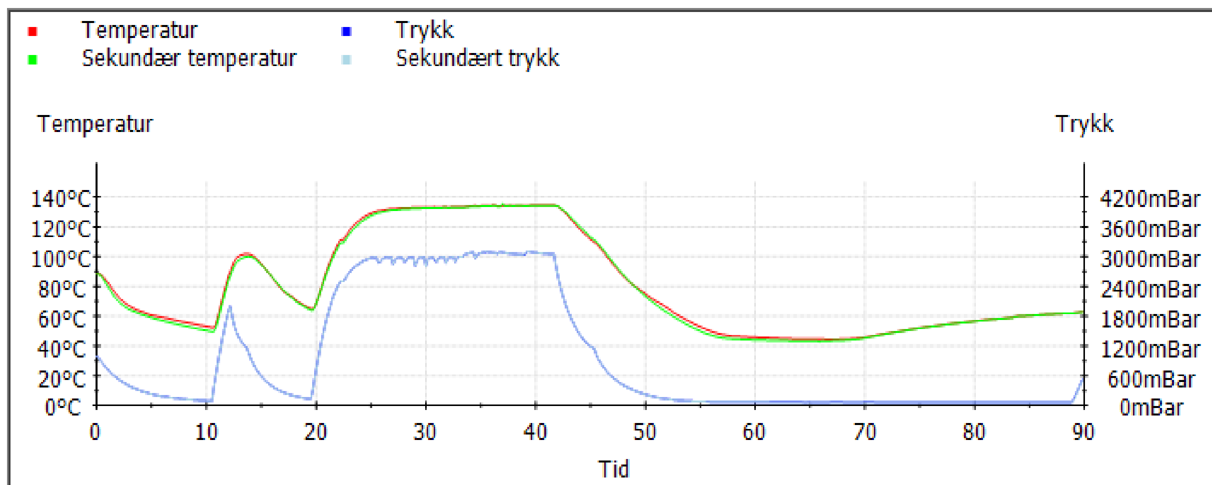
```
// Get process for given process-name
void ProcessHistory::getProcess(String ^name)
```

Deretter gjøres et kall til IO for å hente prosessdata for aktuell prosess.



Figur 5-7: Sekvens ved generering av prosessrapport

Disse dataene benyttes for å generere grafen over prosessforløpet og genereres med tidsintervall som ved reell kjøring av prosessen, dvs. 1 sekund.



Figur 5-8: Prosessgraf

Etter generering av grafen lages et bitmap-objekt av grafikken.

```
// Create rectangle for grapg size-reference
Rectangle rc = Rectangle(0, 0, ahProcessGraph->Width,
                             ahProcessGraph->Height);
// Create bitmap for the graph
Bitmap ^bm = gcnew Bitmap(ahProcessGraph->Width,
                          ahProcessGraph->Height);
// Draw the graphics content of the graph to bitmap
ahProcessGraph->DrawToBitmap(bm, rc);
```

Dette bitmappet sendes videre ned til IO for inkludering i selve HTML-filen.

```
// Creates html-document for viewing an returns the path to the
// document
wProcessViewer->Url = gcnew Uri(pData->getProcessHtmlDoc(name, bm));
```

Videre blir bitmappet lagret til fil for å kunne inkluderes i HTML-dokumentet.

```
// Save graph-image
pGraph->Save(pGraphPath, Imaging::ImageFormat::Png);
```

Stylesheetet lastes og via XSL-transformer genereres HTML-dokumentet og lagres til fil. Lagringen av prosess-bitmap og HTML-dokument er midlertidig og vil overskrives ved neste HTML-generering.

```
// Create the XSL transformer
XslCompiledTransform ^trans = gcnew XslCompiledTransform ();
// Load the stylesheet
trans->Load(xslPath);
//Transform the given xml-file to the given output-file
trans->Transform(xmlPath, xslArgs, htmlWriter);
// Return the path to the generated html-document
return htmlPath;
```

Stien til HTML-dokumentet returneres opp til AH::Pride::GUI som laster dokumentet i en intern web-browser for visning.

5.7 GUI::RAW

Denne pakken inneholder funksjonaliteten for å finne ut hvilken pekeenhet som ble brukt.

5.7.1 Managed og native kode

Som beskrevet i avsnitt 4.3.7 må dette gjøres i native kode og ikke i managed[5], dette fører med seg en del ting som burde forklares nærmere.

Det er behov for å få større mengder data tilbake fra kallene som blir kjørt nede i Windows kjernen, altså ikke kun enkle datatyper. .NET må da fortelles hvordan denne datatypen er representert i minne. Hvordan dette er representert i native kode finnes i winuser.h, en fil som kommer med utviklingsverktøyet og blir brukt under kompilering av native Windows programmer. Et eksempel er RID_DEVICE_INFO som blir definert slik.

```
typedef struct tagRID_DEVICE_INFO {
    DWORD cbSize;
    DWORD dwType;
    union {
        RID_DEVICE_INFO_MOUSE mouse;
        RID_DEVICE_INFO_KEYBOARD keyboard;
        RID_DEVICE_INFO_HID hid;
    };
} RID_DEVICE_INFO, *PRID_DEVICE_INFO, *LPRID_DEVICE_INFO;
```

Dette må skrives om slik at det kan brukes i .NET. Det ble gjort på følgende måte.

```
[StructLayout(LayoutKind::Explicit)]
public ref class RID_DEVICE_INFO {
public:
     [FieldOffset(0)] System::UInt32 cbSize;
     [FieldOffset(4)] System::UInt32 dwType;
     // same place in memory, this is a UNION
     [FieldOffset(8)] RID_DEVICE_INFO_MOUSE mouse;
     [FieldOffset(8)] RID_DEVICE_INFO_KEYBOARD keyboard;
     [FieldOffset(8)] RID_DEVICE_INFO_HID hid;
};
```

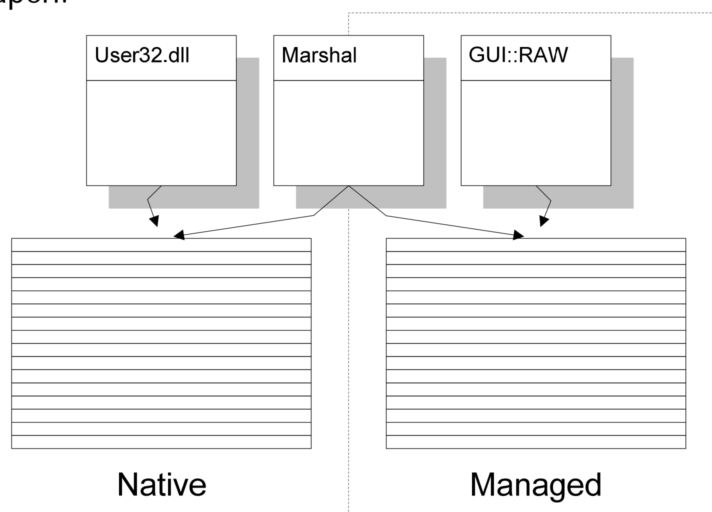
.NET kan representere datastrukturer i minnet slik den finner det best. For at minnelayouten i den managede delen skal være lik som den i native, må layouten være gitt. Dette gjøres ved å sette layouten til Explicit og angi hvilke minneplasser de forskjellige variablene skal befinne seg i. Union er heller ikke brukt i .NET, men dette løses ved å angi at alle datatypene i unionen har samme minneplass.

