# Indledning

# Håndtering af use cases

## Start manuel vanding

# 

Figur 1: FlexPMS' håndtering af at starte manuel vanding

## Stop manuel vanding



Figur 2: FlexPMS' håndtering af at stoppe manuel vanding

## Åben indløbsventil

## 

Figur 3: FlexPMS' håndtering af at åbne indløbsventilen

## Luk indløbsventil

## 

Figur 4: FlexPMS' håndtering af at lukke indløbsventilen

## Åben afløbsventil

## 

Figur 5: FlexPMS' håndtering af at åbne afløbsventilen

## Luk afløbsventil

## 

Figur 6: FlexPMS' håndtering af at lukke afløbsventilen

# Threading

FlexPMS er dybt afhængig af trådteknologi. De tre store komponenter, Socket serveren, Bridge og Kar bus, kører parallelt i hver sin tråd. Trådene kommunikerer med hinanden gennem et event-baseret beskedsystem. Al trådhåndtering er skrevet specifikt til at køre på Linux, og FlexPMS er derfor ikke understøttet af andre operativsystemer.



## Thread

*Thread* er en abstrakt basis-klasse for alle klasser, som skal afvikles i sin egen tråd. Ved at nedarve fra *Thread* kan en klasse nøjes med at implementere en run() metode, der kaldes, når tråden startes via start(). Tråden lever indtil run() returnerer, eller indtil der kaldes cancel() på en tråd, og tråden dør. *Thread* er udelukkende skrevet til at understøtte *pthread* på Linux.



Figur 7: Livscyklus for en tråd

### Arkitekturspecifikke metoder

void start()  
 Beskrivelse: Starter tråden.  
 Returværdi: Ingen

void cancel()  
 Beskrivelse: Stopper tråden, hvis annullering er slået til, ellers gør funktionen ingenting. Tråden stoppes først når der stødes på et såkaldt cancellation point. Kun tråden selv kan tillade eller forbyde annullering.  
 Returværdi: Ingen

void join()  
 Beskrivelse: Blokerende kald, som ikke returnerer før tråden er færdig med eksekvering.  
 Returværdi: Ingen

virtual void run()  
 Beskrivelse: Abstrakt metode, som kaldes når tråden startes. Tråden lever så længe run() er under afvikling, eller indtil den annulleres.  
 Returværdi: Ingen

static void\* run\_thread(void\* arg)  
 Beskrivelse: C-style funktion hvori tråden startes. Denne funktion kaldes af start() og kalder til gengæld run() på Thread-objektet.  
 Returværdi: Ingen  
 arg: En pointer til det Thread-objekt, som skal køres i en tråd

void enable\_cancel()  
 Beskrivelse: Tillader annullering af tråden, så tråden kan stoppes hvis cancel() kaldes.  
 Returværdi: Ingen

void disable\_cancel()  
 Beskrivelse: Forbyder annullering af tråden, så hvis cancel() kaldes ignoreres det.  
 Returværdi: Ingen

void ssleep(unsigned int sec)  
 Beskrivelse: Lægger tråden til at sove i et antal sekunder (minimum).  
 Returværdi: Ingen  
 sec: Antal sekunder tråden minimum skal sove i

void msleep(unsigned int msec)  
 Beskrivelse: Lægger tråden til at sove i et antal millisekunder (minimum).  
 Returværdi: Ingen  
 msec: Antal millisekunder tråden minimum skal sove i

# Event-baseret beskedsystem

FlexPMS er opbygget af adskillige tråde, som alle kan snakke sammen ved at sende beskeder til hinanden. Trådene håndterer udelukkende beskeder sendt til dem udefra, og står derfor udelukkende i blokerende kald til en besked-kø, så længe de ikke er ved at håndtere en indkommende besked. Trådene nedarver fra *MessageThread* og har pointers til de tråde, som de skal kunne sende beskeder til.

