



SILVER BULLET SORT

4. SEMESTERPROJEKT

Systemarkitektur for Silver Bullet Sort

Author:

Gruppe 5

Supervisor:

Poul Ejnar ROVSING

29. maj 2012

Versionshistorie:

Ver.	Dato	Initialer	Beskrivelse
1.0	02-04-2012	RAB og BH	Opsætning af skabelon til rapportskrivning
1.1	04-04-2012	LH og LA	Use Cases påbegyndes med beskrivelse.
1.2	10-04-2012	CSJ	Hardware afsnit tilføjet
1.3	29-04-2012	RAB	Identify tilføjet
1.4	02-05-2012	CW	Database tilføjet
1.5	09-05-2012	LS	IDE tilføjet
1.6	11-05-2012	KV	Diagrammer af Use Cases tilføjet
1.7	15-05-2012	BH	Rettet dokument
2.0	20-05-2012	Gruppe 5	Sidste diagrammer tilføjet

Godkendelsesformular:

Forfatter(e):	Michael Bojsen-Hansen (MBH) Kasper Vinther Andersen (KVA) Lars Anker Christensen (LA) Lasse Hansen (LH) Christian Smidt-Jensen (CSJ) Rasmus Bækgaard (RAB) Christoffer Lousdahl Werge (CW) Lasse Sørensen (LS)
Godkendes af:	Poul Ejnar Rovsing.
Projektnr.:	4. semesterprojekt.
Filnavn:	SBS_Systemarkitektur.pdf
Antal sider:	118
Kunde:	Poul Ejnar Rovsing (PER).

Sted og dato: _____

09421 Lasse Lindsted Sørensen

10063 Lasse Hansen

10648 Lars Anker Christensen

10719 Michael Bojsen-Hansen

10750 Kasper Vinther Andersen

10770 Christian Smidt-Jensen

10832 Christoffer Lousdahl Werge

PER Poul Ejnar Rovsing

10893 Rasmus Bækgaard

Indhold

1 INTRODUKTION	8
1.1 Formål	8
1.2 Referencer	9
1.3 Definitioner og forkortelser	9
1.4 Dokumentstruktur og læsevejledning	9
1.5 Dokumentets rolle i en iterativ udviklingsproces	11
2 SYSTEM OVERSIGT	12
2.1 System kontekst	12
2.2 System introduktion	13
3 SYSTEMETS GRÆNSEFLADER	14
3.1 Grænseflader til person aktører	14
3.1.1 Operatør	14
3.1.2 Programmør	17
3.2 Grænseflader til eksterne system aktører	20
3.2.1 Database	20
3.3 Grænseflader til hardware aktører	20
3.4 Grænseflader til eksternt software	20
3.4.1 USBC.dll	21
4 USE CASE VIEW	24
4.1 Oversigt over arkitektursignifikante Use Cases	24
4.2 Use Case 1 scenarier	26
4.2.1 Use Case mål	26
4.2.2 Use Case scenarier	26
4.2.3 Use Case undtagelser	26
4.3 Use Case 1.1 scenarier	27
4.3.1 Use Case mål	27
4.3.2 Use Case scenarier	27
4.4 Use Case 1.2 scenarier	27

4.4.1	Use Case mål	27
4.4.2	Use Case scenarier	27
4.4.3	Use Case undtagelser	28
4.5	Use Case 2 scenarier	28
4.5.1	Use Case mål	28
4.5.2	Use Case scenarier	28
4.5.3	Use Case Undtagelser	29
4.6	Use Case 3 scenarier	29
4.6.1	Use Case mål	29
4.6.2	Use Case scenarier	29
4.6.3	Use Case Undtagelser	30
4.7	Use Case 4 scenarier	31
4.7.1	Use Case mål	31
4.7.2	Use Case scenarier	31
4.7.3	Use Case Undtagelser	31
4.8	Use Case 5 scenarier	32
4.8.1	Use Case mål	32
4.8.2	Use Case scenarier	32
4.8.3	Use Case Undtagelser	32
5	LOGISK VIEW	33
5.1	Oversigt	33
5.2	Arkitektursignifikante designpakker	35
5.2.1	Pakke 1 - Presentation Layer	37
5.2.2	Pakke 2 - Business logic layer	37
5.2.3	Pakke 3 - Data access layer	37
5.2.4	Pakke 4 - Hardware layer	37
5.3	Use Case realiseringer	37
5.3.1	Use Case 1: Sorter klods realisering	37
5.3.2	Use Case 1.1: Mål og vej klods realisering	39
5.3.3	Use Case 1.2: Bestem materialetype realisering	41
5.3.4	Use Case 2: Programmer robot realisering	41

5.3.5	Use Case 3: Rediger materialetype realisering	42
5.3.6	Use Case 4: Tilgå log realisering	45
5.3.7	Use Case 5: Test program realisering	45
6	PROCES/TASK VIEW	46
6.1	Oversigt over processer/task	46
6.2	Proces/task kommunikation og synkronisering	47
6.3	Procesgruppe 1	48
6.3.1	Proceskommunikation i gruppe 1	48
7	DEPLOYMENT VIEW	52
7.1	Oversigt over systemkonfigureringer	52
7.2	Node-beskrivelser	52
7.2.1	Node 1. beskrivelse - Client PC	52
7.2.2	Node 2 beskrivelse - ATmega16 microcontroller	52
7.2.3	Node 3 beskrivelse - Relationel database	53
8	IMPLEMENTERINGS VIEW	54
8.1	Oversigt	54
8.2	Komponentbeskrivelser	54
8.2.1	Komponent 1: Sorteringsprogram	54
8.2.2	Komponent 2: Simulering	63
8.2.3	Komponent 3: Bestem placering af Transportbånd	67
8.2.4	Komponent 4: Weight	78
8.2.5	Komponent 5: Log	81
8.2.6	Komponent 6: Beskedkø, observer pattern og database tilgang	83
8.2.7	Komponent 7: GUI/MVVM	88
8.2.8	Komponent 8: IDE	95
9	DATA VIEW	96
9.1	Data model	96
9.1.1	Design af database	96
9.1.2	Triggers og stored procedures	97

9.2 Implementering af persistens	100
10 GENERELLE DESIGNBESLUTNINGER	103
10.1 Arkitektur mål og begrænsninger	103
10.2 Arkitektur mønstre	103
10.3 Generelle brugergrænsefladeregler	105
10.3.1 Arkitekturspecifikke	105
10.3.2 Udseendesspecifikke	105
10.4 Exception og fejlhåndtering	106
10.4.1 Exception i database tilgangen	106
10.4.2 Exception i serial communication	106
10.4.3 Exception i Robot funktionerne	106
10.5 Implementeringssprog og værktøjer	107
10.5.1 C	107
10.5.2 C++	107
10.5.3 C#	107
10.5.4 Microsoft Visual Studio 2010	107
10.5.5 Visual paradigm	107
10.5.6 AnkhSVN og tortoiseSVN	107
10.5.7 TexMaker	107
10.5.8 MSSQL server management studio	108
10.6 Implementeringsbiblioteker	108
11 STØRRELSE OG YDELSE	109
12 KVALITET	110
12.1 Brugervenlighed	110
12.2 Pålidelighed	110
12.3 Integritet	110
13 OVERSÆTTELSE	111
13.1 Oversættelses-hardware	111
13.2 Oversættelses-software	111

INDHOLD

13.3 Installation	111
14 KØRSEL	112
14.1 Kørsels-hardware	112
14.1.1 Opstilling	112
14.2 Kørsels-software	112
14.3 Start, genstart og stop	113
14.4 Fejludskrifter	113
Bilag	118

1 INTRODUKTION

Dette dokument beskriver designet og arkitekturen af softwaren for I4PRJ Gruppe 5 - Silver Bullet Sort. Softwaredesignet og arkitekturen er opbygget ud fra Use Cases fundet i *SBS_Kravspecifikation*.

I dette dokument findes blandt andet:

- En systemoversigt samt dets grænseflader
- Use Case view
- Logisk view
- Proces og task view
- Deployment view
- Implementerings view
- Data view

I disse forskellige views kan der findes domænemodel, klassediagrammer, sekvensdiagrammer og lignende. Derudover findes diverse illustrationer, matematik og kodeeksempler. Alle disse elementer samt øvrige dele, er med til at beskrive systemet i detaljer. I løbet af dette dokument vil der blive brugt diagrammer såvel som figurer. Disse kan alle findes i fuld størrelse i bilaget. I hvert afsnit vil der refereres til hvor i bilaget tingene kan findes

1.1 Formål

Formålet med dokumentet er:

- At fastlægge systemets overordnede design, arkitektur og virkemåde. Dette er gjort jævnfør kravspecifikationen.
- At definerer og beskrive de nødvendige klasser og vise samspillet mellem disse.
- At give individer med tilpas faglig viden indblik i systemets opbygning.

Dokumentets målgruppe er en person med grundlæggende viden om det overordnede faglige emne, dog uden viden inden for det specifikke emne.

1.2 Referencer

I dette dokument vil der optræde referencer til følgende dokumenter:

- SBS_Kravsspecifikation
- SBS_HardwareSpecifikation
- SBS_Database
- SBS_Processrapport
- SBS_IDE-funktioner
- USBC-documentation

De ovenstående dokumenter kan findes i medfølgende bilagsmappe.

1.3 Definitioner og forkortelser

- dll - Dynamic Link Library
- MVVM - Designmønstret Model - View - ViewModel
- GUI - Graphical User Interface, brugergrænsefladen
- IDE - Integrated development environment, software udviklingsmiljø
- WPF - Windows Presentation Foundation, system til udvikling af grafisk applikationer på Windows.

1.4 Dokumentstruktur og læsevejledning

Afsnit 1 – INTRODUKTION: Introducerer læseren til projektet og dokumentet.

Afsnit 2 - SYSTEM OVERSIGT: Dette afsnit giver en kort oversigt over systemet og dets omgivelser. Derudover vises der diagrammer over systemets aktører, samtidig med at systemet kort beskrives.

Afsnit 3 – SYSTEMETS GRÆNSEFLADER: Her beskrives grænsefladerne for de forskellige aktører til systemet.

Afsnit 4 – USE CASE VIEW: Afsnittet beskriver alle Use Cases og Use Case scenarier fra Use Case modellen. De forskellige Use Cases beskrives i prosa form, samtidig med at et use case diagram præsenteres.

Afsnit 5 – LOGISK VIEW: Beskriver systemets opdeling i delsystemer og pakker og deres organisering i en lagdelt struktur. Viser hvordan de forskellige Use Cases er realiseret, blandt andet ved brug af simple- og udvidede systemsekvensdiagrammer.

Afsnit 6 – PROCES/TASK VIEW: Dette afsnit beskriver systemets opdeling i processer og tråde, og hvorledes disse processer kommunikerer og synkroniserer. Giver et overblik over de forskellige tråde anvendt i systemet.

Afsnit 7 – DEPLOYMENT VIEW: Viser den fysiske struktur af systemet, hvilket er computere og andre hardware enheder.

Afsnit 8 – IMPLEMENTERINGS VIEW: Dette afsnit beskriver den endelige implementering. Her er de forskellige komponenter beskrevet i detaljer.

Afsnit 9 – DATA VIEW: Er en beskrivelse af persistent data lagring i systemet.

Afsnit 10 – GENERELLE DESIGNBESLUTNINGER: Dette afsnit fastholder de generelle designbeslutninger, der tages under arkitekturdesignet eller som er givet som ultimative krav. Her beskrives systemets begrænsninger, anvendte designmønstre, samt anvendte værktøjer og (kode)biblioteker.

Afsnit 11 – STØRRELSE OG YDELSE: I dette afsnit er angivet de kritiske størrelser og ydelsesparametre for systemet..

Afsnit 12 – KVALITET: Her opremmes de kvalitetskrav for systemet der er med til at udforme arkitekturen. Ligeså beskrives hvorledes arkitekturen opfylder andre af de ikke-funktionelle krav vedrørende for eksempel udvidbarhed.

Afsnit 13 – KØRSEL: Her beskrives hvordan programmet installeres og køres.

Afsnit 14 – BILAG: Liste over anvendte bilag.

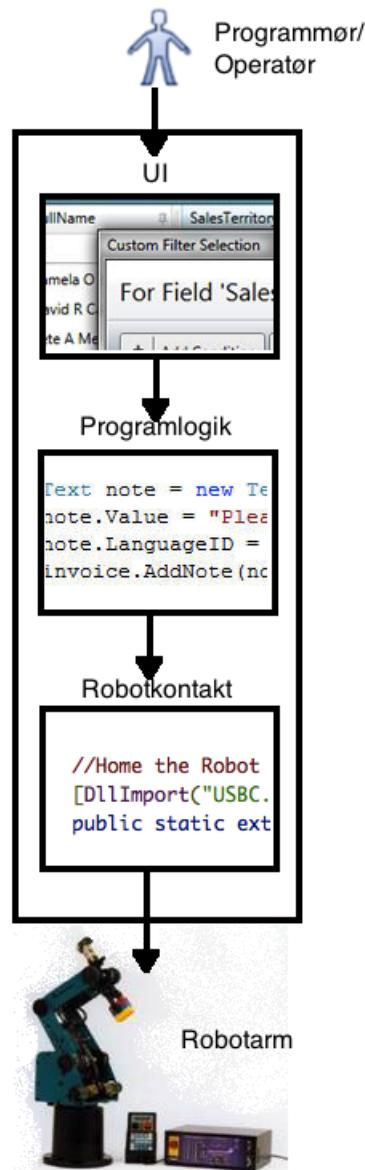
1.5 Dokumentets rolle i en iterativ udviklingsproces

Da der er blevet fulgt en iterativ udviklingsprocess (især retningslinjer fra SCRUM) er systemet blevet udviklet over 6 iterationer. I hver iteration er selve Silver Bullet Sort systemet blevet videreudviklet, og har fået flere funktionaliteter samtidig med at systemet er blevet effektiviseret. At processen er en iterativ udviklingsproces kommer til udtryk ved at dokumentationen er opdateret løbende i takt med at softwaren har udviklet sig, og større erfaring blev opnået. Selve udviklingprocessen beskrives i Processrapporten.

2 SYSTEM OVERSIGT

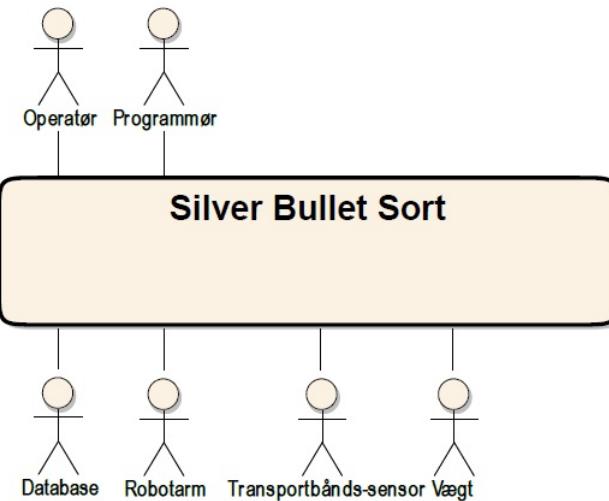
2.1 System kontekst

Systemet består af en udviklet brugergrænseflade og et programmel, samt hardwaren der er udleveret. Systemet kan skitseres som på figur 1.



Figur 1: Simpel systemoversigt

Systemets påvirkninger fra omverden, kan overskueliggøres via et aktørkontekst-diagram vist på figur 2.



Figur 2: Aktørkontekst-diagram

2.2 System introduktion

Systemet hovedfunktion er, at sorterer klodser efter materialetype, samt lade en programmør programmere egne programmer til systemet.

Programmøren tilgår systemet vha. en GUI og har herefter mulighed for at starte系统的 forskellige programmer. Derudover har han mulighed for at skrive egne programmer, for at tilgå databasen og redigere i materialetyper og klodser samt mulighed for at indstille placeringen af transportbåndet klodserne kommer kørende på.

Hvis den rigtige robot ikke skulle være tilgængelig, er der mulighed for at simulere programmerne, hvorved alle informationer om hvad robotten ville gøre, bliver skrevet ud på skærmen samt lagt i en logfil. Disse logfiler kan naturligvis tilgås. Der bliver også oprettet logfiler, når ens program køres med den rigtige robot. Disse kan naturligvis også tilgås.

Ligeledes skal systemet kunne betjenes af en operatør, som dog ikke har mulighed for at omprogrammere robotten. Han har kun mulighed for at køre standardprogrammet (der sorterer klodserne efter materialetype), eller køre et af de brugerdefinerede programmer programmøren har lavet. Derudover har operatøren også mulighed for at til logfiler.

Sidst skal det nævnes at både operatør og programmør har mulighed for at printe lister over materialer og klodser.

3 SYSTEMETS GRÆNSEFLADER

3.1 Grænseflader til person aktører

Systemet har to forskellige personaktører, der begge fungerer som primæraktører, hvilket også ses ud af Use Case Diagrammet for systemet (Use Case diagram findes i afsnit 4 *USE CASE VIEW* figur 13). Disse to har forskellige mål med systemet, og derfor benytter de hver især forskellige elementer i brugergrænsefladen. De har dermed også forskellige rettigheder, til at bruge nogle af de forskellige elementer.

Begge aktører benytter naturligvis en PC, for at interagere med systemet. De starter i et log-in vindue¹. Her logges der ind som enten programør eller operatør. Brugerfladen ses her nedenfor på figur 3. Herefter kommer brugeren til et vindue, hvor der kan vælges imellem kategorier af funktionaliteter.