[5] *Kode som kjøres og kontrolleres av .NET rammeverket.*

For å få tilgang til rutinene i dllene som er med Windows må det angis signaturen og hvor den befinner seg, dette gjøres på følgende måte.

```
[DllImport("user32.dll")]
extern unsigned int GetRawInputData(IntPtr hRawInput,
                                    unsigned int uiCommand,
                                    IntPtr pData,
                                    unsigned int &pcbSize,
                                    unsigned int cbSizeHeader);
```

Det brukes direktivet dllImport for å angi filen etterfulgt av signaturen til metoden som skal brukes. Signaturen kommer fra MSDN[6], med visse forandringer. Alle pekere til data må være av typen IntPtr siden dette er pekeren til minnet på den unmanagede heapen.



Figur 5-9: Datautveksling mellom managed og unmanaged heap

Kopieringen av data mellom den managede og den native heapen er det en klasse i .NET kalt Marshal som tar seg av som vist i Figur 5-9.

[6] *Microsoft Developer Network*

5.7.2 Bruk i løsningen

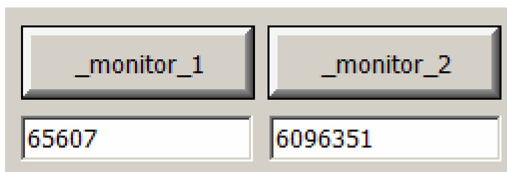
Dette brukes for eksempel da listen over alle pekeenhetene på maskinen blir generert. Nedenfor er et utsintt av metoden som bygger opp listen av pekeenheter.

Nedenfor er et utsnitt av koden. Det blir gjort to kall til GetRawInputDeviceList, det første kallet blir gjort for å finne størrelsen på reslutatet, det andre kallet henter de faktiske verdiene. Det må gjøres på denne måten på grunn av at det må være allokert et minneområde før vi henter verdiene.

```
// first we get the size
GetRawInputDeviceList(IntPtr::Zero, num, 8);
// allocate the memory we want to save in
list = Marshal::AllocHGlobal(8 * num);
// let windows fill it with information
GetRawInputDeviceList(list, num, 8);
// loop through all and add it to our list
for(unsigned int i = 0; i < num; i++){
  RawInputDevice ^device = gcnew RawInputDevice();
  device = (RawInputDevice^)Marshal::PtrToStructure(
                                  *(gcnew IntPtr((Int32)(list.ToInt32()
                                  + i * 8))), device->GetType());
  // is it a mouse?
  if(device->dwType != RIM_TYPEMOUSE) {
    // if not we just skip this one
    continue;
  }
  // add the device to the list of devices and to the list of mice
  devices->Add(device);
  MouseInfo^ mouse = gcnew MouseInfo();
  mouse->hDevice = device->hDevice;
  mice->Add(mouse);
}
// free the memory allocated further up
Marshal::FreeHGlobal(list);
```

Dette blir brukt når en komponent i løsningen tar imot rawinput beskjeder fra Windows. Dette gjøres ved at å implementere WndProc og håndterer WM_INPUT beskjeden. Håndtering går ut på å sende parmeteren med den aktuelle informasjonen til RawMouse klassen som igjen bare sammenligner handelen til den som ble brukt med de vi har i listen og returnerer den som matchet.

```
void ahButton::WndProc(Windows::Forms::Message %m){
  switch(m.Msg){
    case 0x00FF: // <---- WM_INPUT
      lastMI = rm->update(m.LParam);
    break;
  }
  UserControl::WndProc(m);
}
```



Figur 5-10: Identifisering av pekerenheter

Et eksempel på bruk av dette finnes i MConfig som representerer skjermbildet for å konfigurere maskinen. Når en av knappene blir trykket på, registreres det hvilken skjerm som ble brukt i feltet under. Tallet som lages her er handlen til pekeenheten som ble brukt.

```
void MConfig::cmdMonitor1_Click(Object^ sender, System::EventArgs^  e){
  ahButton ^btn = safe_cast<ahButton^>(sender);
  MouseInfo ^mi = btn->getMouseInfo();
  this->tbMonitor1->Text = mi->hDevice.ToString();
}
```

Tips til hvordan dette kunne løses fant vi på en side på nettet der slik funksjonalitet var lag til i et emulatorprogram av gamle konsoller.[7]

[7] *Kilde: http://www.jstookey.com/arcade/rawmouse/*

6. Test

Testing av løsningen er veldig viktig siden løsningen kan brukes i medisinsk sammenheng. I tillegg kan en autoklav være farlig om den blir styrt feil. Derfor er det viktig at løsningen er testet så godt som mulig. Dette kapittelet beskriver hvordan systemet er testet samt hvilke tester som bør bli utført i videre arbeid.

6.1 Uførte tester

Denne seksjonen beskriver de testene som er blitt utført på systemet og hvorfor disse er utført. Dette er for det meste simulerte tester siden en reell maskin ikke har vært tilgjengelig før helt i slutten av prosjektperioden. For testene beskrevet under har heller ikke dette vært kritisk siden et system for simulering har vært tilgjengelig.

Ut i fra teori vet vi at testing kun kan påvise feil i systemet, ikke bevise at systemet er feilfritt. På grunn av dette vil store feil oppdages under testingen, men det kan dukke opp flere feil under daglig drift. Eventuelle softwarefeil vil systemet logge og kunne gi utviklerne en pekepinn på hvor feilen kan befinne seg. Om det er en annen type feil kan mange av disse finnes ut i fra hendelses- eller prosessloggen.

6.1.1 Komponent- og inspeksjonstesting

Under utviklingsprosessen er det naturlig å teste koden samtidig som man skriver den. I dette prosjektet er det helt fra starten vært mulig å kompilere og kjøre programmet. Dette har ført til at all kode som er skrevet har blitt kjørt og sjekket om den oppfører seg slik det er ønsket, helt fra starten av.

Den viktigste formen for testing vi har benyttet tidlig i utviklingsfasen er inspeksjonstesting. Dette har vi brukt i deler hvor logikken er komplisert og kritisk for systemet. Måten dette er gjort på er at den som har skrevet koden får med seg et annet gruppemedlem og forklarer hva koden skal gjøre. Så er det opp til det andre medlemmet å kontrollere at koden som er skrevet faktisk gjør det den skal og at det ikke mangler noe. Dette har vært den mest effektive måten for feilretting og har avdekket mange logiske feil som ellers ville vært vanskelig å finne.