## 

Figur 8: Et eksempel på event-baseret kommunikation initieret af MessageThread1



## MessageThread

Klassen, som er en specialisering af *Thread*, stiller funktionalitet til rådighed til at indgå i det event-baserede beskedsystem. Ved at nedarve fra *MessageThread* bliver en klasse til en modtager af beskeder, og kan i den forbindelse nøjes med at implementere en dispatch() metode, som kaldes hver gang tråden modtager en besked via dens send() metode. dispatch() modtager to argumenter; et event-ID samt en pointer til et *Message*-objekt, der evt. kan være NULL. Dispatch bør overholde reglen om, at kalde en funktion til at håndtere beskeden alt efter hvilket event-ID den modtager.

send() tager ligeledes to argumenter; et event-ID samt en pointer til et *Message*-objekt. Det er afsenderen, som skal allokere *Message*-objektet, men *MessageThread* sørger selv for, at de-allokere det efter dispatch() er kaldt hos modtageren.

*MessageThread* laver udelukkende blokerende kald til dens besked-kø, og dermed undgår vi, at stå og bruge CPU-tid i løkker, som ikke udfører noget. Det betyder, at alle klasse som nedarver fra *MessageThread* udelukkende håndterer events sendt til dem udefra. På den måde lægges tråde til at sove så længe der ikke er noget at lave, og programmet vil bruge minimalt CPU-tid.



Figur 9: MessageThreads håndtering af events

### Arkitekturspecifikke metoder

virtual void init()  
 Beskrivelse: Abstrakt metode, som kaldes inden der begyndes at hente beskeder fra beskedkøen.  
 Returværdi: Ingen

void send(unsigned long id, Message\* msg = NULL)

Beskrivelse: Lægger en besked i trådens beskedkø.  
Returværdi: Ingen  
id: Et ID, som beskriver det event der sendes  
msg: En pointer til et Message-objekt, som kan holde på yderligere data

virtual void dispatch(unsigned long event\_id, Message\* msg)  
 Beskrivelse: Abstrakt metode, som kaldes hver gang tråden modtager en besked. Metoden skal overskrives af klasser, som nedarver fra MessageThread til at håndtere indkommende beskeder.  
 Returværdi: Ingen  
 event\_id: Et ID, som beskriver det event der sendes  
 msg: En pointer til et Message-objekt, som kan holde på yderligere data. Kan være NULL

## MessageQueue

Klassen er en FIFO kø, som er trådsikret, dvs. sikret mod de problemer der kan opstå i forbindelse med at tilgå den parallelt fra forskellige tråde. *MessageQueue* er implementeret via en *queue* (fra STL) og beytter sig at pthread’s *mutex* og *conditional variable* til at synkronisere mellem tråde. *MessageQueue* er udelukkende brugt internt i *MessageThread*.



Figur 10: MessageQueue FIFO

### Arkitekturspecifikke metoder

void send(unsigned long id, Message\* msg = NULL)  
 Beskrivelse: Putter en besked i køen.  
 Returværdi: Ingen  
 id: Event-ID som skal puttes i køen  
 msg: Pointer til Message-objekt, som skal puttes i køen

Message\* recieve(unsigned long& id)  
 Beskrivelse: Henter den næste besked fra køen. Hvis køen er tom, så blokerer funktionen indtil der bliver puttet noget i køen via send()  
 Returværdi: Pointer til Message-object. Kan være NULL  
 id: Funktionen skriver event-ID til denne variabel

## Item

Struct, som udelukkende benyttes internt i MessageQueue. Den holder på event ID’er og Message-objekter, og er den type, der placeres i køen.

## Message

*Message* gør det muligt at medsende informationer (ud over et event-ID), når tråde kommunikerer med hinanden. Klassen *Message* indeholder en sender attribut, som er en pointer til den *MessageThread*, der sendte beskeden. sender gør det derfor muligt for *MessageThread*’s at svare på beskeder direkte tilbage til afsenderen.

Alt efter hvilke data man vil sende med en besked kan man nedarve fra *Message* og tilføje flere attributter. Der er lavet specialiseringer af *Message* hvor det har været nødvendigt at sende information mellem tråde.

### KarBusMessage

Klassen er en specialisering af *Message* og indeholder – ud over sender – en kar attribut, som er en pointer til det Kar-objekt, der skal sendes data til. Grunden til, at kar kun er implementeret på *KarBusMessage* er, at det altid er relevant at have information med omkring hvilket kar der skal sendes instrukser til når der kommunikeres mellem *Bridge* og *KarBus*, hvorimod dette ikke er nødvendigt når der kommunikeres mellem *Bridge* og *SocketClient*.

Der er adskillige specialiseringer af *KarBusMessage*, som benyttes alt afhængigt af beskeden (event-ID).