Figur 3: Login Vindue

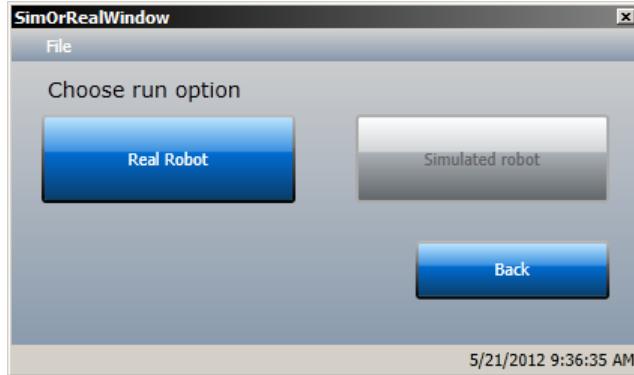
3.1.1 Operatør

Når der logges ind som operatør, har man begrænset adgang til systemet. Operatøren har kun adgang til to af funktionaliteterne i hovedvinduet, som det ses på figur 4. Disse er Show log og Run, hvor Run kun leder til benyttelse af den rigtige robot, og ikke har mulighed for at simuler den. Programmøren er den eneste der kan simulere robotten, se figur 5. Brugergrænsefladen for operatøren ses på nedenstående billeder 4:

¹ Alle billeder i afsnittet kan findes i *Bilag 3: Billeder/Brugergrænseflade*



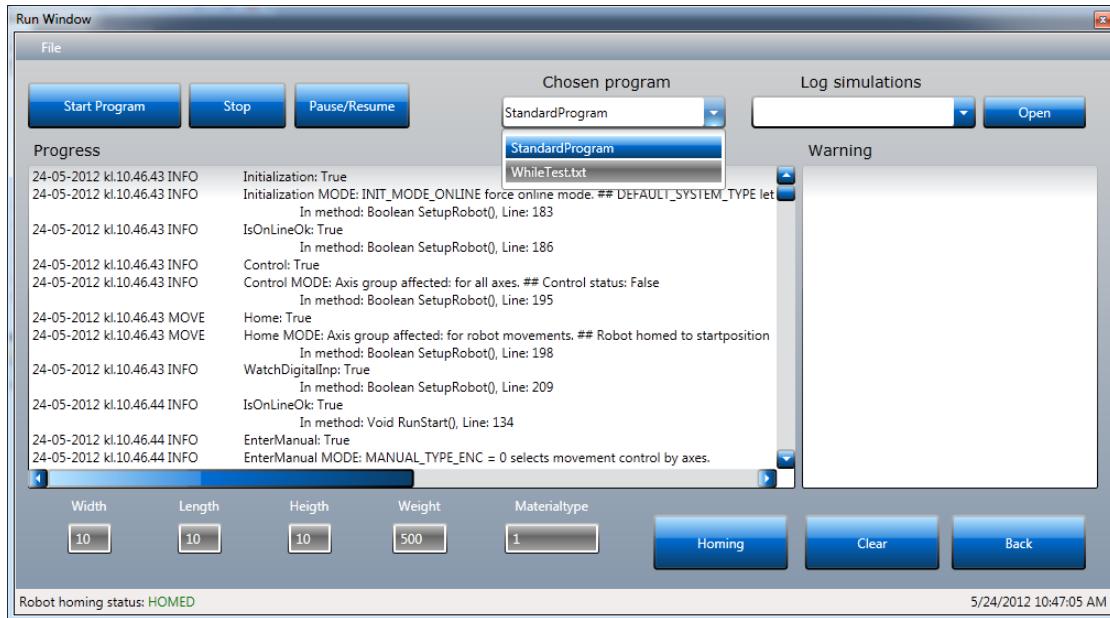
Figur 4: Hovedmenu vinduet set fra Operatørens rettigheder. Som det ses har operatøren kun tilgang til Run og Show log. Det ses yderligere at "Admin Permissions" er "Not Granted"



Figur 5: Valg af robottype vindue

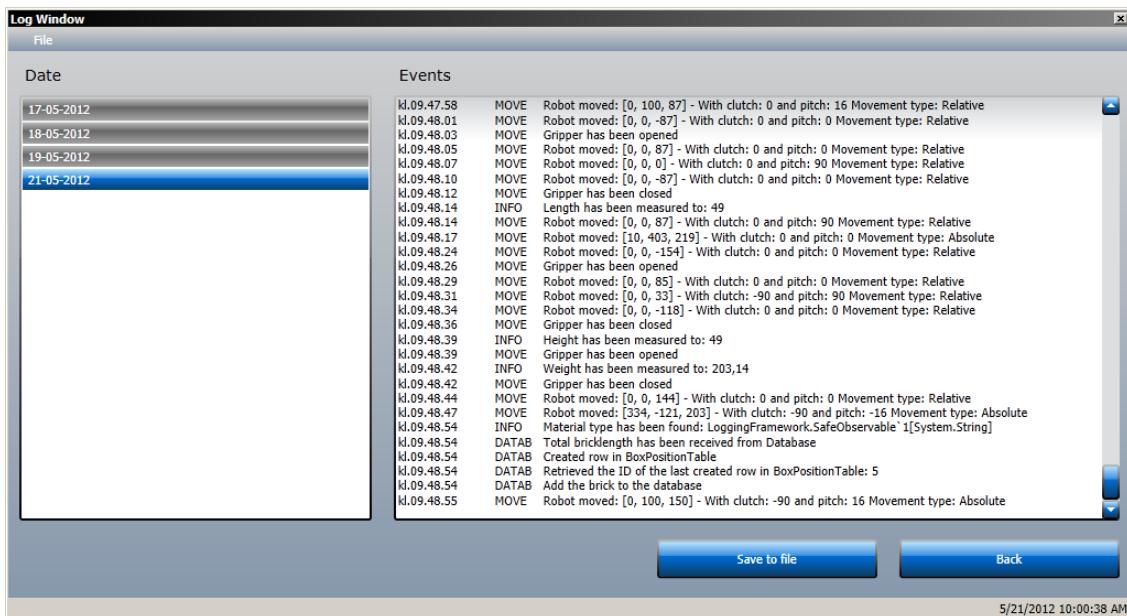
Nedenstående billeder 6, viser grænsefladen, hvor operatøren kører programmerne ud fra. Operatøren har naturligvis mulighed for at starte systemet med en af de brugerdefinerede programmer eller med standardprogrammet. Derudover kan operatøren stoppe og resette programmet, hvis der skulle opstå problemer, eller blot pause programmet. Til at holde øje med systemet, er der først og fremmest en processboks, der viser hvad robotten laver og har lavet. Operatøren kan også hurtigt få et overblik, over den aktuelle klods der sorteres nederst i vinduet, hvor der opdateres data omkring kloksen. Derover er der en notifikationsboks i siden, hvor der kommer beskeder, hvis der er noget operatøren skal være opmærksom på.

3 SYSTEMETS GRÆNSEFLADER



Figur 6: Run vinduet, med logging undervejs.

Nedenstående billede 7, viser grænsefladen, hvor operatøren kan tilgå de logs der er lavet ved hver programkørsel. Han har mulighed for at gemme en selvvalgt log til en tekstfil, og dermed behøves programmet ikke køres, hvis han vil tilgå en logfil flere gange.



Figur 7: Log vinduet, med tidligere logs.

3.1.2 Programmør

Programmøren har samme grafiske brugergrænseflade som operatøren, dog med fuld tilgængelighed. Brugergrænsefladen for operatøren ses på nedenstående figur 8:

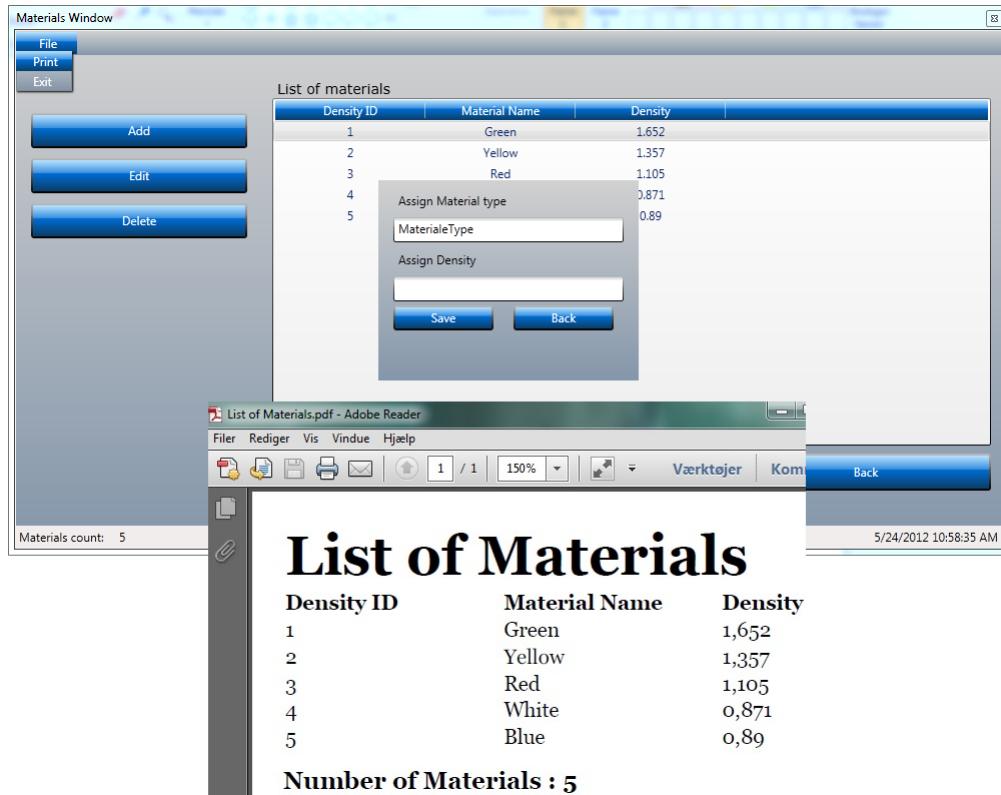


Figur 8: Hovedmenu vinduet ser fra Programmørens rettigheder. Som det ses har programmøren tilgang til alt. Det ses yderligere at "Admin Permissions" er "Granted" skrevet med grønt

Nedenstående billede viser log materialevinduet, hvor der er trykket på Add/Edit knappen. Som det ses kan programmøren tilføje en ny materialetype, eller fjerne en eksisterende.² Derudover kan han også redigerer i eksisterede materialetyper, herunder navn og densitet. Derudover kan programmøren printe en listen over materialetyper, via menuen i toppen, hvilket der også ses et lille eksempel her på figur 9.

²Det er ikke muligt at slette de fem første densiteter, der svarer til de fem klodser, der er gemt i systemet.

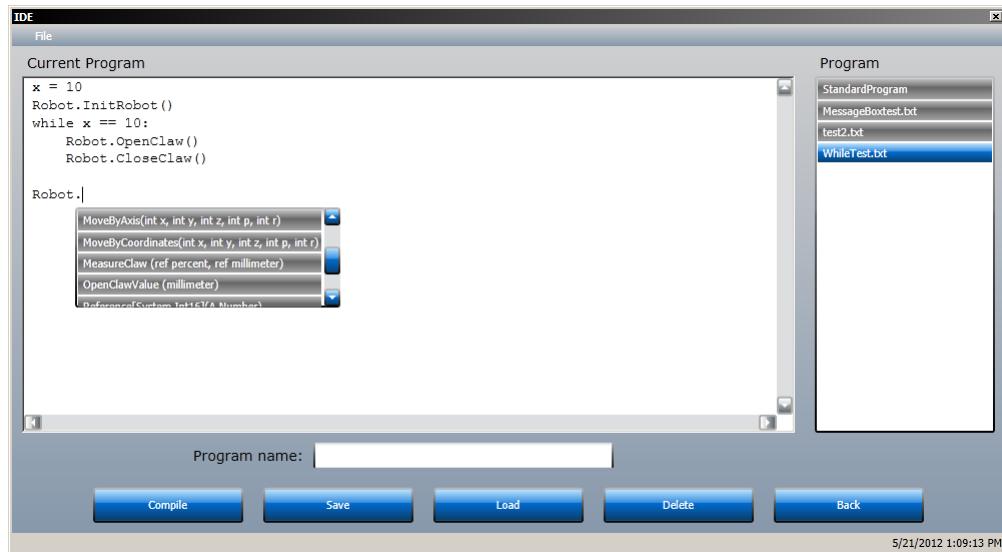
3 SYSTEMETS GRÆNSEFLADER



Figur 9: Materials vinduet

Nedenstående figur 10 viser vinduet, hvor programmøren kan gå ind og lave egne sekvenser til robotten. Når programmøren skriver et program, er der en smule intellisense, der viser de funktioner der er tilgængelige, når han bruger punktum operatoren. Han kan hente gamle programmer og redigerer dem, gemme programmer, slette dem samt compile det program der er indlæst i vinduet, hvor programøren vil få ad vide om der er nogle syntax fejl, og i så fald hvor fejlen er.

3 SYSTEMETS GRÆNSEFLADER



Figur 10: IDE vinduet

Nedenstående billede 11 viser vinduet, hvor programmøren kan få et overblik over de klodser der er sorteret, samt slette hvis han har fjernet nogle fra boksen. Derudover kan han ligesom i materialevinduet printe en listen over sorterede klodser



Box Column	Box Position	Length	Height	Width	Volume	Mass	Density	Material Type
2	0	3,9	5,9	5,9	135,759	192,4	1,357	Yellow
3	0	3,9	5,9	5,9	135,759	153,34	1,105	Red
4	0	4,9	5	5,9	144,55	130,88	0,871	White
1	0	3,9	5,9	5,9	135,759	237,31	1,652	Green
1	1	3,9	3,9	4,9	74,529	130,88	1,652	Green
1	2	4,9	5	5,9	144,55	248,05	1,652	Green
1	3	4,9	4,9	4	96,04	167,01	1,652	Green
1	4	4,9	5	4,9	120,05	209,97	1,652	Green

Figur 11: ManageBricks vinduet

Nedenstående billede 12 viser vinduet, hvor programmøren skal gå ind og angive en række længder, hvis han har rykket om på opstillingen. Når der trykkes i en boks, flyttes pilen på billedet, så programmøren ved hvilken længde han skal angive.



Figur 12: Edit Conveyer Belt Position vinduet

3.2 Grænseflader til eksterne system aktører

Databasen er en ekstern system aktør. Sorteringssystemet gemmer data om de forskellige objekter i denne database.

3.2.1 Database

Der kan læses nærmere om databasens opbygning i dokumentet *SBS_Database*. Kommunikation med databasen sker igennem en beskedkø. For mere detaljeret information se sektion *9.2 Implementering af persistens*

3.3 Grænseflader til hardware aktører

Grænseflader til hardwareaktørerne kan findes i dokumentet: *SBS_HardwareSpec* (se referencedokumenter i bilagsmappen)

3.4 Grænseflader til eksternt software

Programmet skal kommunikere med en dll-fil (Se *Bilag: DLL filer/UDBC.dll*, for at se nærmere på denne), hvor forskellige funktioner til robotten findes. Denne dll-fil var udle-

veret sammen med robotten.

3.4.1 USBC.dll

I USBC.dll-filen findes alle funktioner der er nødvendige, for at få robotten til at bevæge sig, som der ønskes. Funktionerne i biblioteket er dog skrevet i C++, og som det kan læses i *SBS_Kravspecifikation*, benyttes C# som programmeringssprog. Dermed blev kommunikationen med denne en smule mere kompliceret, da C# er managed code og C++ unmanaged code. C++ laver name mangling på funktionerne, og derfor skulle funktionernes "specielle" navn findes.

Et andet problem var, at når en pointer (delegate) i C# sættes til at pege på noget data fra C++, kan garbagecollectoren ikke se, at der bliver peget på noget. Dermed vil garbagecollectoren gå ind og deallokere pointeren. For at løse dette problem er funktioner i marshall biblioteket blevet benyttet. Disse benyttes netop til at få managed kode til at kunne kalde unmanaged kode fra en dll-fil. Dermed ignorerer garbagecollectoren pointer-funktionerne.

De forskellige funktioner til robotten og deres tilhørende beskrivelse er først og fremmest fundet i dokumentet *USBC-documentation*. Der gøres opmærksom på, at ikke alle funktioner er beskrevet fyldestgørende i dette dokument.

Når en funktion skal benyttes, findes den i dll-filen via visual studios kommandoprompt. Her benyttes kommandoens dumpbin på dll-filen, og dermed listes samtlige funktioner med navn, samt hvordan navnet vil se ud i C#. Disse er blevet gemt i et excel dokument (*se bilag: DLL filer/USBC excel*). Det underlige navn der ses, er hvordan funktionsnavnet ser ud i C#.

Fra PInvoke benyttes 'DllImport'. Denne skal kende et EntryPoint, som er navnet på funktionen i dll-filen (navnet som det vil se ud i C#). Dermed kan compileren kende navnet, og nu skal funktionen bare kaldes med en identisk prototype.

Kodeudsnit 1: Funktionskald

```

1 [DllImport("USBC.dll",
2     EntryPoint = "?Initialization@YAHFFP6AXPAX@Z1@Z",
3     CallingConvention = CallingConvention.Cdecl)]
4 public static extern bool Initialization(
5     short initMode,
6     short systemType,
7     CallBackFun.CallBackFunInitEnd fnInitEnd,
8     CallBackFun.CallBackFunError fnErrMessage
9 );

```

Kodeudsnit 1 viser et eksempel på et funktionskald fra dll'en. CallingConvention specificerer den måde funktioner i unmanaged kode skal kaldes på.

Alle funktionskald til dll-filen er lagt i en fil for sig (USBCDLL.cs). I parameterlisten er der to CallBack funktioner. Disse er pointere til andre funktioner. Alle disse CallBack funktioner, som er defineret som delegates, ligger også i deres egen fil, så der er en overskuelig opdeling (CallBackFun.cs).

Udover de omtalte filer, skulle der også laves to klasser, hvor der ligger noget information til de forskellige funktioner i dll'en. Den information funktionerne skulle bruge ligger i to headerfiler("USBCDEF", og "ERROR"), der også er skrevet i C++ (se *Bilag : DLL filer/UDBC.dll*). Disse skal naturligvis også være identiske i vores projekt, og skal derfor konverteres til C#.

Før klassen er der en attribut, som vist på kodeudsnit 2.