Alle GUI komponenter som er egenutviklet er testet i mindre løsninger før de har blitt flyttet inn i hovedløsningen. Her er det testet de individuelle metodene og funksjonaliteten til komponentene. Dette har ført til at de fleste feil i komponenten er fjernet slik at overgangen inn i løsningen har gått veldig bra.

6.1.2 Whitebox testing

I kritiske deler av koden er det utført enda grundigere tester. Dette er deler som styrer komponentene og prosessen i autoklaven. Det ble utført såkalt whitebox testing der logikken i koden er gjennomgått med forskjellige verdier på variable og verifisert mot slik det burde oppføre seg.
Dette ble for eksempel gjort under utvikling av styrelogikken til kjelen. Det ble brukt en debugger for å steppe gjennom kodelinjene mens en manuelt styrt sensor var

koblet til busskobleren. Denne testingen har ført til at logikken i løsningen har blitt veldig godt gjennomgått og det burde ikke finnes store logiske feil i disse delene av løsningen.

Prosjektgruppa har hatt god oversikt over hva som er kritisk og det har vært naturlig å kjøre denne typen testing underveis i utviklingen. Det har derfor ikke vært behov for en mer formell whitebox testing av hele systemet.

6.1.3 Blackbox testing

Det ble også utført blackbox testing av de kritiske delene av systemet. Her ble output observert ut i fra manuelt styrte sensorer og sammenlignet med forventet resultat. Denne typen testing ble det gjort mye av i koden som styrer prosessen.

I det reelle miljøet ble det utført blackbox testing og ikke whitebox testing. Whitebox testingen ble ikke brukt på grunn av at stopping av programmets gang her kunne ført til at systemet mistet kontrollen over autoklaven.
Denne testingen førte til at det ble avdekket feil i hvordan løsningen oppførte seg, altså funksjonelle feil. Denne er testingen er utført av gruppedeltagere. Det har blitt gjort uformelt uten egenutviklede testcase. Programmet ble startet opp og vi fulgte med på hvordan autoklaven oppførte seg. Vi var så godt kjent med prosessen og styrelogikken at de fleste feilene som oppstod ble identifisert med en gang. Feilene og manglene som oppsto ble notert ned. Så vi løsningen på problemene der og da ble feilene rettet og programmet ble kjørt på nytt. Feil som vi ikke direkte så grunnen til ble det jobbet videre med på skolen. Siden hovedprioriteten i denne korte fasen var å få maskinen opp til å kjøre en fullstendig prosess ble dokumentering av feilene nedprioritert.

Et eksempel på en feil vi fant i det reelle miljøet var at settpunktene en oppvarmingsfase skulle kjøre mot var byttet om. Når trykket skulle senkes ble det sjekket mot maksimumsverdiene og når trykket skulle økes ble det sjekket mot minimumsverdiene. Denne feilen så vi grunnen til så å si umiddelbart og den ble rettet på stedet.

6.2 Fremtidige tester

Denne seksjonen beskriver hvilke tester som må utføres før løsningen blir satt i drift. Whitebox og Blackbox testingen kan ikke anses som ferdig, og fremtidig testing er meget viktig for dette systemet. Grunnet faste tidsfrister og stadig utsettelser av å få satt opp et reelt testmiljø har det blitt meget lite tid til testing av systemet i reell drift. Den testingen som er utført har påvist flere feil i systemet og dette i seg selv vitner om at testprosessen er langt fra ferdig.

Det må utvikles formelle testcase som beskriver hvordan systemet skal testes og testene må dokumenteres.

6.2.1 Brukertesting

I denne testingen bør det primært benyttes personer som faktisk skal bruke systemet i daglig drift. Tilbakemeldinger fra disse personene brukes til å styre eventuelle forandringer i brukergrensesnittet. Fra oppdragsgiver ble det gitt klare føringer for hvordan brukergrensesnittet skulle se ut. Dette gjaldt blant annet en utvikling av en ny type knapp som skulle gi en bedre 3D-følelse av at brukeren faktisk trykker inn knappen. Dette må testes av brukerne for å bekrefte at knappens effekt.
Designet av brukergrensesnittet i sin helhet må også testes slik at dette blir så lett å bruke som mulig. Det kan være at brukerne ønsker å flytte om på rekkefølgene i menyen ut i fra de menyvalg som brukes mest. Hvis de må flytte hele hånden til en annen del av skjermen for å navigere mellom menyene kan dette være et irritasjonsmoment.

6.2.2 Stresstesting

For å kunne teste kravene om stabilitet må det kjøres stresstester. Dette kan for eksempel foregå ved at tiden mellom oppdateringene fra multiplekseren blir senket slik at vi får mange flere oppdateringer og dermed større last på systemet. I tillegg til å teste stabiliteten kan det ved denne testen provosere frem feil under synkronisering av tråder siden mange tråder vil ha arbeid å utføre samtidig.

Det burde også kjøres mange prosesser etter hverandre for å se om systemet er så stabilt det burde være. Denne prosedyren burde bli automatisert slik at det ikke er nødvendig å ha folk tilstede under testingen siden et program kan ta opptil 90 minutter. Det er ikke noe funksjonalitet for dette i løsningen så forandringer i koden er nødvendig for å kjøre en slik test. Det er imidlertid viktig å kjøre en slik test for å avdekke eventuelle feilsituasjoner som kan oppstå under lengre drift.

Testing av oppetid vil gjøres ved å la systemet kjøre normalt over en lengre periode for å se om det da oppstår noen feil.

6.2.3 Akseptanse- og systemtest

Til slutt burde alle funksjoner i systemet testes i sin helhet for å se om de utfører det som kravene sier de skal gjøre. Dette burde gjøres i samarbeid med oppdragsgiver for å se om systemet er blitt det han ønsker seg. Testing av funksjonalitet burde bli gjort som formelle blackbox tester der det blir beskrevet hva som skal testes og hva godkjente utfall vil være for så å se om dette faktisk stemmer overens.

7. Avslutning

7.1 Evaluering av oppgaven

Under utviklingen mener vi at vi har laget et godt design. Når vi ser tilbake er det imidlertid noen løsninger som vi ser at ikke er optimalt løst. Et eksempel på dette er måten konfigurasjonsinnstillinger blir lest inn og ut. Vi burde her hatt en klasse i domene eller konfigurasjonspakken som holdt på minneresident konfigurasjonsinnstillinger slik at disse kunne blitt benyttet dynamisk mens programmet kjører. Dette ville også ført til at konfigurasjonspakken ikke hadde fått så mange koblinger til andre klasser i løsningen.