### SessionMessage

Klassen er en specialisering af *Message* og bruges i forbindelse med, at *SocketClient* skal registrere sig selv hos *Bridge*. *Bridge* sender en *SessionMessage* til *SocketClient* når klienten er registreret. Klassen indeholder en session\_id attribut.

### GuiMessage

Klassen er en specialisering af *Message* og indeholder – ud over sender – flere attributter, som bruges i forbindelse med kommunikation mellem *SocketClient* og *Bridge*. Klienten (webserveren) har mulighed for at sende kommandoer, som er specifikke for enten Kar eller SensorØ, og det er derfor relevant at tilføje disse attributter på *GuiMessage*. Klassen indeholder desuden også en session\_id attribut, så *Bridge* ved hvilken klient den skal sende til i tilfælde, hvor svar til klienten er nødvendigt.

# Logging

Vi benytter logging til at gemme alle handlinger, som har med sensorer og aktuatorer at gøre. Loggen skrives til en fil med tidsstempel. Det er udelukkende *Bridge*, som skriver til loggen.

Følgende handlinger bliver logget:

* Der modtages data fra sensorer på et Kar
* Der modtages data fra sensorer på en SensorØ
* Der modtages information om, hvorvidt en ventil på et Kar er blevet åbnet eller lukket
* Der modtages information om, hvorvidt en ventil på en SensorØ er blevet åbnet eller lukket
* Der modtages information om en pumpes status (slukket eller tændt/pumpe-hastighed)
* Når GUI anmoder om at starte manuel vanding logges der, for hver SensorØ, at dens ventil blev anmodet om at åbne. Der logges, at Karrets pumpe blev anmodet om at startet
* Når GUI anmoder om at stoppe manuel vanding logges der, for hver SensorØ, at dens ventil blev anmodet om at lukke. Der logges, at Karrets pumpe blev anmodet om at stoppet
* Når GUI anmoder om at åbne eller lukke indløbs- eller afløbsventil på et Kar

Alt andet debugging skrives til FlexPMS’ stdout. Det er derfor muligt at omdirigere stdout til en fil for at logge samtlige debugging udskrifter.



## Log

Klassen benyttes til at skrive til loggen. Den er implementeret vha. Singleton, dvs. at kun én instans af klassen kan eksistere på tværs af hele FlexPMS. Instansen bliver oprettet gennem kald fra main.cpp, hvorefter den lever som en statisk member på klassen Log så længe FlexPMS kører.

Hver gang der skrives til loggen tilføjes tidsstempel samt linjeskift efter teksten.

### Arkitekturspecifikke metoder

static Log\* getInstance()  
 Beskrivelse: Funktionen returnerer en instans af Log. Hvis en instans endnu ikke er blevet oprettet, så oprettes den først.  
 Returværdi: Instans af klassen Log

void write(std::string line)  
 Beskrivelse: Funktionen skriver en linje til logfilen.  
 Returværdi: Ingen  
 line: Tekst, som skal skrives til logfilen

# Database og domænemodeller

Systemet benytter sig af en MySQL database. For at forbinde til databasen gennem FlexPMS benyttes det officielle bibliotek fra MySQL til at forbinde gennem C++, *mysqlcppconn*. FlexPMS kører i øvrigt ikke transaktionsbaseret.

Tilgang til databasen er pakket ind i forskellige domæneklasser, som håndterer de lavpraktiske SQL forespørgsler. Domæneklasserne stiller en højniveau-grænseflade til rådighed for resten af programmet.

Forbindelsen til databasen oprettes i main.cpp når FlexPMS starter, og holdes åben så længe programmet lever. Der gives en pointer til DB-forbindelsen med til *Bridge*, som er den eneste klasse, der arbejder med domæneklasserne.



## Kar

Klassen repræsenterer ét Kar, og stiller en højniveau-grænseflade til rådighed, til at opdatere værdier for et kar i databasen.