Kodeudsnit 2: Titel der skal rettes

```

1 [StructLayout(LayoutKind.Sequential, Pack = 0, CharSet = CharSet.←
    Ansi)]

```

Denne søger for at feltet i klassen bliver lagt i hukommelsen sekventielt, hvilket er vigtigt for at klassen kan bruges og forstås. Derudover bruges [MarshalAs(..)] attributten, over array, string og andre typer der er forskel på i C# og C++. Klassen kan heller ikke bare initieres via new operatoren, da det vil se ud som om de ikke bruges, og dermed vil garbagecollectoren fjerne dem igen. Derfor skal de initieres på en speciel måde, som vist på kodeudsnit 3

Kodeudsnit 3: Titel der skal rettes

```
1 public void InitConfigData(ConfigData configData, IntPtr ←
  configDataPtr)
2 {
3     configData = new ConfigData();
4
5     configDataPtr = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(
  ConfigData)));
6
7     Marshal.StructureToPtr(configData, configDataPtr, false);
8
9 }
```

Via Marshal.AllocHGlobal() allokeres instansen, så garbagecollectoren ikke fjerner den. Parameteren er størrelsen af det, der skal allokeres. Derefter sættes en pointer til at pege på det.

4 USE CASE VIEW

I dette afsnit forklares, hvordan Use Case view'et er sat op, samt hvad de forskellige Use Cases gør.

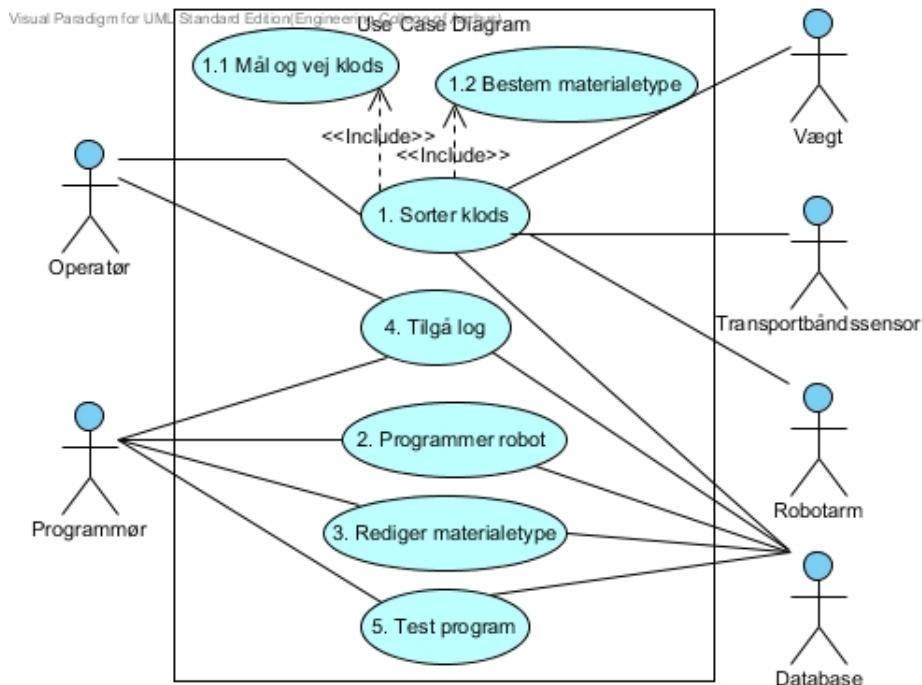
4.1 Oversigt over arkitektursignifikante Use Cases

I dette afsnit er de enkelte Use Cases præsenteret. Alle Use Casene bliver præsenteret, da de alle er centrale for systemets funktionalitet, samtidig med at systemet kun består af fem Use Cases.

Use Casene i systemet er følgende:

- Use Case 1: Sorter klods
 - Use Case 1.1: Mål og vej klods (include)
 - Use Case 1.2: Bestem matrialetype (include)
- Use Case 2: Programmer robot
- Use Case 3: Rediger materialetype
- Use Case 4: Tilgå log
- Use Case 5: Test program

Use case diagram kan findes i bilag under Diagrammer/Use Case Diagram



Figur 13: Use Case diagram

Som det fremgår af Use Case diagrammet, figur 13, er der to primære aktører, operatøren og programmøren, mens forskellige hardware (vægt, robotarm og Transportbåndssensor) og softwareenheder (database) indgår som sekundære aktører.

Ligeledes kan det ses, at systemet har fem Use Cases. *Use Case 1: Sorter Klods* er initialiseret af operatøren, mens *Use Case 2: Programmer robot*, *Use Case 3: Rediger materialetype* og *Use Case 5: Test Program* er initialiseret af programmøren. Det skal nævnes at et af normalforløbene i *Use Case 2: Programmer robot* også kan initialiseres af operatøren. *Use Case 4: Tilgå log* kan både initialiseres af operatøren og af programmøren, da de begge kan have behov for at se, hvad sker i systemet. Include Casene til *Use Case 1: Sorter klods* initieres gennem *Use Case 1: Sorter klods*, da de blot er en hjælp til at få en klods sorteret.

Alle Use Casene har en tæt relation til brugergrænsefladen, og de fleste bliver initialiseret gennem denne, så hvis det ønskes at se eksempler på dette, henvises til afsnit 3.4 *Grænseflader til eksterne softwareaktører*, hvor selve brugergrænsefladen præsenteres og beskrives.

4.2 Use Case 1 scenarier

4.2.1 Use Case mål

Målet med denne Use Case er at få en klods fra transportbåndet sorteret og placeret efter dennes matrialetype og få den lagt i den boks, der indeholder klodser af den givne matrialetype.

4.2.2 Use Case scenarier

Denne Use Case sortere klodser efter matrialetype. Det kræves at robotten er i startposition, samt at den ikke har fat i en klods. Klodserne kommer sekventielt kørerende på transportbåndet, og når en klods registreres af transportbåndssensoren samles denne op af robotarmen. Her tilgåes Use Case 1.1: *Mål og vej klods* (include). Klodsen vendes, således at alle sider måles, og placeres slutteligt på vægten. Dernæst findes matrialetypen gennem Use Case 1.2: *Bestem matrialetype* (include). Her bliver densiteten udregnet, og matrialetypen hentes i databasen, hvorefter den aktuelle klods data persisteres på databasen. Klodsen bliver dernæst placeret i sin respektive kasse, indeholdende den givne matrialetype, hvorefter robotarmen kører til startposition.

Et vigtigt ikke-funktionelt krav for denne Use Case er at det maksimalt må tage 3 minutter at sorterer en klods, altså fra klodsen er registeret på transportbåndet, til den succesfuldt er placeret i den korrekte kasse.

4.2.3 Use Case undtagelser

Undervejs i forløbet har brugeren mulighed for at afvige fra normalforløbet. Der kan trykkes på enten en fysisk, eller softwarebaseret nødstopknap, hvilket stopper sorteringen. Ligeledes kan sorteringsmekanismen stoppes ved tryk på en pauseknap, som stopper sorteringsmekanismen midlertidigt, til det igen startes gennem brugergrænsefladen.

Da der undervejs i Use Casen skrives et loggingindlæg til databasen omkring, hvordan processen forløber, er der risiko for, at der mistes forbindelse til databasen. Hvis dette sker, gemmer systemet selv beskederne og persisterer dem på databasen, når forbindelsen genetableres. Det samme sker, mht. klodsens data. Denne undtagelse er også gældende for de to include Use Cases tilhørende denne Use Case, da de ligeledes laver et logging

indlæg til databasen.

Hvis der ikke er plads til kloden i boksen med den tilhørende materialetype, modtager operatøren en besked herom, hvorefter robotarmen placerer kloden til højre for sensoren på transportbåndet. Dette resulterer i, at kloden kasseres, når sorteringsalgoritmen igen startes.

4.3 Use Case 1.1 scenerier

4.3.1 Use Case mål

Målet med denne Use Case er, at få målt og vejet en given klods, så *Use Case 1: Sorter klods* kan sortere en given klods.

4.3.2 Use Case scenerier

I denne Use Case tager robotarmen fat i kloden, hvorefter den første side måles. Dernæst slippes kloden, og robotarmen løftes og roteres 90 grader, så det er muligt at få fat i kloden på næste led og måle denne side. Når den anden side er målt vendes kloden, hvorefter den slippes på vægten, som vejer kloden. Til sidst tager robotarmen fat i kloden på den sidste led, som så måles.

4.4 Use Case 1.2 scenerier

4.4.1 Use Case mål

Målet med denne Use Case er, at få bestemt materialetypen på den givne klods, således at *Use Case 1: Sorter klods* kan få den sorteret.

4.4.2 Use Case scenerier

Efter at klodens sider og vægt er blevet fundet i *Include Use Case 1.1: Mål og vej klods*, kan rumfanget nu udregnes, og dernæst kan densiteten findes. Denne densitet sammenlignes med de på databasen gemte materialetyper. Dette skulle gerne resulterer i, at klodens materialetype er fundet, og således kan de defineres hvor den skal placeres. Klodens data gemmes så på databasen.

Når klodsens mål, vægt og densitet på databasen, gemmes målene i gram, vægten i centimeter og densiteten i $\frac{g}{cm^3}$.

4.4.3 Use Case undtagelser

Hvis der mistes forbindelse til databasen, mens den aktuelle klods' materialetype sammenlignes med materialetyper på databasen, modtager operatøren en besked der fortæller at forbindelsen er blevet afbrudt. Her får han så mulighed for at vente på at forbindelsen bliver genoprettet, eller lukke hele programmet. Hvis han vælger at vente, vil systemet forsøge at genetablere forbindelsen, og hvis det ikke lykkes, vil han modtage samme besked igen. Hvis forbindelsen bliver genoprettet, kører sorteringsalgoritmen videre.

Det er også risiko for, at den fundne densitet ikke stemmer overens med de på databasen gemte materialer. I dette tilfælde bliver kloden placeret til højre for sensoren på transportbåndet, og således bliver den kasseret, næste gang sorteringsalgoritmen startes.

4.5 Use Case 2 scenarier

4.5.1 Use Case mål

Målet med denne Use Case er at omprogrammere robotten, så funktionaliteten ændres efter programmørens behov. Ligeledes er det et mål, at det nye program bliver anvendt.

4.5.2 Use Case scenarier

De tre første normalforløb i Use Casen initialiseres ved, at programmøren angiver, at et nyt program ønskes defineret. Det eneste krav er, at der er logget ind som programmør. Et kravet overholdt sendes programmøren ind i programmeringsmenuen, hvor gemte programmer bliver listet. Her kan programmøren så enten lave et nyt program, redigere i et eksisterende program, eller slette et eksisterende program. Når programmøren vælger at oprette et ny program, fremkommer faciliteter for at kunne ændre styringen til robotten. Når programmøren er færdig med programmet, vælger han at gemme det. Hvis han vil redigere i et eksisterende program, vælger han programmet på listen, hvorefter programmet igen lister faciliteter for at ændre styringen til robotten. Når et program ønskes slettet, vælger programmøren et program på listen, og tilkendegiver overfor systemet, at

programmet ønskes slettet, hvorefter det fjernes.

Det fjerde normalforløb kræver ikke, at der er logget ind som programmør, da operatøren også kan tilgå de gemte programmer og anvende disse. Dette normalforløb initialiseres ved, at en af de to primære aktører tilkendegiver overfor systemet, at de vil anvende et program, hvorefter systemet lister tilgængelige programmer i systemet. Herefter vælges der, hvilket program der skal anvendes, og således anvendes det valgte program.

4.5.3 Use Case Undtagelser

Undtagelserne til de tre første normalforløb giver bl.a. programmøren en advarsel, hvis han annullerer oprettelsen af et program, uden at have gemt. I dette tilfælde modtager han en besked, om han er sikker på at han vil annullere handlingen. Hvis programmøren har glemt at navngive programmet, modtager han en besked om, programmet skal navngives. Hvis programmøren vil redigere i gamle programmer, og der ikke er nogle tilgængelige, er det blot standardprogrammet der bliver vist. Hvis dette program vælges, modtages der en besked om, at der ikke kan ændres i standardprogrammet. Hvis programmøren forsøger at slette et program, bliver han spurgt om dette ønskes, og hvis det er standardprogrammet der forsøges slettet, modtages der en besked om, at dette ikke kan lade sig gøre.

Hvis programmøren eller operatøren vil anvende et gammelt program, og der ikke er programmer tilgængelig, er det kun muligt at vælge standardprogrammet.

4.6 Use Case 3 scenarier

4.6.1 Use Case mål

Målet med denne Use Case er at tilføje en ny materialetype til systemet eller redigere i de eksisterende materialetyper. Dette betyder at det er muligt at slette et materiale, samt ændre navnet og densiteten på et eksisterende materiale.

4.6.2 Use Case scenarier

Denne Use Case har tre normalforløb, idet at man både kan tilføje en ny materialetype, ændre i en eksisterende eller slette en eksisterende materialetype. Det er kun programmøren, der kan initialisere denne Use Case.

Normalforløb 1 beskriver, hvordan en ny materialetype tilføjes. Dette foregår ved, at programmøren tilkendegiver overfor systemet, at han vil tilføje en ny materialetype. Herefter gør systemet det muligt at indtaste et navn og densitet for det nye materiale. Når programmøren har indtastet de ønskede værdier, vælger programmøren at gemme materialetypen, som gemmes så på databasen.

I normalforløb 2 fjernes en materialetype. Denne initieres ved, at programmøren tilkendegiver overfor systemet, at han vil fjerne en materialetype. Herefter listes alle tilgængelige materialetyper. Programmøren vælger så hvilken type der skal slettes, hvorefter systemet spørger på en bekræftelse af dette. Således fjernes materialetypen fra databasen.

Det sidste normalforløb fortæller hvordan der redigeres i en eksisterende materialetype. Her tilkendegiver programmøren overfor systemet, at han vil ændre i de eksisterende materialetyper, hvorefter der listes de eksisterende materialetyper. Herefter vælges hvilken type der ønskes redigeret, hvorefter densitet og navn kan indtastes. Til sidst gemmes den nye materialetype.

4.6.3 Use Case Undtagelser

Da databasen indgår i alle normalforløb, der der risiko for at forbindelsen mistes. Hvis dette er tilfældet, bliver programmøren spurgt, om han vil vente på at forbindelsen bliver genetableret, eller om programmet skal lukke ned. Vælges det at vente, bliver han igen præsenteret for samme besked, hvis programmet forbindelsen ikke kunne genetableres. Hvis det er lykkedes at genetablere forbindelsen, kører systemet videre.

Når der indtastes nye data valideres det, om densiteten er gyldig, og om alle data er indtastet. Hvis dette ikke er tilfældet, modtager programmøren en besked herom. Densiteten er ugyldig, hvis den er mindre end $0 \frac{g}{cm^3}$ eller større end $23 \frac{g}{cm^3}$.

Når der redigeres i et eksisterende materiale eller tilføjes et nyt, er det muligt at annulere handlingen, hvilket resultere i at ændringerne eller materialetypen ikke bliver gemt. Ligeledes slettes en materialetype selvfølgelig ikke, hvis handlingen annulleres.

Da der er 5 standardmaterialer i systemet, må disse ikke kunne slettes. Derfor modtages der en besked om, at det ikke er muligt at slette en af disse, hvis dette forsøges.

4.7 Use Case 4 scenarier

4.7.1 Use Case mål

Målet med denne Use Case er, at få vist en logfil, da denne indeholder informationer om et programs kørsel.

4.7.2 Use Case scenarier

Det er både operatøren og programmøren, der kan tilgå denne Use Case. Den initialiseres ved, at en af aktørerne tilkendegiver overfor systemet, han vil se gemte logfiler, hvorefter tilgængelige logfiler bliver listet. Herefter vælger aktøren hvilken logfil der skal vises, og således bliver data for denne logfil vist. Dernæst bliver logfilen gemt, ved at aktøren tilkendegiver overfor systemet, at filen ønskes gemt. Aktøren bliver dernæst præsenteret for et vindue, hvor placering for logfilen kan vælges, og hvor navnet på filen kan indtastes. Til sidst gemmes logfilen det ønskede sted, med det ønskede navn.

4.7.3 Use Case Undtagelser

Da databasen er sekundær aktør i Use Casen, og da logfiler hentes på databasen, er der risiko for, at forbindelsen til databasen mistes. Hvis dette er tilfældet, bliver aktøren spurgt, om han vil vente på at forbindelsen bliver genetableret, eller om programmet skal lukke ned. Vælges der at vente, bliver han igen præsenteret for samme besked, hvis forbindelsen ikke kunne genetableres. Hvis det er lykkedes at genetablere forbindelsen, kører systemet videre. Er der ingen gemte logfiler på databasen, er det ikke muligt at få vist data for en logfil.

Aktøren har mulighed for blot at få vist en logfil, uden at få denne gemt, og hvis dette er tilfældet, skal han blot lade være med at tilkendegive, at filen ønskes gemt. Hvis han fortryder at gemme filen, er det også muligt at annullere handlingen. Når navnet på filen skal indtastes, er det muligt at aktøren vælger et navn, der allerede eksistere i den givne sti. Hvis dette er tilfældet, bliver han spurgt om han vil overskrive den eksisterende fil. Hvis han ikke vil dette, får han mulighed for at indtaste et andet navn, der ikke er optaget.

4.8 Use Case 5 scenarier

4.8.1 Use Case mål

Målet med denne Use Case er, at give programmøren mulighed for at teste et givent program uden brug af den fysiske robot.

4.8.2 Use Case scenarier

Det eneste krav for at tilgå denne Use Case er, at brugeren skal være logget ind som programmør. For at initialisere Use Casen tilkendegiver programmøren overfor systemet, at robotten ønskes simuleret. Herefter lister systemet tilgængelige programmer i systemet, hvorefter programmøren vælger, hvilket program der skal indlæses. Herefter startes simuleringen, og der vises data fra simuleringen på skærmen på samme måde, som hvis robotten var tilsluttet. Sidst afsluttes simuleringen.

Når loggen udskrives, udskrives den med et tidspunkt for hændelsen efterfulgt af informationer om selve hændelsen.

4.8.3 Use Case Undtagelser

Den eneste undtagelse til denne Use Case består i at simuleringen afsluttes, hvis brugeren trykker på "Stop".