En annen designmessig 'glipp' vi har gjort er at Presentasjonslaget har koblinger til Tjenestelaget, dvs. at det hopper over domenelaget. Dette er i strid med lagmodellen vi har valgt, og er antagelig et resultat av en hektisk programmeringsfase der design av enkeltdeler ikke var like gjennomtenkt.

I ettertid ser vi at testingen kunne blitt gjort med mer formalitet der vi for eksempel hadde utarbeidet forventet utfall av blackbox tester før vi utførte dem. Uten det vil forventet resultat lett påvirket av det vi ser under selve testen.

I forprosjektrapporten ble det beskrevet at gruppen skulle gi en anbefaling rundt hvilken hardware som systemet burde kjøre på. Noe slik notat er ikke utviklet siden oppdragsgiver har selv vært veldig aktiv under valg av hardware. Vi har kommet med generelle anbefalinger og kommentarer på hans forslag.

Det har underveis i prosessen kommet til ønsker om mer funksjonalitet en det i utgangspunktet var tenkt. Dette har kommet i forbindelse med at oppdragsgiver ser nye muligheter ved å bruke en datamaskin til å styre autoklaven. De viktigste av disse er muligheten for å styre komponentene i autoklaven manuelt via et maskindiagram samt å kunne skifte mellom de forskjellige bildene mens prosessen kjøres. Disse kravene er beskrevet i kravspesifikasjonen og innfridd underveis.

7.2 Gruppearbeidet

Gruppearbeidet har fungert veldig godt, vi har alle jobbet like mye med løsningen. Som beskrevet tidligere har gruppa jobbet sammen tidligere og det har også da fungert uten større problemer. Vi føler det har vært viktig for samarbeidet at vi kjenner hverandre godt og hverandres arbeidsmetoder.

Det var tidlig diskusjoner der forskjellige gruppemedlemmene hadde forskjellig oppfattning av hvordan ting burde løses. Dette kunne fort ha ført til konflikter, men i stedet utviklet det seg til å bli positivt ved at vi så problemet fra flere sider. Dette har ført til at vi har fått et meget godt design.

Under utviklingen av koden har vi forsøkt å jobbe på forskjellige deler, dette har fungert meget godt. Siden vi kun var tre personer foregikk mye av kommunikasjonen muntlig under utviklingen. Om det var behov for funksjonalitet i et sted i koden en av

de andre jobbet på, ble vi bare enig om hvordan dette skulle bli lagt til. Dette hadde nok ikke fungert like godt med en større gruppe.

Kommunikasjon med oppdragsgiver har stort sett foregått over telefon eller ved personlig oppmøte. Dette har vært korte spørsmål og gir ikke noe grunnlag for møtereferater. Tidlig i prosjektet hadde vi mer formelle møter der det ble generert møtereferater, eksempel på dette ligger vedlagt i Vedlegg F.

Det har i dette prosjektet som i de fleste andre blitt lengre dager mot slutten, men vi har helt fra prosjektets start brukt mye tid. Timeforbruket og et eksempel på loggingen vår ligger vedlagt i Vedlegg E.

7.3 Videre arbeid

Vi har realisert de fleste UseCasene beskrevet i kravspesifikasjonen bortsett fra Administrere driftstider og Hente prosessdata. I tillegg mangler det deler av tørkeprosessen samt en del internasjonalisering. Disse delene er ikke med på grunn av for liten tid. De ble nedprioritert i samarbeid med oppdragsgiver for å kunne fullføre andre og viktigere krav. For utskrift av prosessdata gjenstår en del arbeid med tilpassning til forskjellige typer printere. Løsningen som den er nå er kun beregnet for utskrift på A4-printer, og det vil i tillegg til dette være behov for utskrift til labelprinter. Disse delene må derfor ferdigstilles før løsningen kan kalles ferdig.

Som del av systemets ferdigstilling må det utføres mye mer testing. Se avsnitt 6.2 for detaljer.

Oppdragsgivers langsiktige visjon er å la autoklaven være en del av et sporingssystem for alt utstyr som blir brukt på sykehus. Integrering med dette sporingssystemet vil være et steg videre for produktet.

7.4 Konklusjon

Gjennom prosjektets gang har vi 'endelig' fått gjennomført et virkelig prosjekt slik vi har lært at det skal gjøres fra tidligere 'tenkte' utviklingsprosjekter. Det har vært meget lærerikt å kunne starte med krav gitt av en ekstern oppdragsgiver for så å konkretisere disse til et utgangspunkt for videre design og utvikling av en reell programvareløsning.

Vi dratt nytte av teknikker og metoder vi har lært i tidligere fag, ikke bare innen systemutvikling og programmering, men også andre fag som operativsystemer, økonomi, matematikk og fysikk.

Arbeidet med selve prosjektet gitt oss verdifulle erfaringer med tanke på kommunikasjon innad i gruppa i tillegg til samarbeidet med oppdragsgiver og veileder.
Oppgavens art har også gitt spennende utfordringer som har vært uvante for oss. En autoklav er en fysisk 'maskin' som bråker og rister, og det å utvikle programvare for denne 'maskinen' har vært spesielt og ikke minst meget lærerikt.
Vi føler at arbeidet vi har gjort har resultert i et godt produkt. Prosjektet har vist at det ikke er nødvendig med tradisjonelle systemer for prosesstyring. Systemet har dessverre ikke blitt testet nok til at vi kan påstå at det er like stabilt som et PLS-system, men prosessen blir kjørt på korrekt måte selv uten at det er implementert i et realtime system.
Vi konkluderer med at det å arbeide strukturert og målbevisst gir et godt og kvalitetssterkt sluttprodukt. Utgangspunkt i teori og erfaringer har vært et uvurderlig verktøy som har hjulpet oss med å ta de riktige beslutningene underveis.

Vedlegg A. Autoklavens oppbygning

Grunnet innholdets konfidensielle art, er dette vedlegget ikke med i denne offentlige utgaven av rapporten.

Vedlegg B. Autoklavens Funksjon

Grunnet innholdets konfidensielle art, er dette vedlegget ikke med i denne offentlige utgaven av rapporten.

Vedlegg C. Hardware grensesnitt

Grunnet innholdets konfidensielle art, er dette vedlegget ikke med i denne offentlige utgaven av rapporten.

Vedlegg D. Detaljerte funksjonelle krav

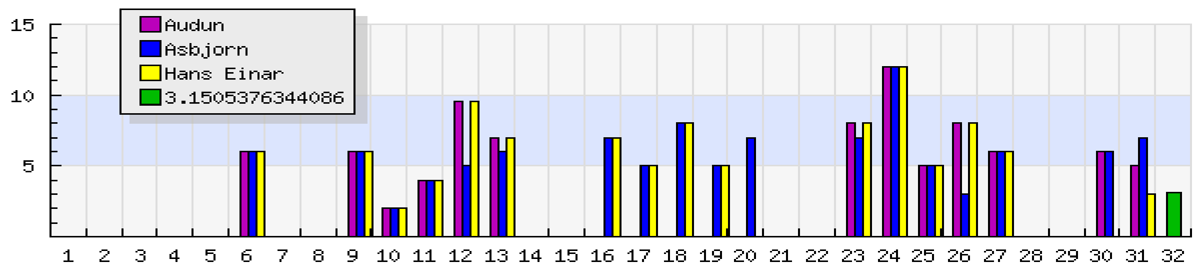
Grunnet innholdets konfidensielle art, er dette vedlegget ikke med i denne offentlige utgaven av rapporten.