### Arkitekturspecifikke metoder

void set\_mwstatus(bool s)  
 Beskrivelse: Funktionen sætter status for manuel vanding.  
 Returværdi: Ingen  
 s: Status for manuel vanding. True er tændt, False er slukket

void set\_ivalvestatus(bool s)  
 Beskrivelse: Funktionen sætter status for, hvorvidt indløbsventilen er åben eller lukket  
 Returværdi: Ingen  
 s: Status for indløbsventilen. True er åben, False er lukket

void set\_ovalvestatus(bool s)  
 Beskrivelse: Funktionen sætter status for, hvorvidt udløbsventilen er åben eller lukket  
 Returværdi: Ingen  
 s: Status for udløbsventilen. True er åben, False er lukket

void add\_sensor\_data(int type, double value)  
 Beskrivelse: Funktionen registrerer data fra en sensor koblet til karret  
 Returværdi: Ingen  
 type:   
 value: Den målte værdi

## SensorOe

Klassen repræsenterer én sensor ø, stiller en højniveau-grænseflade til rådighed, til at opdatere værdier for en sensor ø i databasen.

### Arkitekturspecifikke metoder

void set\_valvestatus(bool s)  
 Beskrivelse: Funktionen sætter status for, hvorvidt ventilen er åben eller lukket  
 Returværdi: Ingen  
 s: Status for ventilen. True er åben, False er lukket

void add\_sensor\_data(int type, double value)  
 Beskrivelse: Funktionen registrerer data fra en sensor koblet til sensor ø’en  
 Returværdi: Ingen  
 type:   
 value: Den målte værdi

## DBContainer

*DBContainer* er en abstrakt basis-klasse for alle klasser, som skal holde på lister af objekter. *DBContainer* implementerer funktionalitet til at tilgå en række af resultater fra et databaseudtræk. Den nedarvede klasse skal selv implementere databaseudtrækket via reload() metoden, men får derudover stillet alle andre nødvendige metoder til rådighed til at tilgå resultaterne.

*DBContainer* er implementeret med et C++ map (fra STL), hvor nøglen er et unikt ID og værdien er et objekt, som repræsenterer én række i databasen. Da *DBContainer* er en template-klasse kan objekterne være af hvilken som helst type. Nøglerne skal dog være af typen unsigned int.

Nedarvede klasser skal implementere reload() metoden, som kaldes umiddelbart efter at objektet er blevet instantieret. reload() metodens formål er, at hente data fra databasen, instantiere objekter og tilføje dem til mappet.

### Arkitekturspecifikke metoder

I beskrivelsen af metoderne nedenfor er T typen på de objekter, som opbevares i mappet.

T\* get(unsigned int id)  
 Beskrivelse: Returnerer en pointer til objektet med nøglen id. Hvis nøglen ikke findes returneres NULL  
 Returværdi: En pointer til objektet med nøglen id. Hvis nøglen ikke findes returneres NULL  
 id: Primærnøglen på det objekt, som skal søges efter

bool contains(unsigned int id)  
 Beskrivelse: Tjekker hvorvidt et objekt eksisterer i mappet  
 Returværdi: True hvis objektet eksisterer, ellers False  
 id: Primærnøglen på det objekt, som skal søges efter

const unsigned int size()  
 Beskrivelse: Tæller antallet af objekter i mappet  
 Returværdi: Antallet af objekter i mappet

void iter()  
 Beskrivelse: Forbereder objektet til at blive itereret over fra begyndelsen.  
 Returværdi: Ingen

T\* next()  
 Beskrivelse: Returnerer det næste objekt i en iteration. Kald iter() inden iterationen begyndes for at være sikker på, at der startes fra begyndelsen.  
 Returværdi: Det næste objekt af typen T

virtual void reload() = 0  
 Beskrivelse: Abstrakt metode, som skal implementeres af nedarvede klasser. Dens formål er, at hente data fra databasen og putte det ind i mappet. Funktionen kaldes umiddelbart efter at *DBContainer* bliver instantieret.  
 Returværdi: Ingen

## KarContainer

Klasse, som er en specialisering af *DBContainer*, giver mulighed for at arbejde med en liste af *Kar*-objekter.

## SensoeOeContainer

Klasse, som er en specialisering af *DBContainer*, giver mulighed for at arbejde med en liste af *SensorOe*-objekter.

# Socket server

Denne del af FlexPMS giver GUI’en adgang til at kommunikere direkte med FlexPMS via TCP/IP, og bruges til at informere FlexPMS om handlinger, som skal startes eller stoppes, og er den eneste direkte vej for GUI at kommunikere med FlexPMS.