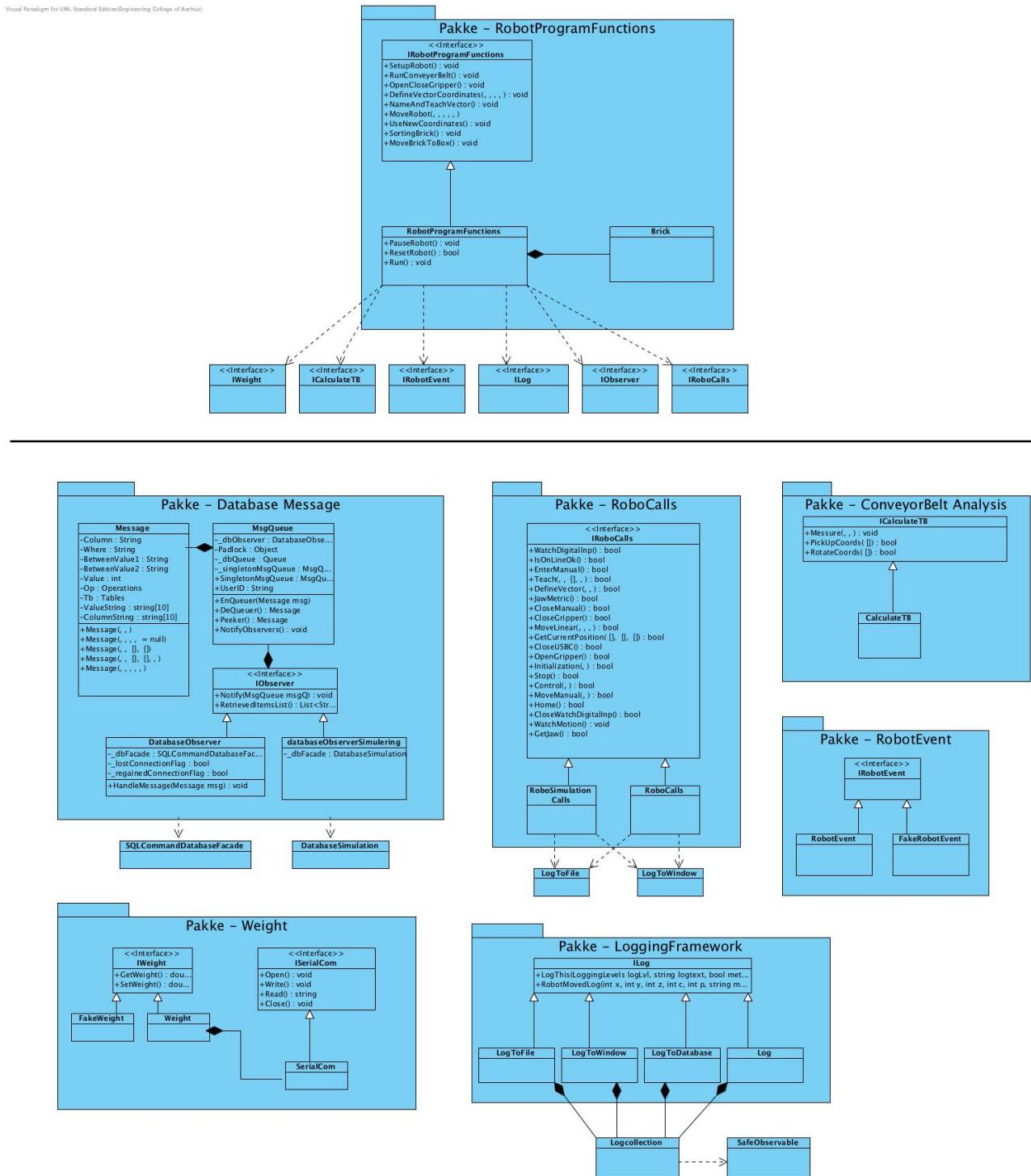
5 LOGISK VIEW

5.1 Oversigt

Systemet er bygget op efter 3-tier arkitekturen, med de 3 lag: Presentation layer, Business logic layer og Data acces layer, som også ses på figur 3 i afsnit 2.1 *Systemkontekst*. Derudover har vi et fjerde lag som er selve hardwarelaget. De følgende afsnit vil beskrive lagene i dybden. Det er også disse fire lag der ligger til grund for systemets pakker.

Herunder ses først systemets klasser opdelt i pakker (se afsnit 5.2 *Arkitektursignifikante designpakker for at se hvordan pakkerne spiller sammen i en lag-opdeling. Alle diagrammerne kan ses i stor format i Bilag: Diagrammer*):

5 LOGISK VIEW

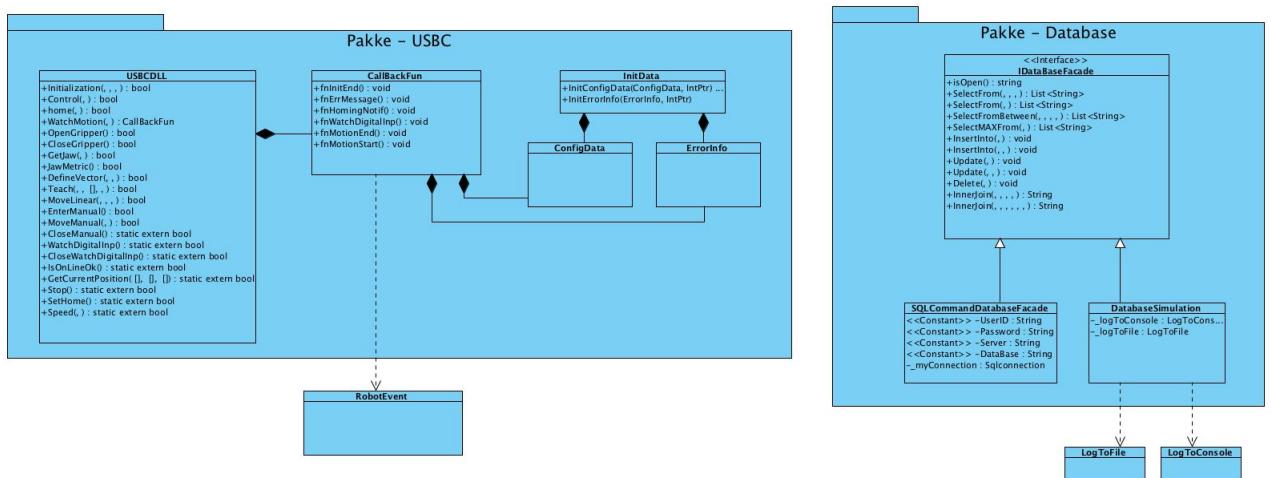
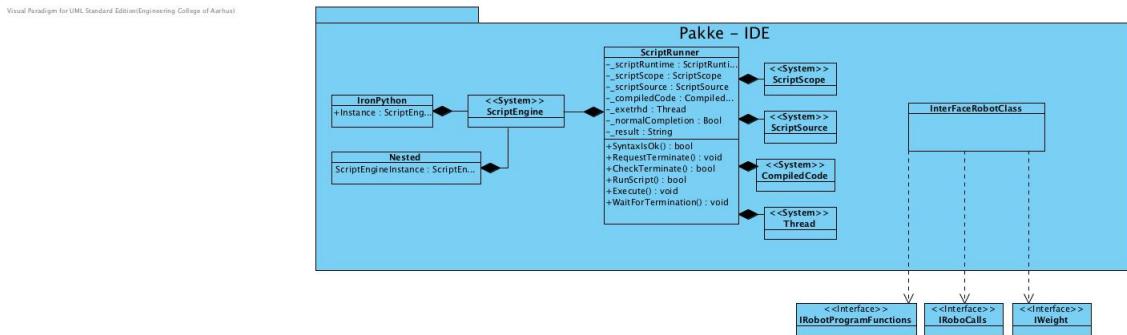


Figur 14: Hovedklasse med dets afhængigheder

Ovenstående figur viser hovedklassen (RobotProgramFunctions³, i øverste pakke), som indeholder funktioner der udgør vores standardprogram. Nedenunder ses alle dets afhæn-

³Se afsnit 8.2.1 Komponent 1: Sorteringsprogram - for nærmere information om denne klasse

gigheder⁴



Figur 15: Resterende klasser

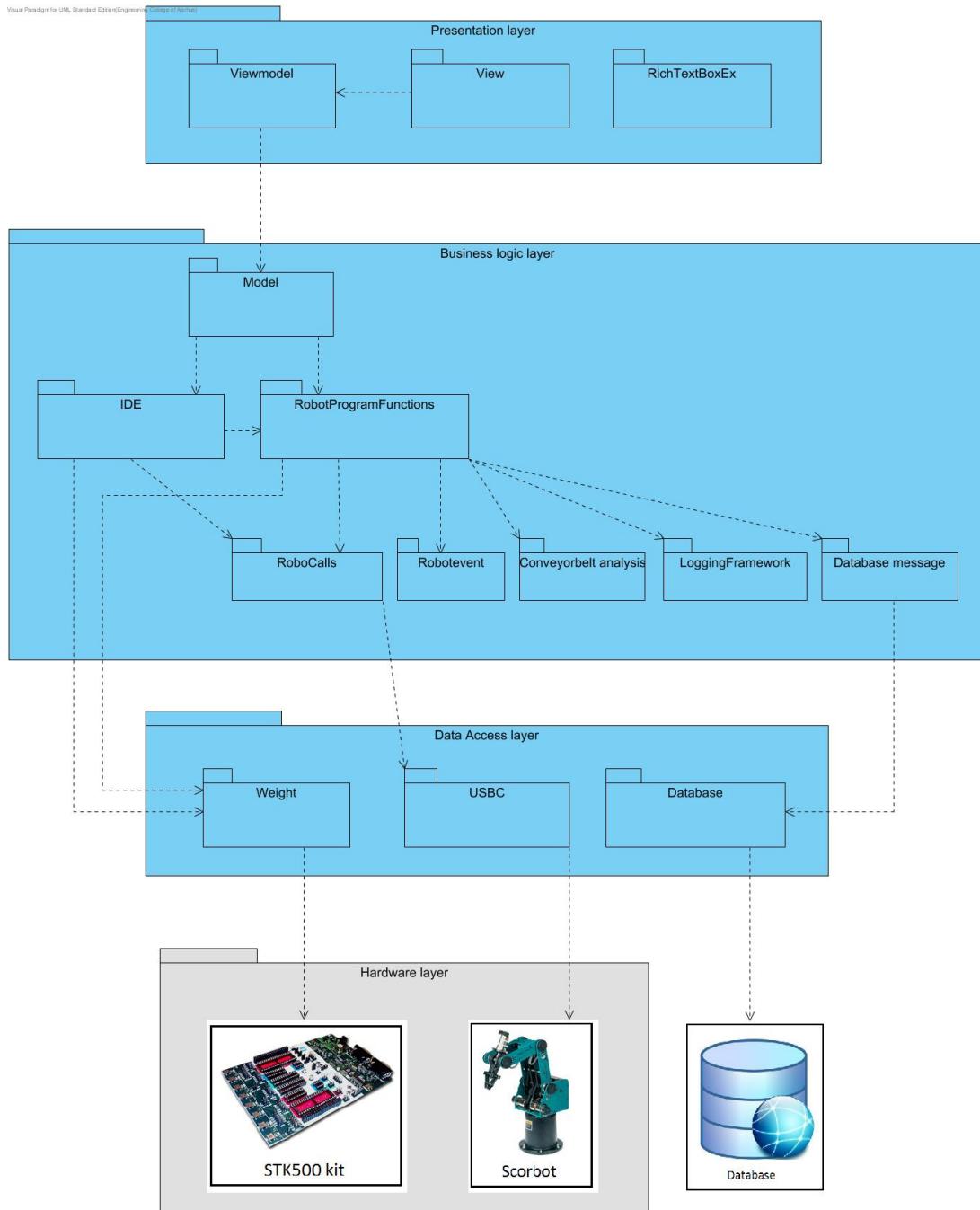
Ovenstående figur viser først IDE'en⁵, der er den anden klasse der kan tilføje funktionalitet til robotten. De to andre pakker, benyttes både af standardprogrammet og IDE'en.

5.2 Arkitektursignifikante designpakker

Som nævnt i forrige afsnit, er systemet opdelt efter 3-tier modellen. Herunder ses sammenhængen mellem pakkerne i forrige afsnit.

⁴ Afhængighederne bliver også brugt af andre klasser end RobotProgramFunctions.

⁵Se afsnit 8.2.8 Komponent8: IDE - for nærmere information om denne klasse



Figur 16: Lag-opdeling

View'et og Viewmodel pakken i presentation layer samt Model pakken i Business logic layer, udgør MVVM mønstret. Disse pakker består af flere filer hver især. Dette kan der læses mere om i afsnit *8.2.7 Komponent 7: GUI*

5.2.1 Pakke 1 - Presentation Layer

Denne pakkes ansvar er at vise en grafisk brugergrænseflade. Denne er baseret på MVVM og pakken indeholder derfor ud over viewet også en.viewmodel.

5.2.2 Pakke 2 - Business logic layer

Denne pakke indeholder selve programlogikken, og har til ansvar at håndterer businesslogikken. Pakken indeholder kun decideret højniveau software. Modellen ligger i denne pakke. Pakken spiller sammen med Presentation layer og Data access layer. Kontakten til Presentation layer sker via modellen til viewmodellen.

5.2.3 Pakke 3 - Data access layer

Denne pakke fungerer som et slags link mellem business logic layer og hardware layer. Den har til ansvar at hente data fra hardware og videreforsmide det til logikken højere oppe i systemet. Derudover har det også ansvar, for at muliggøre skrivning til hardwaren. Pakken indeholder et interface til microcontrolleren, interface til robotten samt interface til databasen.

5.2.4 Pakke 4 - Hardware layer

Denne pakke indeholder selve hardwaren, dll'er og drivere dertil. Udenfor hardwarelaget ligger selve databasen som fungerer som datakilde.

5.3 Use Case realiseringer

Dette afsnit beskriver hvorledes softwaren fungerer ved at beskrive hvorledes de udvalgte Use Cases er realiseret og hvorledes design elementerne bidrager til at opnå den funktionalitet, der er beskrevet i Use Casen. Her referenceret til de Use Cases og Use Case scenarier, der er beskrevet i forrige afsnit, Use Case viewet.

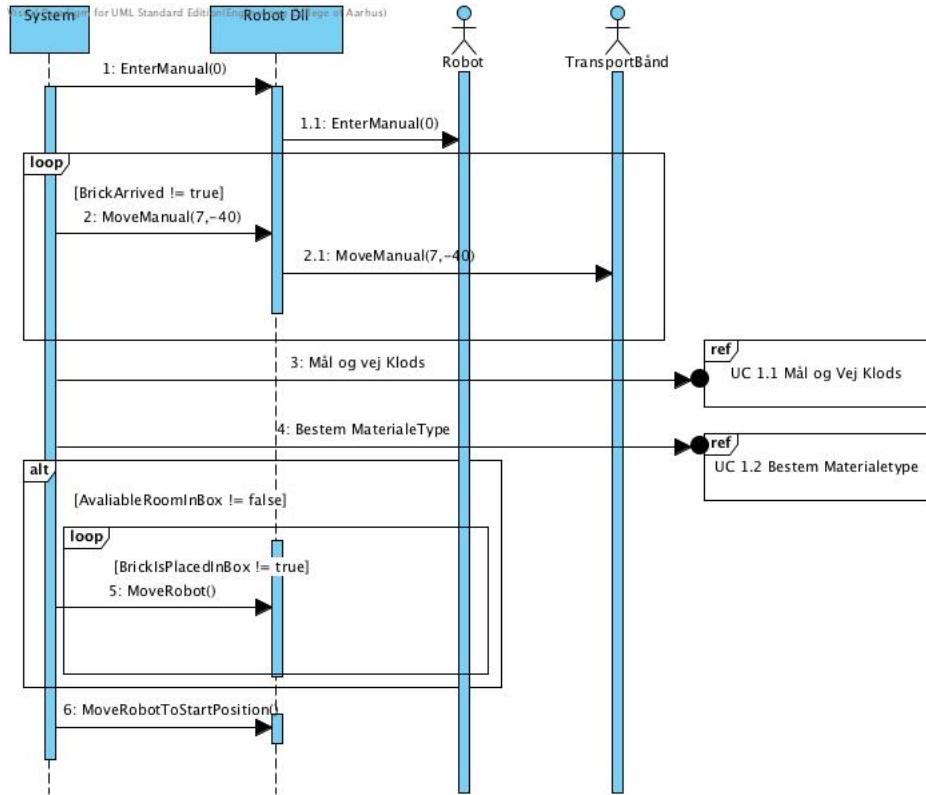
5.3.1 Use Case 1: Sorter klods realisering

Sorter klods Use Casen er realiseret ved at systemet starter med at kalde ned i Data Access Layer, som muliggør kommunikation med Robot Dll'en. Her benyttes funktionen

EnterManual(), hvor der indikeres at robotten skal styres via. akser og ikke koordinater. Herefter bliver der startet en while-løkke, som benytter funktionen MoveManual som har to parametre. Den ene sættes til at styre aksen, hvori transportbåndet er og den anden angiver hastigheden på transportbåndet. While-løkken er aktiv, indtil der er en klods der afbryder sensoren på transportbåndet. Når dette indtræffer vil en variabel blive ændret og programmet vil afbryde while-løkken.

Use Casen gør herefter brug af Use Case 1.1 og Use Case 1.2 som er beskrevet i afsnit 5.3.2 og 5.3.3. I disse include Use Cases bliver den givne klods målt og vejet, samt dens materialetype bestemmes.

Inden det givne materiale flyttes over til kassen, bliver der først valideret om der er plads til klodsen. Hvis dette ikke er tilfældet bliver klodsen afleveret på enden af transportbåndet og kørt ud over kanten. I tilfældet af der er plads til klodsen flyttes det givne materiale over til kassen, ved hjælp af funktionerne DefineVectorCoordinates, NameAndTeachVector samt MoveLinear, som findes i Robot Dll'en. DefineVector definere den givne vektor med koordinater og NameAndTeach navngiver den og indsætter vektoren i hukommelsen. MoveLinear bliver herefter brugt og den flytter robotarmen hen til den givne vektors koordinat. I implementeringen er disse 3 funktioner sat ind i en hjælpe funktion(MoveRobot()), som bliver kaldt flere gange i en sekvens for at få robotarmen til at lægge den givne klods det rigtige sted. Når klodsen er lagt på plads flyttes robotarmen hen til et absolut punkt som angiver startpositionen for robotarmen - Use casen er derefter udført og opfyldt.