Vedlegg E. Timelogg

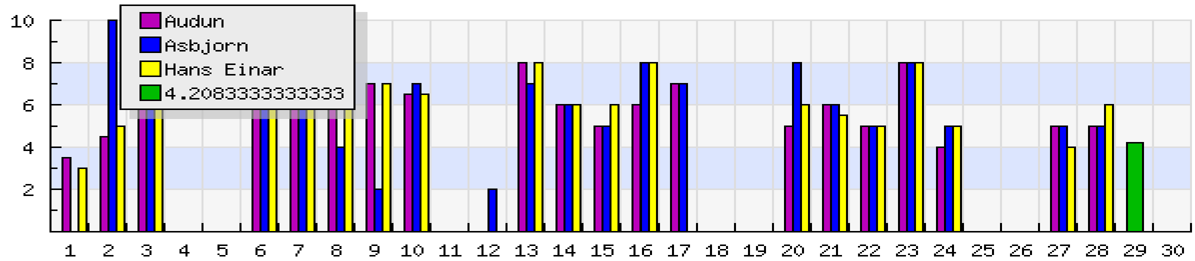
Her blir grafer over arbeidsmengden vår vist, samt eksempel på hvordan timeloggen er ført. For en mer utfyllende beskrivelse av hva som er blitt utført de forskjellige dagene se prosjektets hjemmeside, http://www.hig.no/imt/index.php?id=736. I mai mangler de siste dagene på grunn av prosjektrapporten skulle ferdigstilles.

February 23, 2006			
	Asbjorn	Timer: 8	Minutter: 0
	Jobbet med ahProcessGraph. Noe diskusjoner rundt Config-pakke og format på xml-filer for programdefinisjoner. Startet med tegning av symboler for maskindiagram.		
	Hans Einar	Timer: 8	Minutter: 0
	Leste om xml og så på hvordan vi skulle bruke det i systemet vårt		
	Audun	Timer: 8	Minutter: 0
	Jobbet mer med config, lagt alt som skal settes av config i vakuumtest som private variable og gjort klart for at disse hentes fra config		
Totalt 24 timer og 0 minutter.			

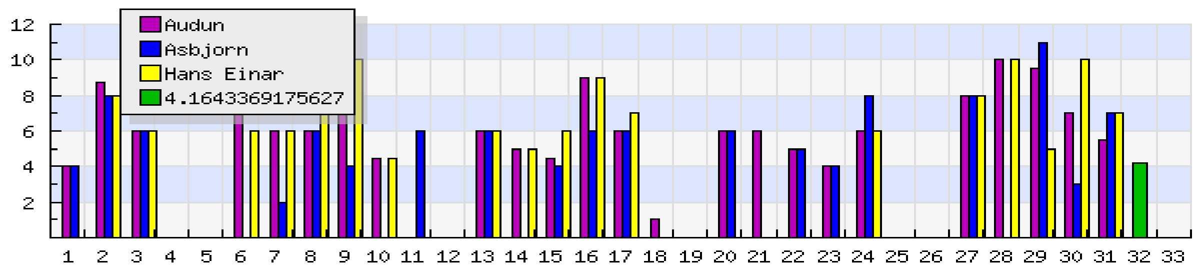
February 20, 2006			
	Asbjorn	Timer: 8	Minutter: 0
	Møte med Bård. Startet med koding av custom controller for systemet. ahButton og ahProcessGraph.		
	Hans Einar	Timer: 6	Minutter: 0
	EUREKA!!! løste "problemet" med adressering av bussen..		
	Audun	Timer: 5	Minutter: 0
	Implementerte events i stedet for Observer Observable		
Totalt 19 timer og 0 minutter.			



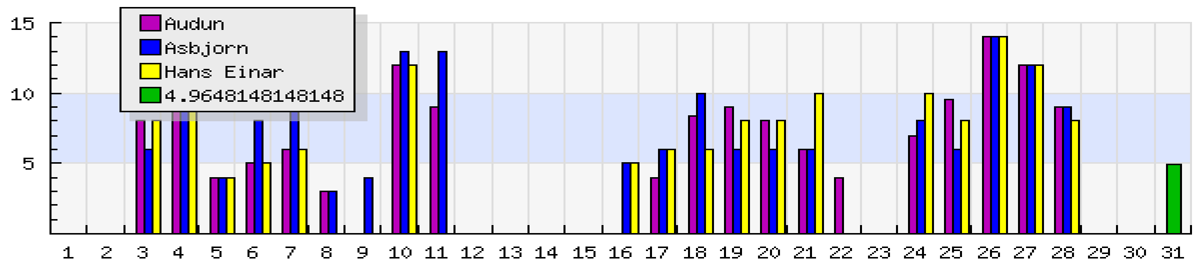
Figur E-1: Januar



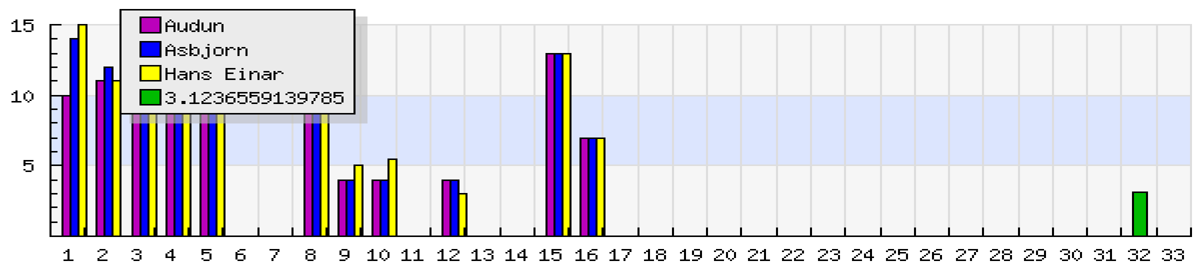
Figur E-2: Februar



Figur E-3: Mars



Figur E-4: April



Figur E-5: Mai

Vedlegg F. Møtereferat (eksempel)

Møtereferat

Type møte:	Prosjektmøte	Dato:	12.01.2006
Referent:	Audun Klundby	Klokkeslett:	12:00
		Sted:	AH Partner AS, Gjøvik

Deltagere:
Bård Marken, AH Partner AS
Hans Einar Øverjordet, HiG
Audun Klundby, HiG
Asbjørn Konstad, HiG

Innhold:

Vi startet med å snakke om adressering av kontrollerenheten vi skal bruke under prosjektet. Vi ble enige om å ta et møte med PhoenixContact Norge like etter at forprosjektrapport er levert, altså etter 26.01.