## SocketServer

Klassen, som er en specialisering af *MessageThread*, har det ene formål, at lytte efter indkommende forbindelser fra GUI’en over TCP/IP. Når *SocketServer* modtager en ny forbindelse startes en ny tråd, *SocketClient*, hvori al kommunikation mellem GUI og FlexPMS foregår. Når *SocketServer* har oprettet og startet en ny *SocketClien*t mister den al forbindelse til den, og kender derfor ikke til åbne forbindelser til klienter.

Serveren lytter på adresse 127.0.0.1 (localhost) port 5555.



Figur 11: SocketServer’s håndtering af indkommende forbindelser

## SocketClient

*SocketClient,* som er en specialisering af *MessageThread*, håndterer al kommunikation mellem klienten (GUI) og *Bridge*. Den bliver oprettet af *SocketServer* når der kommer en ny indkommende forbindelse. Den består desuden af en privat *SocketReader* klasse, hvis eneste formål er, at læse data fra en socket. SocketReader er implementeret, så der kan laves blokerende læse-kald fra socket, og på den måde undgår vi, at SocketClient står i løkker hvor den laver to ikke-blokerende kald (henholdsvis at læse fra socket, samt at læse fra sin egen besked-kø).

*SocketClient* modtager beskeder fra enten *SocketReader* eller Bridge. *SocketReader* sender beskeder når der enten er modtaget nyt data fra klienten, eller når klienten lukker forbindelsen. Bridge sender beskeder når der enten er data at sende til klienten, eller når forbindelsen til klienten skal lukkes.

Når *SocketClient* oprettes får den givet en file-descriptor til den socket, som den skal læse fra og skrive til. Lige efter at klassen er blevet oprettet, registrerer den sig hos *Bridge*, der svarer tilbage med et unikt sessions ID, som skal gives med hver gang *SocketClient* sender beskeder til *Bridge*. *Bridge* bruger dette ID til at identificere klienter i tilfælde, hvor der skal sendes et svar tilbage til klienten. Bridge har derfor en intern mapning af, hvilke sessions sessions ID’er der hører til hvilke *SocketClient* klasser. Det er med andre ord *Bridge*, og ikke *SocketServer*, som holder styr over åbne forbindelser til klienter.

Når *SocketClient* er blevet registreret hos Bridge starter den *SocketReader*, som begynder at læse data fra socket. Herefter kan en udveksling af data mellem GUI og FlexPMS begynde.

SocketClient dør når én af tre handlinger finder sted:

1. *SocketReader* fik en fejl, da den forsøgte at læse fra socket
2. *SocketClient* fik en fejl, da den forsøgte at skrive til socket
3. *Bridge* giver besked om, at forbindelsen til klienten skal lukkes

I de to første tilfælde skal *SocketClient* give *Bridge* besked om, at forbindelse til klienten er død, og *SocketClient* skal stoppes. I det sidste tilfælde skal *SocketClient* reagere på beskeden og stoppe sig selv. Herefter ved *Bridge*, at den skal fjerne alle spor af *SocketClient* klassen (herunder session og brugt hukommelse).



Figur 12: State Machine for SocketClient



Figur 13: SocketClient's livscyklus

# Bridge

*Bridge,* som er en specialisering af *MessageThread*, er den centrale controller i FlexPMS. Den kommunikerer med både GUI og RS485, og håndterer al den logik der ligger ind imellem. Det er også *Bridge*, der håndterer timing-baserede events som f.eks. at poll’e data fra kar. Det er også *Bridge*, der tilgår databasen gennem domænemodeller.

## Sessions

Når *Bridge* modtager en forespørgsel fra en *SocketClient* om at blive registreret, så tildeler *Bridge* den forespørgende *SocketClient* det næste ledige sessions ID. Herefter kan *SocketClient* begynde at sende beskeder til *Bridge*, som videreformidler beskederne til *KarBus*. Beskeden til *KarBus* skulle indeholde et sessions ID på den forespørgende *SocketClient*, så i tilfælde af, at *SocketClient* skulle have svar, kunne *KarBus* sende ID’et med tilbage til Bridge, som kunne identificere hvilken *SocketClient* den skulle sende svaret til. Vi fik dog aldrig brug for at sende svar tilbage til *SocketClient*, så sessions ID’er er ikke blevet implementeret i kommunikationen mellem *Bridge* og *KarBus*.

Se Figur X for en SocketClient’s livscyklus, som inkluderer registrering med sessioner.