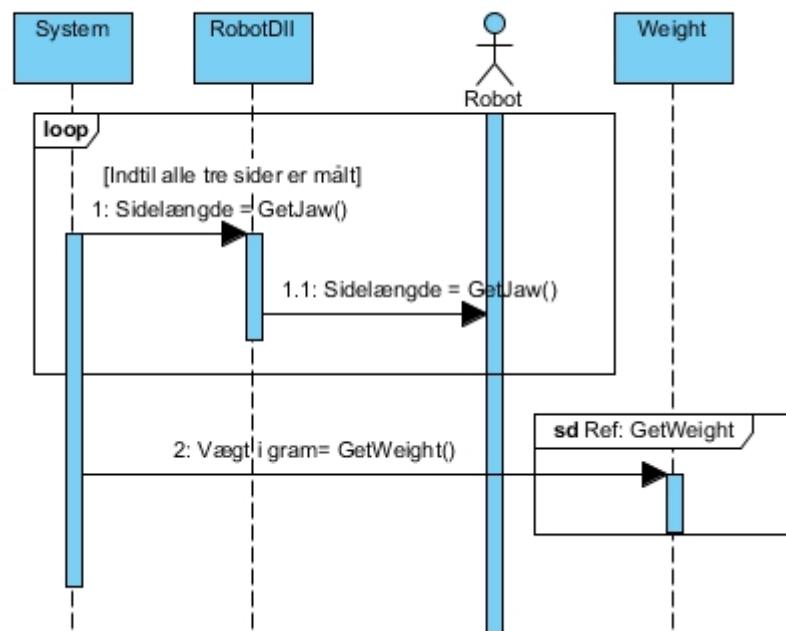


Figur 17: Simpelt sekvensdiagram over Sorter Klods

5.3.2 Use Case 1.1: Mål og vej klods realisering

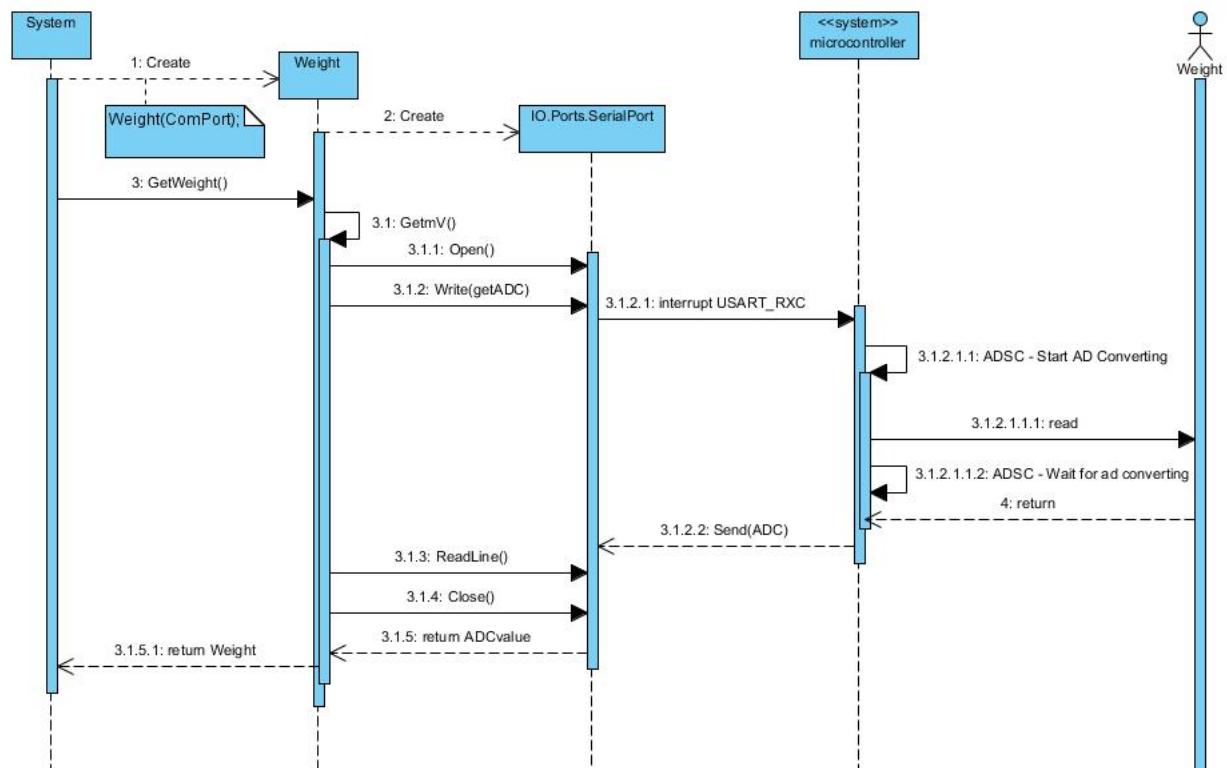
Mål og vej klods Use Casen er realiseret på sådan en måde, at systemet kaldet ned i Data access layer som muliggør kommunikation til robot dll'en. Her benytter den funktionen GetJaw(), hvorpå den får robotarmens klo aktuelle millimeter afstand. Denne funktion udføres tre gange for at få samtlige sider på kloden. Kloden placeres ydermere på vægten, hvor systemet henter klodens vægt via Data access layer. Denne gang er det GetWeight() funktionen der kaldes. Her opnås den nuværende vægt. GetWeight() fungere ved at sende et kald ned til microcontrolleren der får denne til at konvenerer den pågældende spænding fra strain-gauge-vægten til en ADC-værdi, der returneres op til systemet. ADC-værdien omregnes via lineær regression til en vægt målt i gram.

Interaktionsdiagram over "Mål og vej klods":



Figur 18: Simpelt sekvensdiagram af Mål og vej klods

Specifieret interaktionsdiagram af vægten:

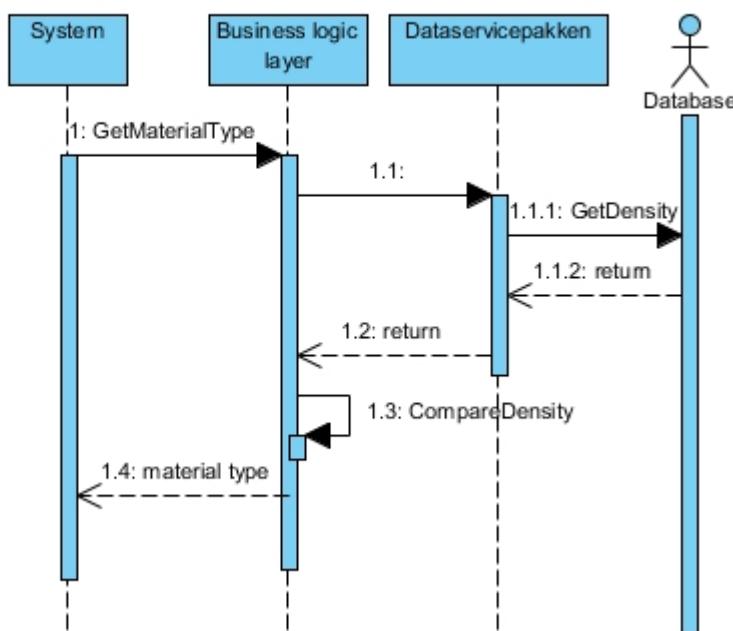


Figur 19: Detaljeret sekvensdiagram af vægten

5.3.3 Use Case 1.2: Bestem materialetype realisering

Use Case 1.2 "Bestem materialetype" er opbygget ved at system har noget data fra forrige Use Case (UC 1.1 - Mål og vej klods) der udregnes til en bestemt densitet i Business logic layer. Derefter kalder systemet ned i Data access layer, hvor kommunikationen til databasen findes. Her sammenlignes den udregnede densitet med den persistede densitet. Derudfra kan materialetypen bestemmes og Use Casen er udført samt udfyldt.

Interaktionsdiagram over "Bestem materialetype":



Figur 20: Simpelt sekvensdiagram af bestem materialetype

5.3.4 Use Case 2: Programmer robot realisering

Programmer robot Use Casen er blevet udfærdiget ved at lave en IDE, som benytter ironpython samt microsoft.scripting biblioteket. Denne Use Case bliver initialiseret ved at brugeren trykker på 'manage program' knappen fra menu vinduet. Herefter bliver brugeren taget til 'mange program' vinduet, hvor der er mulighed for at oprette, gemme, rediger, slette og indlæse programmer. Normalforløb 1, 'Opret nyt program' bliver initialiseret når programmøren begynder at skrive en ny robot sekvens i den richtextbox der er i vinduet, hvorefter programmøren kan navngive og gemme det skrevet program. Normalforløb 2, 'Rediger gammelt program' omhandler at kunne hente et gammel program, som programmøren har skrevet, og gemt, tidligere. Dette gøres ved at vælge et program, fra

listen over gemte programmer, og trykke på 'Load' knappen. Herefter vil sekvensen for det valgte program blive vist i richtextbox'en. Normalforløb 3, 'Slet program' omhandler at kunne slette et gemt program. Dette gøres ved at vælge et gemt program fra listen, og derefter trykke på 'Delete', dette vil slette det valgte program permanent. Normalforløb 4, 'Anvend program' kræver at programmøren, eller operatøren, er i 'Run window' vinduet, her kan brugeren vælge et af de gemte programmer fra en liste, hvorefter der trykkes på 'Run' knappen(robotten skal være homet, før dette er muligt). Dette vil få robotten til at udføre sekvensen fra det valgte program. Denne IDE er simpelt lavet, ved at gøre brug af ironpython, og microsoft.scripting biblioteket, som gør det muligt at oversætte python kode til C# kode, som kan bruges til at styre robotten. Dette forgår alt i Iron-Python klassen, som har en singleton scriptEngine med en Python engine. I det tilfælge at brugeren ønsker at køre et program på robotten via. 'Run window', bliver funktionen Script(string, Dictionary<string, object>) kaldt, hvor den første string parameter, er den python sekvens der står i den valgte fil, og Dictionary<string, object> parameteren som har et Dictionary med "Robot" og et object af klassen InterfaceRobotClass, som har alle de mulige robot realateret funktioner som brugeren kan kalde, via IDE'en. Ved dette kald bliver der lavet et script, dette script kan blive tjekket for syntaks fejl, ved at kalde SyntaxIsOK() som returnere en string med den første fejl den finde, eller "No Syntax errors", hvis ingen er fundet. SyntaxIsOk(), bliver også kaldt, når brugeren trykker på 'Compile' knappen, i 'programs window' vinduet. funktionen Execute(), eksekvere det script, som script() generere, forudsat at der ikke er nogen fejl i det. denne eksekvering forgår i sin egen tråd, det gør det muligt for brugeren at bruge de andre funktionaliteter i systems, imens robotten køre.

intellisense er også blevet implementeret i den richtextbox som programmøren skriver sin sekvens i, dette gør det muligt at få vist en liste over alle de mulige robot funktioner som er tilgængelige, for mere information om disse funktioner se *SBS_IDE-funktioner* dokumentet

5.3.5 Use Case 3: Rediger materialetype realisering

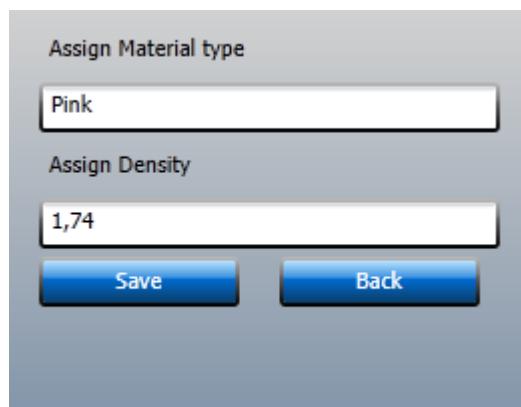
Denne Use Case bliver initialiseret ved, at programmøren på brugergrænsefladen trykker på 'Manage Materials'. Herefter kommer han ind et vindue, hvor der kan manipuleres

med materialetyperne. Billeder kan findes i bilag under Billeder



Figur 21: Materiale vinduet

Normalforløb 1, hvor en ny materialetype tilføjes, bliver så initialiseret ved, at programmøren trykker på 'Add', hvorefter en nyt vindue fremkommer, hvor navnet og densiteten på den nye materialetype kan indtastes.



Figur 22: Add eller Edit vinduet

Normalforløb 2 handler om sletning af materialetype, og det fungerer ved, at en materialetype markeres, hvorefter der trykkes på 'Delete', hvilket selvfølgelig sletter den valgte materialetype. Det er også muligt at redigere i eksisterende materialetyper, hvilket er det sidste normalforløb. Dette gøres ved at markere det ønskede materiale, efterfulgt af

et tryk på 'Edit'. Her fremkommer så et nyt vindue, hvor navnet og densiteten kan ændres.

Når vinduet åbnes hentes alle de gemte materialetyper på databasen, gennem kaldet til LoadMaterials funktionen. Denne funktion lægger så de hentede materialer ind i en observable collection, der kan indeholde MaterialsDataobjekter. MaterialsData klassen indeholder blot en række properties, og klassen bruges udelukkende til at indeholde data for en materialetype. I selve Viewlaget er der bindet til listen, således at materialetyperne kan ses på GUI'en. LoadMaterials efterspørger alle materialetyper i databasen. Dette gør den ved at oprette en besked, som håndteres i data access layer, som igen kalder videre ned til databasen efter informationer omkring alle gemte materialetyper. Disse returneres så til data access layer, hvorefter de kan tilgåes i modellaget, hvor de så ligges ind i en liste, så de kan tilgåes fra GUI'en.

Endvidere er der en række funktioner i modellaget som bruges til at redigere i de eksisterende materialetyper og oprette nye materialetyper. F.eks. er det muligt at slette en materialetype gennem kaldet til DeleteMaterial(). Funktionen skal have en enkelt parameter, som angiver ID'et på selve materialetypen. Det er en stor fordel, at materialetyperne er sorteret i listen, i samme orden som de er i databasen. De vises også i samme orden, som er i i databasen, og derfor kan indexet i listen på selve GUI'en medsendes som parameter, hvilket gør koden forholdsvis simpel og let at forstå.

Det kan diskuteres om det er nødvendigt at indlæse hele listen på ny, efter hver operation f.eks. ved tilføjelse af et materiale. Men ved at gøre dette, sikres det, at materialetyperne som programmøren kan se, altid stemmer overens med dem på databasen. Ligeledes resulterer det også i, at listen af materialetyper på skærmen bliver opdateret efter hver operation. Ligeledes er det ikke nødvendigt med ret meget kode på selve listen, da den hele tiden opdateres, når data hentes fra databasen.

Det skal nævnes, at der er et problem med implementeringen af Use Casen. Det er selvfølgelig muligt at tilføje materialetyper, men de vil i praksis aldrig kunne blive brugt. For det første er der kun 5 rum i boksen, der skal indeholde de sorterede klodser, så der er ikke plads til de nye materialetyper. Desuden er selve sorteringsalgoritmen laves således,

at den anvender nogle rimelig faste koordinater, og hvis en ny materialetype så skulle tilføjes, skulle der laves en ny algoritme til dette. Det er ligeledes heller ikke muligt at slette en af standardmaterialerne, da dette vil resultere i problemer på databasen. Hvis en af disse bliver slettet, vil en klods på databasen komme til at have en fremmednøgle til en materialetype som ikke eksisterer. Til gengæld er det muligt, at redigere i navnet og densiteten på de eksisterende materialetyper, hvis det ønskes.

5.3.6 Use Case 4: Tilgå log realisering

Fra start af blev der blevet bestemt at Loggen skulle gemmes på databasen, men da databasen og specielt databasefaden blev implementeret løbende, blev loggen i første omgang lavet som så den skrev til en konsol, og den kunne derfor tilgåes herigennem. Da databasen blev færdigudviklet blev der implementeret en version af logklassen, som gik igennem data access layer til databasen. Rent programteknisk foregår det ved at brugeren vælger en dato, hvorefter alle logbeskeder fra denne dato hentes fra databasen. Funktionaliteten med at gemme til en lokal fil på klient PC'en blev implementeret via .NET biblioteker og fildialogen i WPF, derfor kaldes det direkte i Business logic layer og Presentation layer.

5.3.7 Use Case 5: Test program realisering

Test program Use Casen, er blevet udfærdiget således at den fungerer som en simulator. Når en bruger vil starte programmet, skal han først vælge om han vil bruge den rigtige robot, eller en simulering af robotten. Hvis simuleringen vælges, bliver den klasse der har robotkaldene initialiseret med en simuleringsklasse af robotkaldene. På den måde bliver alle kald til robotfunktionerne simuleret, lige gyldigt om det er et brugerdefineret program eller standardprogrammet.

Kaldene til de robotfunktionerne, bliver simuleret således, at der bliver taget stilling til hvilke parametre der medsendes, og ud fra disse kan man se hvad mode robotten sættes til, om den rykker sig osv. Disse informationer logges så til vinduet, samt til en logfil der ligger lokalt.