Vi snakket også om hva skolen forventet av dokumenter fra oss og hva vi ville utvikle av dokumenter. Vi snakket om å starte utvikling av kravspesifikasjon parallelt med forprosjektrapport. Dette ble ikke vedtatt.

Kontrakten som skolen tilbyr oss ser grei ut i følge Bård, men han ville ikke at kildekoden og funksjonsbeskrivelsen skulle legges ut for alle gjennom biblioteket. Det ble nevnt at vi muligens skulle legge sensitiv informasjon som vedlegg. Vi ble enige om å ta kontakt med Tom Røise om dette. I tillegg lurte Bård på hva slags krav det var til sensoren han måtte stille med, dette skal vi også ta opp med Tom Røise.

Etter dette snakket vi om fremdriftsplanen, vi ble enige om aktiviteter, men fikk ikke satt fast noen milepæler.

Vi fikk et eksempel på en rapport systemet kunne generere.

Vi fikk også en liten gjennomgang av prosessen.

Vi snakket også om utstyret og ble enige om hva vi trengte av funksjonalitet på maskinen, vi skulle få sjekket ut hva som fantes og gjøre et valg rundt hardware så fort som mulig.

Vedlegg G. Skjermbilder

Grunnet innholdets konfidensielle art, er dette vedlegget ikke med i denne offentlige utgaven av rapporten.

Vedlegg H. Forprosjekt (uten vedlegg)

Forprosjektrapport

Styring og betjeningsenhet for Autoklave

Hovedprosjekt 2006

Audun Klundby
Asbjørn Konstad
Hans Einar Øverjordet

H.1 Mål og rammer

H.1.1 Innledning

Autoklaver er maskiner benyttet i medisinsk sammenheng for å sterilisere utstyr. Alle bakterier, virus, mikrober og sporer destrueres ved at en kombinasjon av temperatur og trykk opprettholdes over tid. Autoklaver benyttes på sykehus, laboratorier og andre institusjoner med behov for sterilt utstyr.

A&H Partner er en Gjøvik-bedrift, som blant annet leverer autoklaver, laboratoriemateriell og måleutstyr til laboratorier og helseinstitusjoner. Til tross for samarbeid med de virkelig store aktører innen steriliseringsutstyr, ser A&H Partner at det eksisterer et stort potensial for utvidelse av funksjonalitet, og forbedring av resultat på de produkter som eksisterer. A&H Partner har derfor startet utviklingen av ny og revolusjonerende teknologi i forbindelse med konstruksjon, operasjon og styring av autoklaver. Denne autoklaven skal styres fullstendig via datamaskin med direkte fokus på stabilitet og sanntidsdata.

H.1.2 Effektmål

Effektmålet for prosjektet er å komplimentere autoklaven med et pc-basert styringssystem som skal innfri krav og normer gitt for prosessforløpet ved sterilisering av medisinsk utstyr. Dette vil bidra til å styrke oppdragsgivers markedsposisjon gjennom et robust og kvalitetssterkt produkt.

H.1.3 Resultatmål

Gruppens mål er å etablere en stabil plattform for systemet samt å utvikle komplett og feilfri programvare for pc-basert styring av autoklaven med fokus på stabilitet og brukervennlighet. Systemet skal utvikles for direkte konfigurering, styring og overvåking av autoklaven. Alle prosesser skal dokumenteres og systemet må kunne produsere detaljert informasjon til printere så vel som andre eksterne systemer. Løsningen skal være skalerbar med tanke på fremtidige muligheter til å variere valg av hardware samt mulighet for gjenbruk i andre maskinkonfigurasjoner.

H.1.4 Læringsmål

Det legges vekt på å benytte tilegnet lærdom fra aktuell studieretning for gjennomføre utviklingsprosessen på en best egnet måte. Dette vil sikre en robust og skalerbar løsning som vil innfri kravet om stabilitet. Det legges spesielt vekt på å etablere en løsning med klare og optimale laginndelinger som muliggjør en fleksibel/skalerbar konfigurasjon.

H.1.5 Rammer

Tidsfrist for prosjektet er satt av instituttet til 24 mai 2006. Det stilles krav til gjennomføring og dokumentasjon av prosjektet samt frekvens for statusmøter. Forøvrig etterkommes oppdragsgivers tidsfrister for delleveranser samt eventuelle økonomiske rammer.

Det vil stilles krav til at I/O-hardware for styring av autoklaven er av typen Phoenix Ethernet/Inline Bus Terminal. Dette er en I/O-kontroller som integreres via Ethernet-

grensesnitt. Gruppa vil allikevel jobbe mot en modell som muliggjør alternative I/O-systemer med relativt enkle tilpassninger.

H.2 Omfang

H.2.1 Oppgavebeskrivelse

Oppgaver for gruppen blir å etablere og anbefale en mest mulig stabil plattform som systemet skal kjøre på. Det vil her bli lagt stor vekt på driftsikkerhet i en integrert løsning. I tillegg vil gruppen være ansvarlig for å etablere en komplett programvareløsning med kommunikasjon mot tredjeparts I/O-moduler. Dette er relativ ny teknologi og det vil i starten benyttes betadrivere for dette utstyret. Gruppens mål er å lage programvaren fleksibel med tanke på mulige fremtidige variasjoner i maskinvarekonfigurasjon ved å seksjonere løsningene med standardiserte grensesnitt og utskiftbare moduler.

Systemet skal designes for å benytte trykkfølsomme skjermer. Disse skjermene kan være av begrenset størrelse og det vil av den grunn være viktig med en god oppbygging av GUI. Det er også viktig at systemet blir enkelt og intuitivt å bruke siden brukerne ikke nødvendigvis har hatt opplæring. I tillegg skal det lages en løsning for bruk av on-screen tastatur som erstatter eksternt tilkoblet tastatur.

Det skal etableres et grensesnitt mot systemet slik at administrative systemer enkelt skal kunne få tilgang på data for de individuelle steriliseringsprosessene.

Systemet skal lages for muligheten til å være flerspråklig. Dette skal enkelt kunne konfigureres og endres etter behov fra grensesnitt i applikasjonen.

Oppsett for forskjellige programmer/prosesser skal kunne opprettes og konfigureres enkelt via brukergrensesnittet i systemet.

Autoklaven vil leveres med noe varierende maskinkonfigurasjon. Dette gjør at systemet må lages for å håndtere disse variasjonene. Det vil blant annet ved todørs konfigurasjon være behov for to skjermer og to inputenheter, og prosjektet er obs. på at dette muligens vil kreve noe ekstra fordypning og testing.