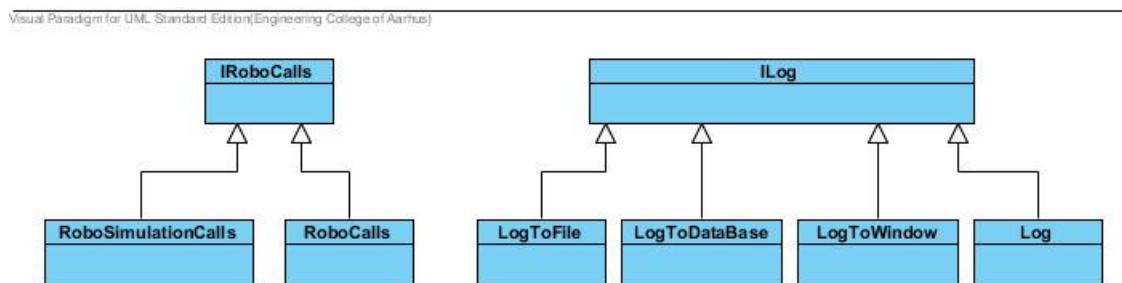
Når brugeren logger ind med en simuleringsrobot, indlæser systemet alle de programmer

6 PROCES/TASK VIEW

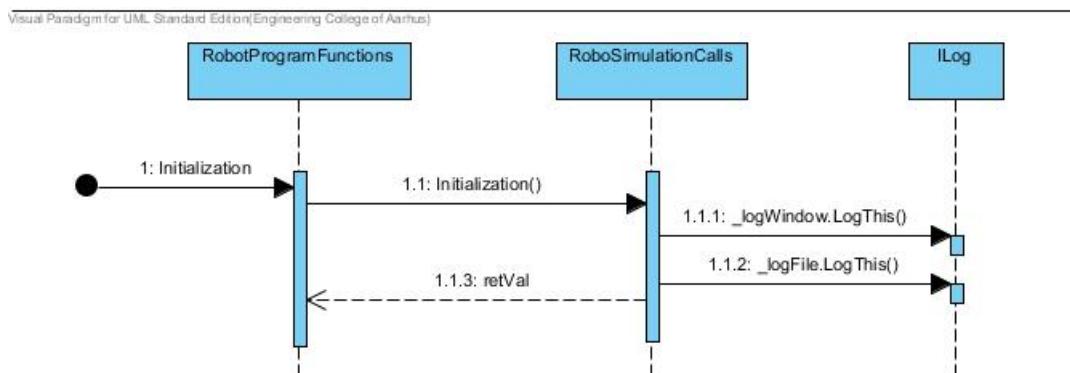
der er tilgængelige. Han har da mulighed for at køre dem, og se nøjagtigt hvad der sker. Han kan også pause, genstarte og stoppe en simulering. Derudover loader programmet også alle simuleringslogs ind i en boks, der opdateres hver gang en simulering er kørt, så bruger har mulighed for at nærlæse hvad der er sket.

Se afsnit 8.2.2 *Komponent 2: Simulering* for en detaljeret indsigt i simulatoren.

Herunder ses en overblik over hvorledes simulatoren er opbygget:



Figur 23: Nedarvning fra ILOG og IRoboCalls



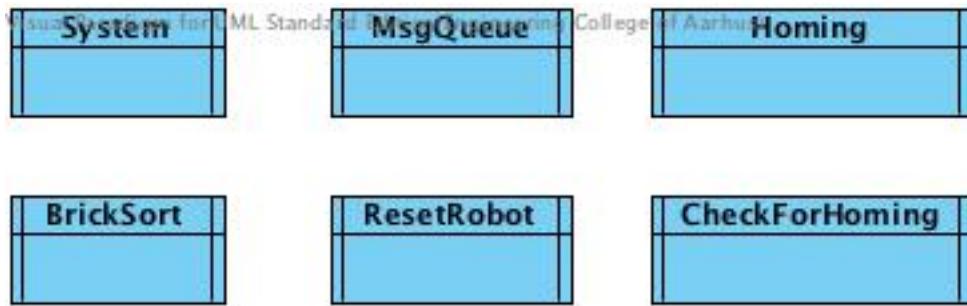
Figur 24: Simulering Detaljeret-sekvensdiagram

6 PROCES/TASK VIEW

I dette afsnit bliver der beskrevet systemets opdeling af tråde, samt hvorledes de kommunikere. Diagrammer kan finde i stort format under bilag: Diagrammer/sekvensdiagrammer/Process Task view

6.1 Oversigt over processer/task

Processen på PC'en består samlet set af 6 tråde.



Figur 25: Oversigt over tråde

System - Hovedtråd som starter og initialiserer robotten. Tråden sørger også for brugergrænsefladen.

MsgQueue - Opretter en besked kø som bruges til at sende eller modtage beskeder til og fra databasen.

Homing - Tråd som gør at robotten kan home igennem brugergrænsefladen.

BrickSort - Tråd der sørger for at sortere de forskellige klodser.

ResetRobot - Tråd der gør det muligt at stoppe programmet i en given sekvens igennem brugergrænsefladen, hvorefter den returnere til startpositionen.

CheckForHoming - Tråd der tjekker hvornår robotten er homet.

6.2 Proces/task kommunikation og synkronisering

Kommunikation

Trådene System og MsgQueue kommunikerer via et publish/subscriber relationship. Overordnet set betyder det, at klassen MsgQueue er publisher og databaseobserveren er subscriberen på indkommende beskeder fra MsgQueue.

Trådene System og Homing samt System og BrickSort kommunikerer ved, at systemet ser på givne værdier for nogen medlemsvariabler, og hvis disse er opfyldt bliver trådene eksekveret.

Trådene System og CheckForHoming bliver kørt i forbindelse med, at Homing tråden er aktiv. CheckForHoming venter på at en medlemsvariabel er blevet ændret i Homing tråden for derefter at ændre noget tekst i brugergrænsefladen.

Trådene System og ResetRobot kommunikere ved at brugeren trykker på en knap på brugergrænsefladen, mens brugeren er ved at sortere en klods, hvilket betyder at tråden BrickSort er igangsat. Tråden BrickSort vil blive stoppet og robotarmen vil vende tilbage til startpositionen.

Synkronisering

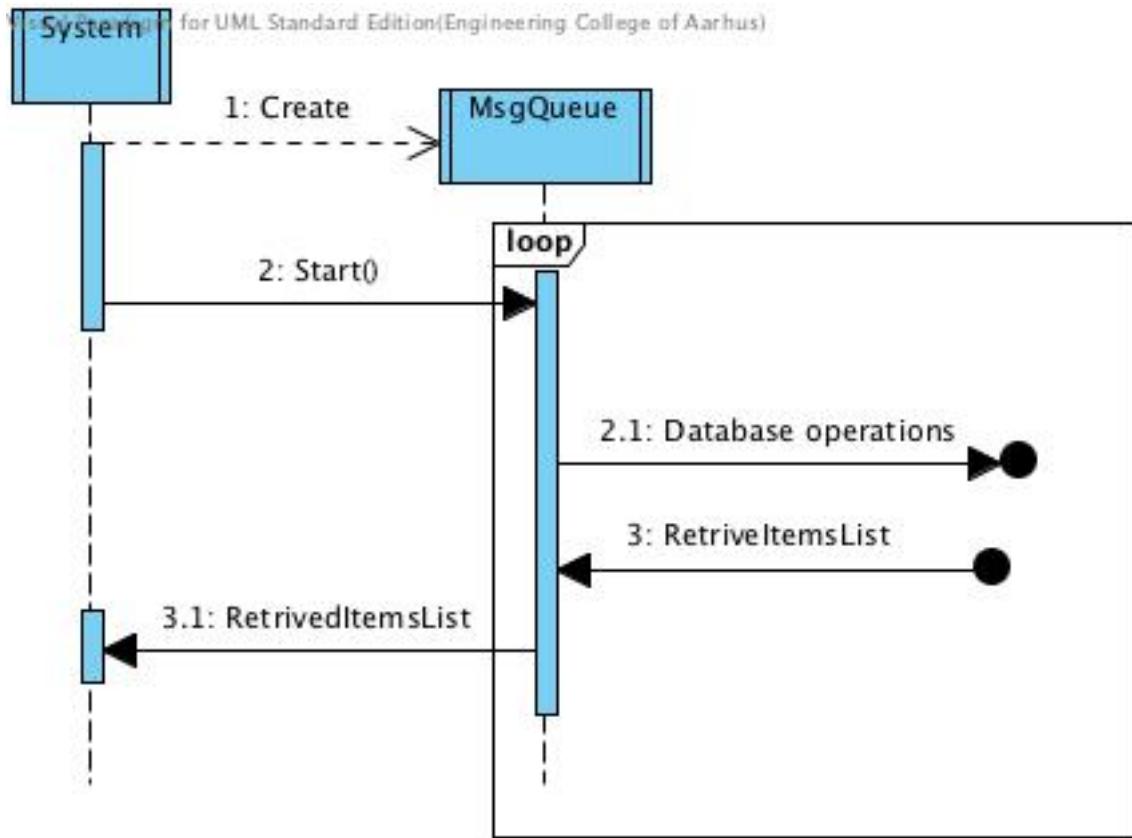
Trådene System og MsgQueue har behov for at blive synkroniseret. Indsætning af beskeder i køen kan gøres lige så ofte, som brugeren har behov for, men i tilfælde af, at der er en besked, der kræver hentning fra databasen, er der behov for synkronisering. System tråden må ikke fortsætte, så længe der er en sådan besked i køen. Derfor er der blevet gjort brug af en semafor. Denne semafor frigives når data er hentet fra databasen og lagt over i RetrievedItemsList. Når beskeden er håndteret må systemet fortsætte. Semaforen kan tilgås med to funktioner, WaitSem og ReleaseSem, fra DatabaseObserver klassen. I tilfælde af, at to beskeder, der skal hente fra databasen, bliver lagt i køen samtidig fra to forskellige tråde, er det nødvendigt at lade den ene tage semaforen, og lade den anden vente i et loop indtil semaforen er frigivet igen. Hvis der ikke tages højde for dette, og der er to tråde der venter på samme tid, er det ikke sikkert, hvilken en af dem, der bliver frigivet når Release bliver kaldt på semaforen.

6.3 Procesgruppe 1

Hele systemet er bygget op af én process med seks tråde, som består af dem beskrevet i afsnit 6.1 Oversigt over processer/task. Disse seks tråde vil i de følgende afsnit bliver beskrevet som Processgruppe 1. For overskuelighedens skyld er diagrammerne lavet forsimplet, dvs. at der ikke er navne på alle de klasser de går igennem, men mere hvordan trådene bliver oprettet og kommunikere. Nedenunder hver diagram er der nærmere beskrevet, hvordan dette foregår.

6.3.1 Proceskommunikation i gruppe 1

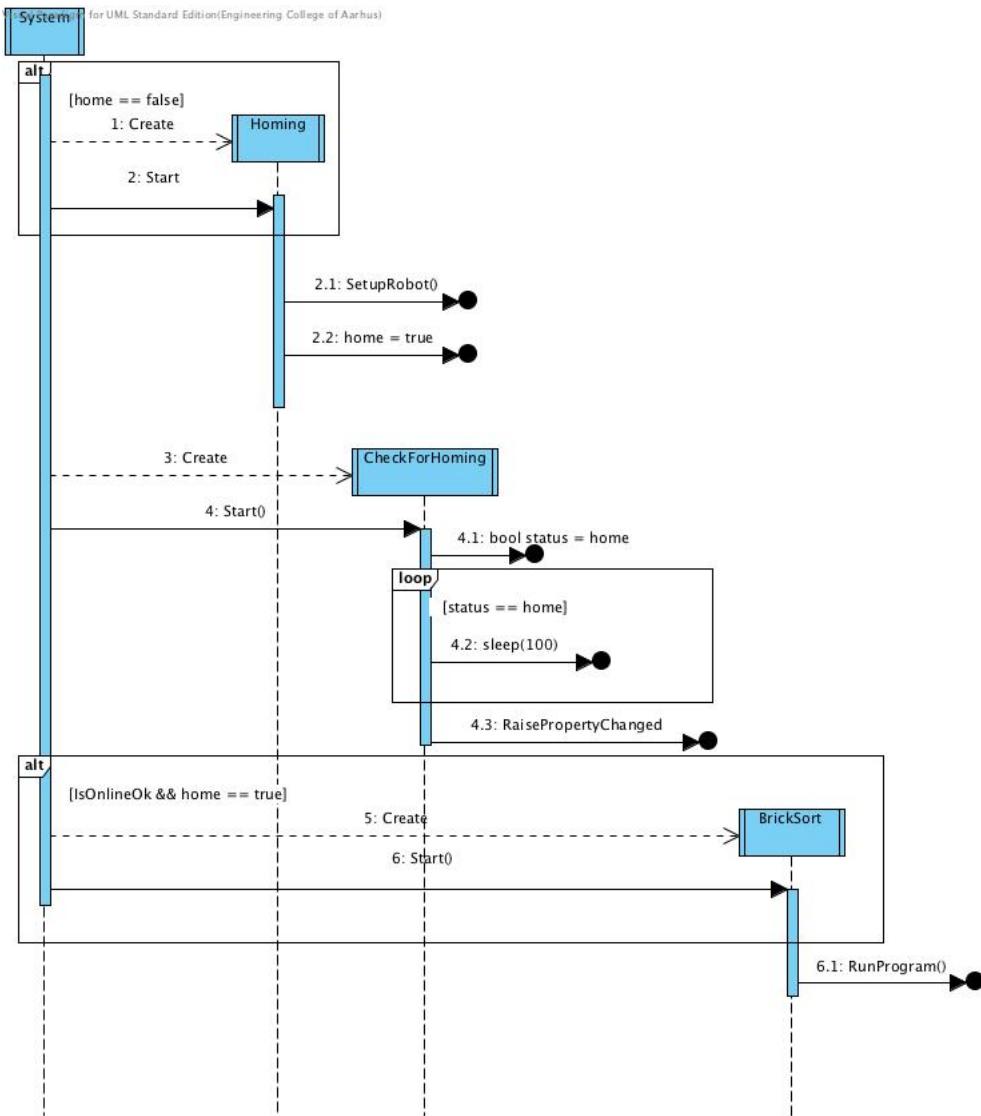
På figur 26 ses hvordan system tråden skaber MsgQueue tråden, og hvad den skabte tråd laver i overordnede træk.



Figur 26: Sekvensdiagram over - System og MsgQueue

Denne tråd bliver skabt når systemet startes op, i klassen model. Det er en baggrundstråd som kører konstant ved siden af system tråden og bliver nedlagt når systemet lukkes ned. Trådens opgaver består i at håndtere og vedligeholde database tilgangen, så systemet ikke har behov for at stoppe op hver gang der skal ske en database tilgang. Tråden samler beskeder fra hele systemet og håndterer dem parallelt med system tråden. Den håndterer også synkronisering, sådan at hovedtråden stille venter, hvis der er behov for den skal vente. Dette sker når der er behov for at hente fra databasen, hvilket gør at tråden starter sin kritiske sektion, som skal håndteres hvis systemet skal fortsætte. Dette *skal* håndteres da det er nødvendigt for systemet at bruge dataen hentet fra databasen. Der kan læses mere om dette i afsnit 6.3 Kommunikation og synkroniseringen.

På figur 27 ses overordnet på, hvordan system tråden skaber de 3 tråde Homing, CheckForHoming og Bricksort samt hvad de skabte tråde laver.



Figur 27: Sekvensdiagram over System og Homing,CheckForHoming,BrickSort

System tråden med brugergrænsefladen er aktiv og igennem dens kørsel kan det ske, at der trykkes på en knap der signalere at robotten skal home. Når dette sker, bliver der set på om 'home' variablen er false, hvilket betyder at den ikke har været homet. Derefter opretter systemet tråden 'Homing' som går ind og kalder en funktion der hedder `SetupRobot()` som gør robotten klar, dvs. robotten bliver initialiseret, samt home. Home variablen sættes derefter til true for at signalere at robotten nu er homet.

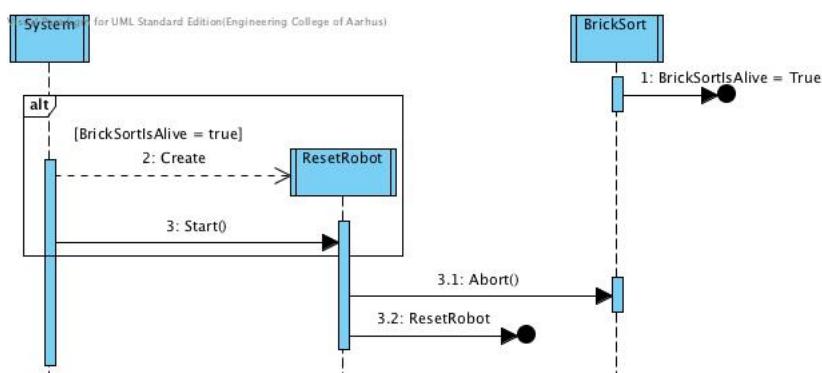
Systemet opretter herefter tråden 'CheckForHoming' som i grove træk har en while løkke der undersøger home variablen. I diagrammet kan det ses at der bliver oprettet en bool variabel `status` på home variablen er. Når programmet starter er den sat til false og det vil den også være i starten når 'CheckForHoming' kører. Når 'Homing' tråden er færdig ændres

home variablen og 'CheckForHoming' tråden sender derefter en RaisePropertyChanged som sender en besked at en property er blevet ændret og skal opdatere sig derefter. I dette tilfælde er det en tekstboks som skifter farve samt tekst.

Systemet ser derefter på om variablen IsOnlineOk, samt Home har værdien true. IsOnlineOK er en variabel der holder styr på om robotten er online. Når begge værdier er true opretter systemet 'BrickSort' tråden som skal styre sorteringsalgoritmen. RunProgram() bliver herefter kaldt og den tjekker på om den skal kører standard sorteringsprogrammet eller om den skal køre et brugerdefineret program.

På figur 28 ses overordnet hvordan system tråden skaber ResetRobot tråden og hvordan de kommunikerer.

Figur 28: Sekvensdiagram over System og ResetRobot



System tråden med brugergrænsefladen er aktiv og der bliver set på, hvilken værdi variablen `BrickSortIsAlive` har. Når den ændrer værdi til true betyder det at 'BrickSort' tråden er igangsat. Dette betyder at 'ResetRobot' tråden bliver lavet, da der så er mulig for at stoppe robotten igennem brugergrænsefladen. Når robotten bliver sat til at stoppe bliver 'BrickSort' tråden afbrudt og funktionen `ResetRobot()` bliver kaldt, som går ind og returnerer robotten til sin start position.

7 DEPLOYMENT VIEW

Systemet indeholder grundlæggende kun 2 processorer: AT mega16 microcontrolleren, og den PC hvor programmet afvikles. Derudover findes desuden en database der bruges til persistent lagring. Diagrammer kan findes i bilag: Diagrammer/Generelt

7.1 Oversigt over systemkonfigureringer

Systemet indeholder kun en egentlig konfiguration, som er vist i de følgende afsnit. Det er selvfølgelig muligt at udskifte Client PC med en anden maskine, som overholder minimumskravene, ligesom det vil være muligt at benytte fx en ATmega32 som microcontroller.