En grovinndeling av løsningen vil være:

- Modul for I/O-kommunikasjon med autoklaven.
- Modul for prosessforløp.
- Modul for GUI.
- Modul for tilgjengeliggjøring av data for eksterne systemer.
- Modul for håndtering av datalagring.

H.2.2 Avgrensninger
Prosjektet har som hovedprioritet å utvikle selve styringssystemet for autoklaven. Utover dette vil det være behov for brukerdokumentasjon og demofunksjonalitet for markedsføringsmessige formål. Grunnet tidsmessige forhold er det ikke fastsatt om denne funksjonaliteten skal lages, men gruppen vil være åpen for dette dersom det blir tid. Beslutning om dette vil kunne tas når prosjektet nærmer seg ferdigstilling.

For grensesnitt mot eksisterende administrative systemer vil det kun tilbys et grensesnitt der prosessdata tilgjengeliggjøres.

H.3 Prosjektorganisering

H.3.1 Ansvarsforhold
Prosjektleder er Hans Einar Øverjordet. Ansvarlig for koordinering og sammensetting av dokumentasjon er Asbjørn Konstad. Ansvarlig for webområdet og Subversion repostory er Audun Klundby.

Videre ansvarsfordeling innad i gruppa vil bli tildelt fortløpende ettersom utviklingsarbeidet starter. Vi ser for oss at det vil bli tildelt en hovedansvarlig for hver modul/funksjon. Denne personen vil igjen være ansvarlig for å ferdigstille modulen/funksjonen og må selv må sørge for å benytte de andre deltagerne etter behov. Denne ansvarsfordelingen vil dokumenteres i endelig prosjektplan.

H.3.2 Rutiner og regler i gruppa
- Det settes som minimumskrav at alle i gruppa skal nedlegge et arbeid på 30 timer i uka på prosjektet. Dette tilsvarer en samlet arbeidskapasitet på ca 270 timer i mnd. Arbeid utover dette må påberegnes og avtales innad i gruppa etter behov.
- Alle skal føre en logg over arbeidet de har gjort dag for dag og timeantallet de har brukt.
- Uenigheter om løsninger skal i første omgang diskuteres og hvis det fortsatt er uenighet vil det bli avgjort med avstemming.
- Ved fravær skal det så snart som mulig gis beskjed til de andre gruppemedlemmer. Fravær skal begrunnes.
- Arbeidet vil hovedsakelig foregå i grupperom A030, men det vil også i perioder være behov for arbeid hos oppdragsgiver i forbindelse med testing.
- All prosjektdokumentasjon skal lages utfra definerte maler. Disse er tilgjengelige for alle i gruppa på prosjektets hjemmeside.

H.3.3 Øvrige roller og bemanning
Oppdragsgiver er AH Partner AS v Bård Marken. Veileder ved instituttet er Tom Røise. I tillegg vil prosjektet benytte seg av teknisk personell hos leverandør av I/O-kontroller i Norge og Tyskland.

H.4 Planlegging, oppfølging og rapportering

H.4.1 Hovedinndeling av prosjektet

Førende for prosjektet er at autoklaven også er under utvikling. Selv om det ligger klare krav for hvordan systemet skal fungere, føler gruppa et behov for en fleksibel modell der vi kan være åpne for endringer underveis samt muligheter for iterativ jobbing. Det vil i tillegg være et stort behov for tidlige prototyper for å kunne starte testing og tilpassning av maskinvaren i autoklaven.

Siden systemet skal brukes i medisinsk sammenheng, er det viktig at den er pålitelig og stabil. Dette sender oss i retning av en modell som har en del tyngde på dokumentasjon og testing. RUP har nettopp dette, men vi ser heller nytten av å bruke maler fra RUP i stedet for hele rammeverket. En ren evolusjonær utviklingsmodell vil være uheldig i vår sammenheng da vi ser en risiko for denne tilnærmingen ikke vil gi oss den strukturen og kontrollen som gir oss en stabil programvare.

Vi vil først prøve å kjøre en så fullstendig kravspesifikasjon som mulig. Etter dette vil vi gjøre en overordnet designfase der vi deler inn i moduler og bygger opp et foreløpig klassediagram. Vi vil så implementere og teste modul for modul ut i fra en prioritert liste som utarbeides i samarbeid med oppdragsgiver. Vi ser for oss små endringer i kravspesifikasjon og noe flere endringer i design etter hvert. Systemet vil deles opp i initielle inkrementer der prototyping vil være gjeldende for alle viktige systemfunksjoner i de første inkrementene. Innen de forskjellige inkrementene vil tilnærming variere, men vi ser for oss en del iterativ jobbing for å produsere ønsket resultat. Videre inkrementer vil fokusere på forbedring av prototypene og ny funksjonalitet vil legges til.

Vi har derfor valgt å benytte en tilpasset inkrementell modell. Systemet lar seg greit dele opp i moduler for hvert inkrement, og vi har i tillegg behov itererrasjon i hvert inkrement underveis. Hvert inkrement vill ikke produsere en ferdig modul, men generere prototyper først, som ved senere inkrementer vil forbedres.

Ved å velge denne modellen får vi fordelene med god planlegging i tillegg til fleksibilitet rundt endringer underveis. Vi kan også ved å velge denne tilnærmingen få støtte for kravet om en begrenset tidlig versjon. Følgende tidlige prototyper vil bli prioritert parallelt med kravspesifikasjon og design:

- Grafisk brukergrensesnitt.
- Kommunikasjonsmodul mot I/O-enhetene.
- Funksjonalitet for vakuumtest.
- Funksjonalitet for steriliseringsprosess og B&D Test.

Disse inkrementene vil være prototyper og gjenstand for forbedringer senere i etterfølgende inkrementer der vi vil forbedre og legge til ny funksjonalitet.
Videre inkrementer vil være:

- Modul for generell konfigurasjon av systemet.
- Tjenesteleverandør for konfigurasjons –og prosessdata.
- Grensesnitt mot administrative systemer.
- Dokumentasjonsmodul for produksjon av prosessdata.

- **Plan for statusmøter**

Det er satt opp plan for ukentlige møter med veileder hver tirsdag kl 11:00.
Formelle statusmøter med rapportering planlegges ikke på dette tidspunkt, men vil bli gjennomført i forbindelse med større milepæler.

Prosjektmøte med oppdragsgiver planlegges ikke fast, men vil være etter behov.

Etter første utkast av kravspesifikasjon og design, vil gruppa avholde interne møter mandag annenhver uke der kravspesifikasjon og design vil bli løpende revidert.