7.2 Node-beskivelser

7.2.1 Node 1. beskrivelse - Client PC

På denne enhed kørers SilverBulletSort.exe som indeholder systemets primære funktioner, som sortering af klodser, IDE og simulator. Som minimum skal der være tale om en 32-bit windowsmaskine. Helst XP-baseret, selvom også windows 7 32-bit understøttes i nogen grad. Der er ikke store performancekrav til enheden, men for at opleve et flydende programflow anbefales en processor med mindst 1 GHz clock-frekvens. Deuden kræves omkring 40 MB fri harddiskplads.

7.2.2 Node 2 beskrivelse - ATmega16 microcontroller

Denne enhed er en simpel C-baseret microcontroller.

Enhedens specifikationer er som følger:

- 8 bit
- op til 16 MHz clockfrekvens.
- 16 kbytes flash hukommelse
- 512 bytes EEPROM

7.2.3 Node 3 beskrivelse - Relationel database

Denne enhed eksisterer på serveren webhotel10. Den supporter MS SQL 2003, og ligger på et Windows NT styresystem. For at logge ind på databasen kræver det følgende oplysninger:

- **Server type:** Database Engine
- **Server:** webhotel10.ihb.dk
- **Login:** F12I4PRJ4Gr5
- **Password:** F12I4PRJ4Gr5

Databasen er implementeret i MS SQL 2003 ved brug af Microsoft SQL Server Management Studio

8 IMPLEMENTERINGS VIEW

8.1 Oversigt

Dette afsnit beskriver den endelige implementeringsopdeling af softwaren i lagdelte delsystemer. Dette view specificerer opdelingen i det logiske. Alle bilag findes under Diagrammer og Billeder i fuld størrelse.

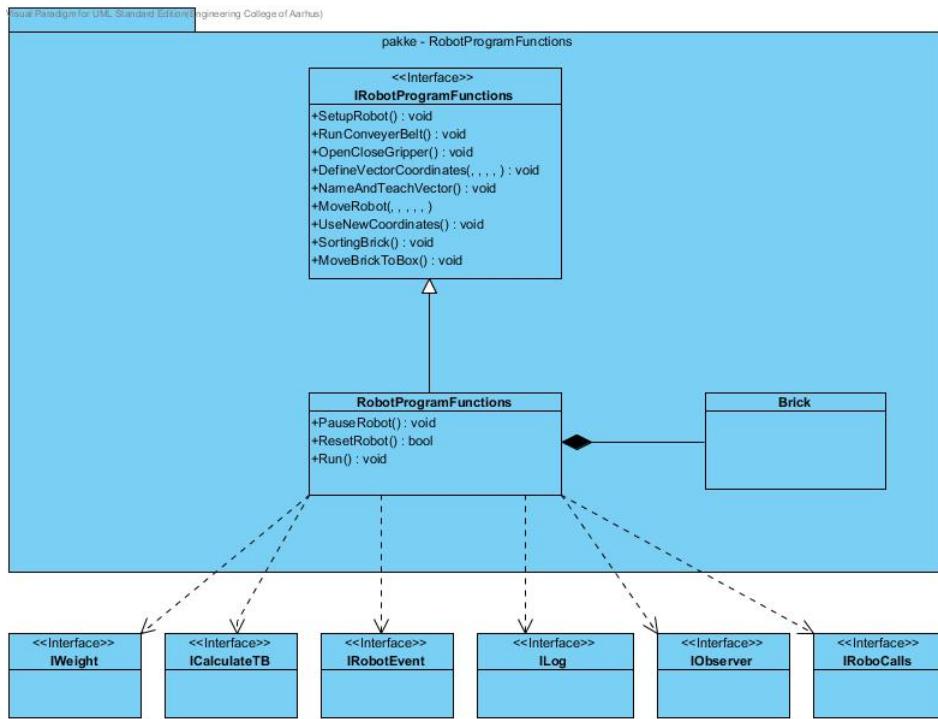
8.2 Komponentbeskrivelser

8.2.1 Komponent 1: Sorteringsprogram

Denne komponent indeholder funktioner, der udformer hovedprogrammet til at sortere klodserne. Denne komponent består hovedsageligt én klasse (RobotProgramFunctions). Da det er hovedfunktionen til sortering af klodserne der ligger i denne klasse, har klassen også mange afhængigheder til andre komponenter i systemet.

Design

Som nævnt har RobotProgramFunctions mange afhængigheder til andre komponenter i systemet. Herunder ses et diagram, der giver et overblik over hvorledes afhængighederne ser ud (se det fulde overblik i *afsnit 5: LOGISK VIEW*).



Figur 29: Klassediagram - RobotProgramFunctions

De seks afhængigheder, der går ud af pakken, er interfaces, som initieres med constructor injection. På den måde kan man skifte mellem rigtige objekter og fakeobjekter beregnet til simulering (se afsnit "7.2.2 Komponent 2: Simulering"). 'IRoboCalls' afhængigheden er den der styrer, om programmet bliver kørt med rigtige funktioner fra USBC dll'en, eller om programmet simuleres med "falske" funktionskald. Derudover har klassen et objekt af 'Brick', hvor information om klodserne gemmes midlertidigt.

RobotEvent

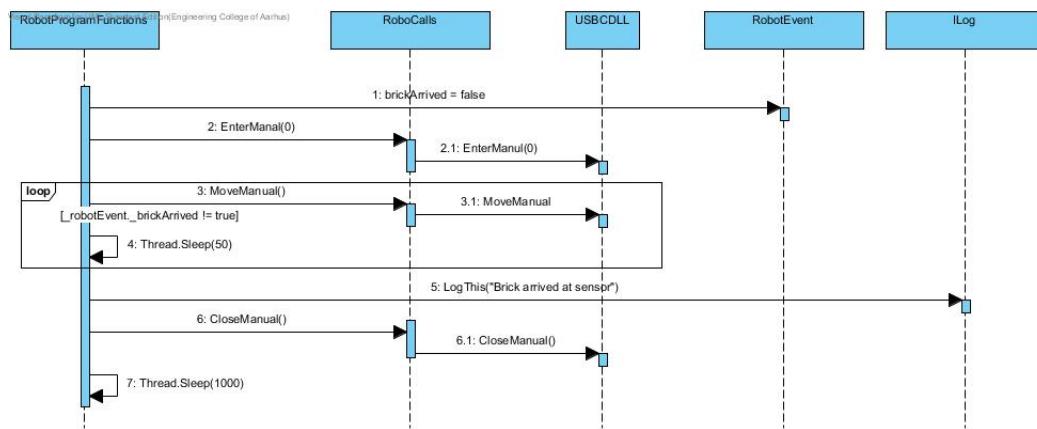
Før de forskellige funktioner dokumenteres, kræves en smule viden om klassen RobotEvent. Funktionerne i RobotProgramFunctions benytter klassen RobotEvent, til at tjekke om der er sket nogle events med robotten, såsom at en klods er ankommet til sensoren. Klassen indeholder tre bools, der bliver sat i de forskellige callbackfunktioner, der tilhører funktioner fra USBC dll'en (se afsnit 2.4.1 *USBC.dll*, for detaljeret dokumentation for USBC funktionskald). RobotEvent er en singleton, da det er vigtigt at der kun oprettes et objekt af denne klasse, da der jo kun er en robot, og eventene skal svarer til denne.

Funktioner

Herunder findes beskrivelse af funktionerne i RobotProgramFunctions.

RunConveyerBelt()

Herunder ses et lille sekvensdiagram over forløbet i funktionen:

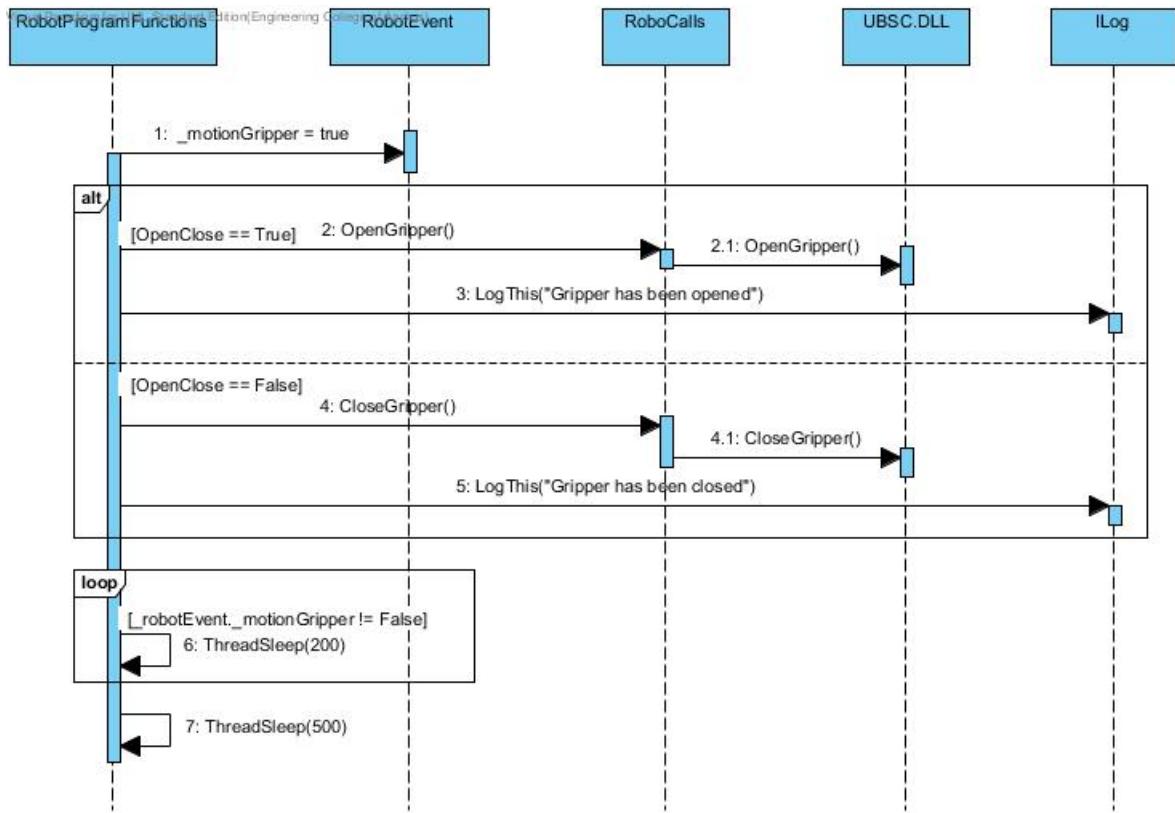


Figur 30: Sekvensdiagram: RunConveyerBelt

Når denne funktion kaldes, starter den med at sætte en bool til false i RobotEvent, som betyder at der ikke er nogen klods foran sensoren. Derefter går funktionen ind i en while-løkke, der tjekker på RobotEvent boolen. Inde i denne sættes transportbåndet i bevægelse (det køre kun i en begrænset periode), hvorefter der er en sleep, der sørger for den ikke står og tager hele CPU'en. Denne bool bliver sat via en callbackfunktion fra funktionen *WatchDigitalInp* i *USBC.dll*, der bliver kaldt, når sensorværdien ændrer sig (se *Se bilag: DLL filer/USBC.dll* for information om *WatchDigitalInp*). Dermed et det vigtigt at *WatchDigitalInp* er blevet kaldt, hvilket også bliver gjort i funktion *SetupRobot()*. Når den går ud af while-løkken, stoppes transportbåndet.

OpenCloseGripper(bool openClose)

Herunder ses et lille sekvensdiagram over forløbet i funktionen:



Figur 31: Sekvensdiagram: OpenCloseGripper

Parameteren 'openClose' bestemmer om gripperen skal åbne eller lukke. Funktionen starter dog med at sætte en bool til true i RobotEvent, der siger at robotten er i bevægelse. Derefter åbnes eller lukkes gripperen via en USBC funktion. Funktioner går, ligesom RunConveyerBelt(), ind i en while-løkke med en kort sleep, for at skåne CPU'en, indtil boolen i RobotEvent ændrer sig. Denne bool bliver sat via en callbackfunktion fra funktionen WatchMotion i USBC.dll, der bliver kaldt, når robotten ikke bevæger sig mere (se *bilag DLL Filer/USBC.dll* for information om WatchMotion). Dermed et det vigtigt at WatchMotion bliver kaldt.

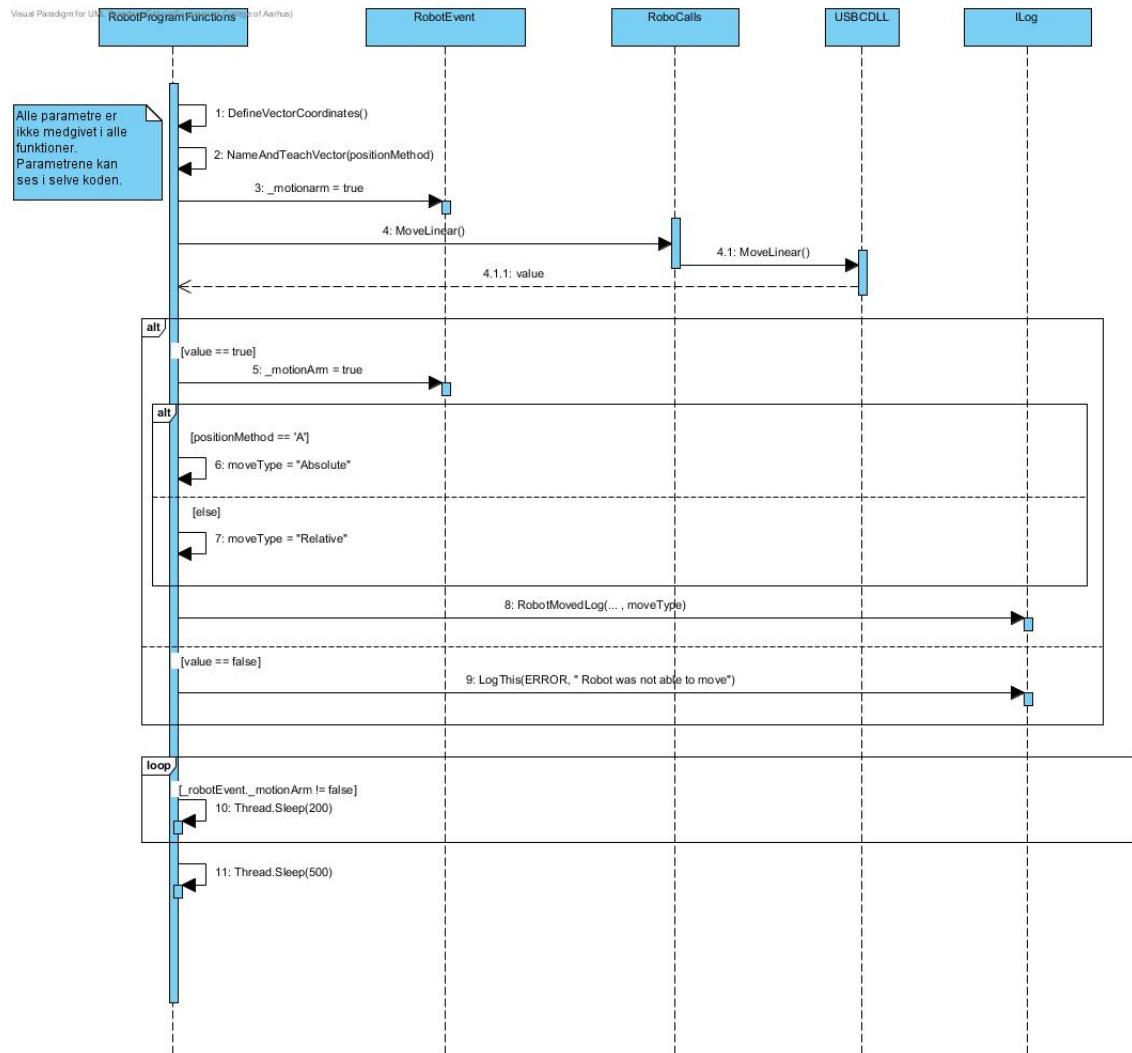
NameAndTeach Vector(char positionMethod)

Funktionen starter med at sætte et vektornavn, hvorefter det inkrementeres således, at de forskellige vektorer der oprettes, får forskellige navne. Vektoren bliver derefter oprettet med navnet samt to andre parameter som er konstante og definerede til en given værdi. Funktionen returnerer true hvis det blev oprettet, samt false hvis det ikke lykkedes. Parameteren positionMethod bliver valideret om det er et 'R', dette betyder at vektor

koordinaterne får en relativ placering. Alt andet end 'R' sætter det til absolut placering. Herefter bliver teach kaldt, som indsætter koordinaterne i den givne vektor og returnerer true, hvis det sker uden fejl.

MoveRobot(Int32 x, Int32 y, Int32 z, Int32 p, Int32 r, char positionMethod)

Herunder ses et lille sekvensdiagram over forløbet i funktionen:



Figur 32: Sekvensdiagram: MoveRobot

Funktionen starter med at bruge de 5 første parameter til funktionen `DefineVektorCoordinates()`. Denne funktion tager de 5 parametere som den indsætter i et array, hvori hver parameter bliver ganget med 1000, således, at det passer med robottens koordinatsæt. Derefter bliver `NameAndTeachVector` kaldt med den sidste parameter. `MoveLinear` bliver herefter kaldt, og det bliver valideret, om det lykkedes at kalde den med vektornav-

net samt, at den er rykket til de koordinater, der er medgivet. I tilfælde af at det lykkes, startes der med at blive sat en bool, som angiver at robotten er i bevægelse. Dette bliver derefter logget til databasen, hvis det er det rigtige program og logget til en fil, hvis det er simulering. En while løkke ser herefter på boolen, der blev sat tidligere, om den har ændret sig til false, dette sker gennem en callbackfunktion som ser på om robotten er i bevægelse. Når den ikke længere er i bevægelse bliver den sat til false og funktionen slutter.