H.5 Organisering av kvalitetssikring

H.5.1 Utviklingsmiljø

Systemet skal utvikles med Microsoft .NET rammeverk. Dette er valgt på grunn av at denne plattformen er en førende standard i målmarkedet for produktet. Dette forenkler utviklingen og tilpassingen av grensesnitt mot eventuelle administrative systemer og gir oss noe fleksibilitet med tanke på type språk. I tillegg er motivasjonen i gruppa høy for å benytte denne teknologien for læringsmessige formål.

H.5.2 Standarder og kildekode

All kode som blir skrevet skal dokumenteres av den som skriver det på engelsk.
Alle dokumenter skal leses av alle på gruppa for å avdekke feil eller mangler.
Det skal bli brukt et konfigurasjonsstyringsverktøy for administrering og versjonshåndtering av kildekode.
Standard for all kildekode generert i prosjektet er beskrevet i vedlegg 1.

H.5.3 Dokumentasjon

I dokumentasjonen kommer det til å være en del sensitiv informasjon som ikke skal være tilgjengelig for allmennheten, dette blir skrevet som vedlegg og dermed ikke publisert i biblioteket. Dette gjør at vi legger mye av det som ellers ville passet i rapporten som vedlegg.

H.5.4 Konfigurasjonsstyring

Prosjektgruppa vil benytte Subversion som konfigurasjonsstyringsverktøy for kildekode. Server og repository leveres og driftes av IT-Tjenesten ved HIG.

For prosjektdokumentasjonen vil en mer manuell og menneskelig konfigurasjonsstyring benyttes. Det er dokumentansvarlig sitt ansvar å sørge for å sette sammen dokumentasjon produsert av gruppedeltagerne og at gjeldende versjoner for prosjektdokumentasjonen alltid er tilgjengelig på prosjektets hjemmeside.

- **Risikoanalyse**

Følgende risikoer er vurdert så langt i prosjektet:

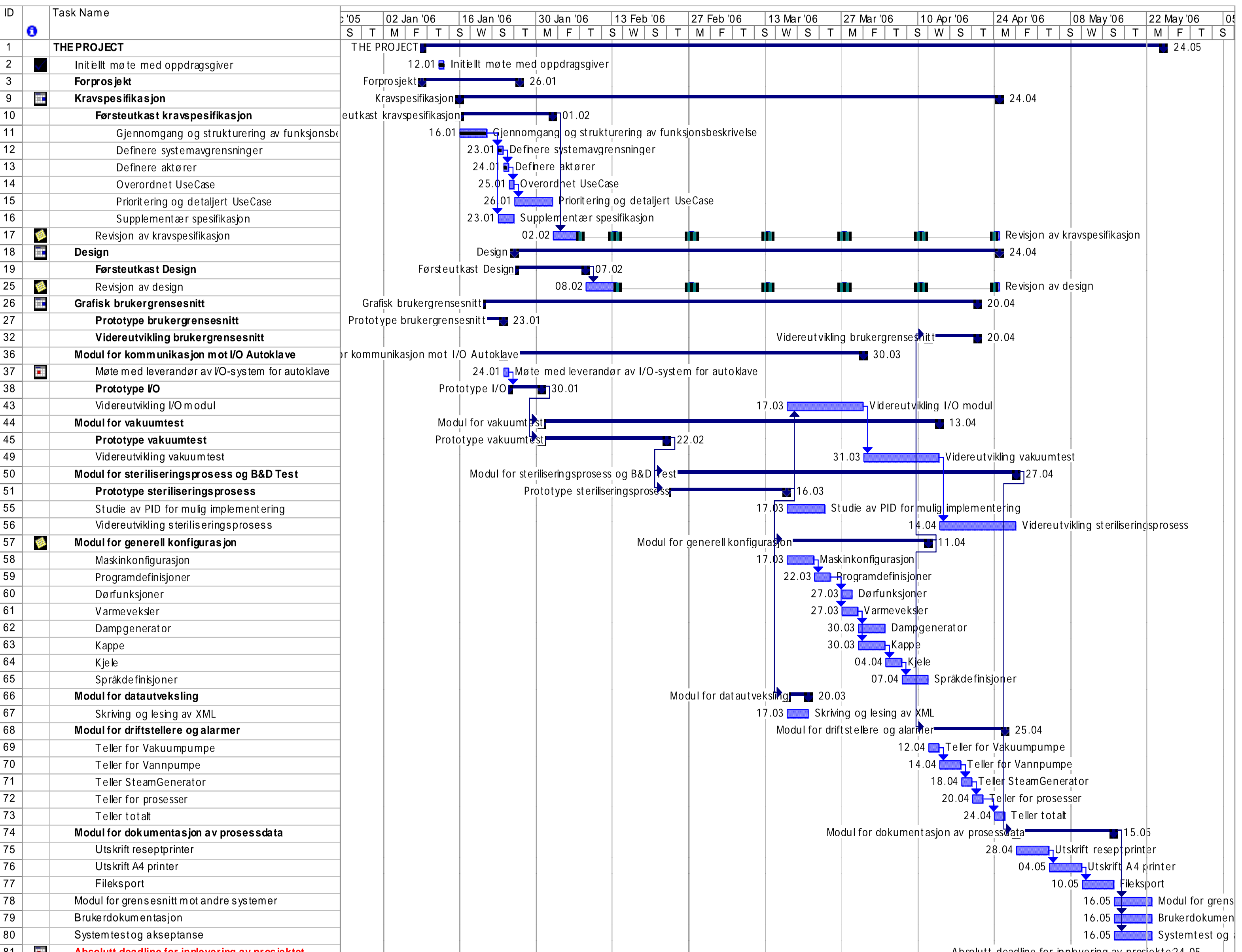
1. **Autoklave blir ikke ferdigstilt til avtalt tidspunkt.**
   Risiko blir vurdert som lav, og tiltak vil være løpende og åpen dialog med oppdragsgiver. Alternativ er å ferdigstille programvaren med en simulert autoklave.

2. **Driverfeil eller driveroppdatering med omfattende endringer.**
   Risiko er vurdert som lav, og tiltak vil være løpende kontakt med leverandør for planer status rundt driveren. Alternativ vil være å benytte betadriver 'as-is' eller vurdere annen løsning runde I/O-hardware.

H.6 Plan for gjennomføring

Plan for gjennomføring av prosjektet er skissert i vedlegg 2 med en grovinndeling av inkrementene. Funksjonsbeskrivelse gitt av oppdragsgiver gir oss relativt klare føringer og innspill for kravspesifikasjon. Det vil imidlertid være noe usikkerhet rundt hvor lang tid vi trenger på hvert inkrement grunnet forskjellig kompleksitet i hver av modulene. Av den grunn ønsker vi ikke på dette tidspunktet og detaljplanlegge hele utviklingsløpet, men heller sette noen få milepæler vi kan forholde oss til. Dette vil gi oss en tidsmessig fleksibilitet innad i inkrementene.

Vedlegg I. Planlagt prosjektplan



Vedlegg J. Faktisk prosjektplan