Sorteringsalgoritme

I sorteringsalgoritmen for at sortere en klods, bliver der lavet mange kald til MoveRobot. MoveRobot har flere parametre, hvor en af dem angiver om det er relativ eller absolut placering. For at gøre algoritmen nemmere at ændre i, er der lavet hovedpunkter med absolut placering. Der er lavet et absolut punkt til startpositionen, som er lige over sensoren for transportbåndet og et andet som er lige over vægten. Endvidere er der lavet forskellige absolutte punkter til de forskellige rum i kassen, hvor klodserne bliver sorteret. Ud fra disse absolutte punkter er der så lavet forskellige sekvenser, som er lavet med relative punkter, som tager udgangspunkt i det sidst satte koordinatsæt og flytter sig derefter. Det har medført, at det har været nemmere at lave justeringer til at lave den givne sekvens helt præcis. Der er lavet funktionalitet, så det er muligt at flytte transportbåndet, som er dybere beskrevet i afsnittet: *8.2.3 Komponent 3: Bestem placering af Transportbånd* , og dette har gjort det nemmere med relative placeringer, da der kun skulle ændres på x og y koordinaterne i det hovedpunkt, som var vores startposition over transportbåndet. Funktionernes sekvensdiagrammer kan findes i bilag

SortingBrick()

Som det første i denne funktion kaldes WatchMotion (se begrundelse i ovenstående *Open-CloseGripper(bool openClose)*). Herefter åbnes gripperen og flyttes til en startposition over sensoren via MoveRobot (se *MoveRobot(Int32 x, Int32 y, Int32 z, Int32 p, Int32 r, char positionMethod)*) i det overstående for nærmere information). Herfra køres en sekvens af relationelle MoveRobot kald, der søger for, at gripperen får greb om alle siderne på en klods, samt at den kommer over på vægten (se *Absolut eller relativ placering* i det oven-

st  ende, for beskrivelse af hvordan bev  gelsessekvensen er opbygget). I denne sekvens, m  les de forskellige sider, og gemmes i den n  vnte Brick klasse. Til sidst i sekvensen flyttes robotarmen tilbage til startpositionen, med klodsen i gripperen. hereftere bliver en funktionen MoveBrickToBox kaldt, der søger for at ligge klodsen i det rigtige rum (se *MoveBrickToBox()* i det nedst  ende).

MoveBrickToBox()

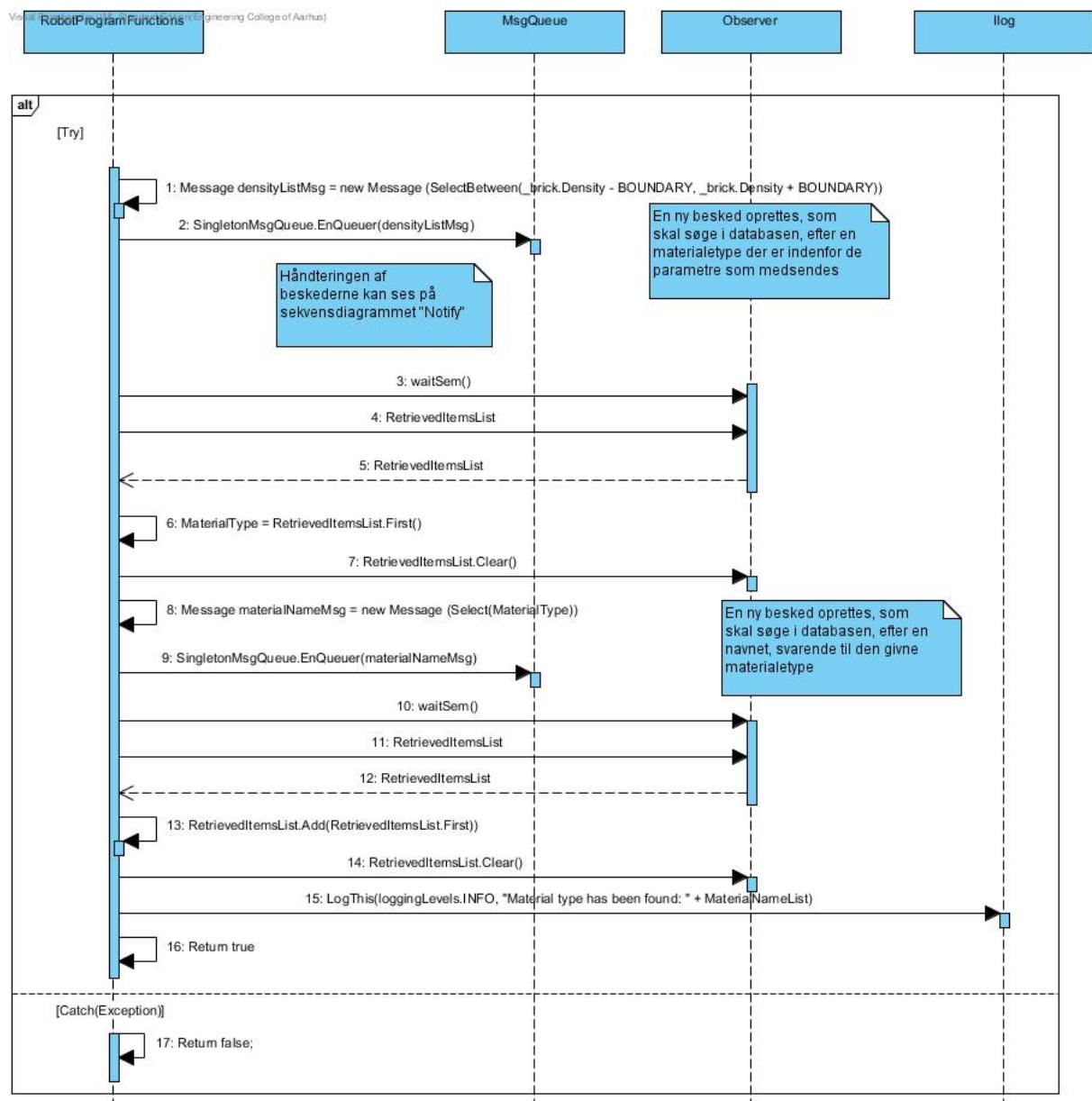
Det f  rste der sker i denne funktion er, at materialetypen og materialenavnet findes, gennem kaldet til *GetMaterialType()*. Denne funktion diskutes yderligere i nedenst  ende afsnit. Hvis det er muligt at finde frem til materialetypen for den givne klod, hentes samtlige l  ngder for alle klodser der er persisteret i systemet, med den fundne materialetype. Dette skal bruges til at finde ud af, hvor langt klodsen skal skubbes frem i boksen. Dette sker gennem kaldet til *GetTotalBrickLength()*. Denne funktion opretter en ny besked, med parametre der indikerer, at det er l  ngden for klodser med den givne materialetype der skal hentes. Herefter sendes beskeden ind i beskedk  en, og tr  den venter p   at den f  r svar ved at tage en semafor. Imens holder notifirtr  den   je med, om der er kommet nogle nye beskeder i beskedk  en. N  r den s   modtager beskeden, behandler den beskeden⁶ , og da dette er en "Select"besked, henter den de   nskede data, og ligger dem i *ReceivedItemsList*. Samtidig frigiver den en semafor, hvilket f  r den anden tr  d til at k  re videre. Tr  den, som kalder *MoveBrickToBox*, ligger s   indholdet fra *ReceivedItemsList* ind i en property, som bruges senere i *MoveBrickToBox* funktionen. Herefter kaldes *InsertPositionToDataBase*, som inds  tter en ny position i databasen, og denne findes ud fra materialetypen, og antallet af eksisterende klodser i databasen med den givne materialetype. Dette foreg  r vha. en "Insert"besked, som ligeledes behandles af notifir-tr  den. Herefter hentes et ID, svarende til den nyoprettede position i boksen. Dette ID skal bruges, n  r klodsen inds  ttes i databasen, hvilket sker i neste funktion: *InsertBrickIntoDataBase*. Denne opretter en "Insert"besked, som skal inds  tte de nyligt fundne data for en klod, samtidig med at der medgives to ID'er, s   klodsen i databasen har to fremmedn  gler; en der peger p   en positionering og en der peger densitet. N  r alle data er persisteret p   databasen, bliver klodsen sorteret alt efter materialeafsnit: *j  vnf  r afsnit: MoveToFirstBox i 8.2.5: i dette*

⁶Se afsnit 8.2.7: implementerings view: observer pattern, for h  ndtering af beskeder

afsnit.

GetMaterialType

Denne funktion hjælper systemet med at finde en materialetype i databasen, som stemmer overens med den udregnede densitet. Dette kan ses på nedenstående sekvensdiagram.



Figur 33: Sekvensdiagram over GetMaterialType()

Først oprettes en "SelectBetween"-besked. Grunden til at der bruges en "SelectBetween"-og ikke en "Select"-skyldes, at der altid vil være et udsving, når densiteten udregnes, og derfor er der nødt til at være en fejmargin, når materialetypen skal findes på da-

tabasen. Derfor medsendes der to parametre i beskeden, den første er densiteten minus en fejlmargin og den anden er densiteten plus en fejlmargin. Fejlmarginen er den største mulige margin, der sørger for, at to standardmaterialetyper ikke overlapper. I koden medgives selvfølgelig også, hvilken tabel der søges i, og hvilke værdier der ønskes fundet osv. Herefter ligges beskeden ind i beskedkøen. Tråden venter så på, at beskeden bliver behandlet af notifiertråden, som ligger de ønskede værdier fra databasen ind i ReceivedItemsList i observeren, selvfølgelig kun hvis dataene kan hentes på databasen. Hvis de ikke kan findes, vil der blive kastet en exception, når det første indhold i ReceivedItemsList bliver tilgået. Denne exception bliver så "fanget" og false bliver returneret. Hvis materialetypen derimod er blevet fundet, gemmes værdierne fra ReceivedItemsList i en property, hvorefter ReceivedItemsList bliver tömt, så den er klar til at indeholde nye data. Når materialetypen er fundet, skal navnet på materialetypen hentes. Hvis det blot havde været en "Select"statement, havde det være muligt at hente begge værdier samtidig, men der er ikke blevet implementeret en "SelectBetween"sætning i databasefacaden, som kan hente flere kolonner. Navnet hentes gennem en "Select"besked, som søger på materialetypen, der netop er blevet fundet. Beskeden ligges ligeledes ind i en beskedkø, og en semafor bliver taget, der søger for at tråden venter på, at beskeden bliver behandlet af notifiertråden. Når den er blevet behandlet, bliver indholdet af ReceivedItemsList igen gemt i en property. Denne property bliver logget samtidig med, at GUI'en⁷ anvender denne property til at udskrive materialetypen på skærmen.

MoveToFirstBox() En af funktionerne fra switch-casen

Funktionen starter med et aktuelt punkt ved boksens bund, med klodsen midt over kanten. Fra dette punkt flyttet klodsen på plads med relative koordinater. Ideen (som også er blevet udført via det relative positioner) er at flytte klodsen ind over kanten, ved at dividere klodsens egen længde med to, og bruge denne længde så klodsen netop går indenfor kanten. Her slippes klodsen hvorefter gripper går om bagved klodsen. herefter skubbes klodsen på plads, så den ligger ved siden af de andre klodser. Dette gøres via boksens længde, minus klodsernes længde⁸, klodsens egen længde, samt gripperens bredde. For at den bliver skubbet lige på plads, har det også være nødvendigt at udregne en rollfaktor.

⁷Vinduet hvor selve sorteringsvinduet bliver vist: RunWindow

⁸Den samlede længde af klodser med den givne materialetype, som er fundet i GetTotalBrickLength()

Denne søger for at gripperen ikke bevæger sig skævt. Når man laver en vektor, skal der medgives en roll-koordinat, der fortæller om gripperen skal rotere. Denne faktor bliver ganget på dette koordinat.

r Faktoren er udregnet på følgende måde

$$\frac{r2 - r1}{\sqrt{(x2 - x1) + (y2 - y1)}} \quad (1)$$

x1 og x2 svarer til koordinatet i den ene ende af boksen og den anden ende af boksen. Ligeledes forholder det sig med r og y koordinaterne. Det har altså været nødvendigt, at finde koordinaterne til denne udregning.

8.2.2 Komponent 2: Simulering

Denne komponent har til formål at simulere robotten. Der er nogle gode grunde til at simulere robotten udeover den, at selve produktoplægget kræver en simulering:

- Først og fremmest skal man være opmærksom på, at der kun er en robot til rådighed. Dermed er det kun en begrænset tid man har adgang til robotten, hvormed det kan blive svært at teste ens programmer.
- Derudover tager det minimum to minutter at home robotten, hvilket ofte vil være en nødvendighed for at teste sit program. Dermed vil den kostbare tid man har ved robotten, blive formindsket endnu mere.

Med disse ulemper taget i betragtning, er det klart, at en simulering vil være en stor fordel.

Design:

Simulatoren består af forskellige simuleringssklasser. Disse oprettes og bruges via constructor-injection på programmet. Der er blevet taget udgangspunkt i, at man skal kunne simulere ens program uden nogen kontakt til robot eller database. Dermed har det være nødvendigt at lave følgende klasser:

- RoboSimulationCalls (implementerer IRoboCalls. Heri ligger samtlige funktioner der bruges til robotten)

- DataBaseSimulation (implementerer IDatabaseFacade)
- DatabaseObserverSimulering (implementerer IObserver)
- FakeWeight (implementerer IWeight)
- FakeRobotEvent (implementerer IRobotEvent)

Forskellen på simulatorerne og fake klasserne, er at simulatorerne logger nogle strenge med information over hvad der er sket, mens fake klasserne sætter nogle variable, der søger for at programmet kan køre uden robotten. Dermed kan man køre det rigtige program, i samspil med robotten, eller nøjagtig det samme program som en simulering. Dette sikres som nævnt via constructor-injection.

En anden vigtig del af simulatoren er loggen. Ud fra log interfacet er der implementeret tre forskellige logningsmetoder, hvor simuleringen bruger to af dem. Simuleringen logger til et vindue, samt til en fil der ligger i en mappen Simuleringslogs i rodmappen (RobotProjekt mappen)

Diagrammerne i afsnit 5.3.7 *Use Case 5* kan give et overblik over simulatoren.

RoboSimulationCalls klassen:

Herunder ses et kodeudsnit fra en af funktionerne i RoboSimulationCalls:

Kodeudsnit 4: Funktionen Initialization(...)

```

1 public bool Initialization( short sMode , short sSystemType )
2 {
3     string sModeS ;
4     string sSystemTypeS ;
5     switch ( sMode )
6     {
7         case 0:
8             sModeS = "INIT_MODE_DEFAULT selects last used mode (←
9                     from ini file).";
10            break ;
11        case 1:
12            sModeS = "INIT_MODE_ONLINE force online mode." ;
13            break ;
14        case 2:
15            sModeS = "INIT_MODE_SIMULAT selects simulation mode."←
16                     ;

```

```

15         break;
16     default:
17         sModeS = "Unknown INIT_MODE.";
18         break;
19     }
20     switch (sSystemType)
21     {
22     case 0:
23         sSystemTypeS = "DEFAULT_SYSTEM_TYPE let the libary ←
24             detect the robot type.";
25         break;
26     case 41:
27         sSystemTypeS = "ER4USB_SYSTEM_TYPE define robot type ←
28             as ER-4 Scrbot with USB connection.";
29         break;
30     default:
31         sSystemTypeS = "Unknown SYSTEM_TYPE.";
32         break;
33     }
34
35     logWindow.LogThis(loggingLevels.INFO, "Initialization: " + ←
36         retVal, false);
37     logWindow.LogThis(loggingLevels.INFO, "Initialization MODE: " ←
38         + sModeS + " ## " + sSystemTypeS, true);
39
40     initRuned = true;
41     return retVal;
42 }

```

Ovenstående eksempel følger den måde, hvorpå simulatoren er opbygget. Den tager parametre, som svarer til et mode eller lignende, og genererer et tekststreg ud fra det. Hvilket mode det svarer til er fundet i USBC-documentationen. Tekststrengeen bliver deretter logget. Derudover er funktionerne opbygget således at de alle returnerer en bool. Denne returværdi er i simuleringsammenhæng en bool, som man selv kan sætte værdien på.

Som det ses i eksemplet sættes et private variabel 'initRuned'. Denne fungerer som validering. Hvis andre funktioner kaldes før denne er sat, får brugeren af vide, at robotten

ikke er initialiseret. Udover denne sættes en variabel i Control og Home funktionen også, da disse er essentielle for at mange af robotfunktionerne kan kaldes.

DatabaseObserverSimulering klassen:

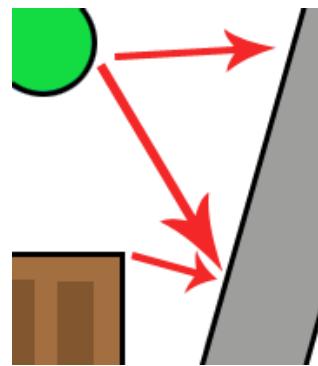
Denne klasse er som sådan ikke en simulering. Den har samme funktionalitet som den rigtige version, men undlader funktionalitet til at håndtere en tabt forbindelse til databasen.

DataBaseSimulation klassen:

Denne håndterer kaldene fra DatabaseObserverSimulering. Den er opbygget efter samme princip som hovedsimulatoren "RoboSimulationCalls", hvor en streng, opbygget ved hjælp af input, logger til konsol og fil.

8.2.3 Komponent 3: Bestem placering af Transportbånd

For at kunne bestemme placeringen for transportbåndet skal der udføres nogle beregninger vha. 3 målinger fra brugeren til transportbåndet. Disse ses på figur 34



Figur 34: Oversigt over målinger til at beregne transportbåndets placering

Med de tre længder kan der beregnes nogle relativt præcise koordinater for, hvor robotten skal samle klossen op på transportbåndet, og hvor den skal sætte den på transportbåndet for at rotere den.

De 3 punkter, *BTBe*(Box-TransportBåndEnde), *RTBe* (Robot-TransportBåndEnde) og *RTBs* (Robot-TransportBåndStart), har bestemte punkter der skal måles mellem.

Eftersom robottens centrum er defineret med koordinatsættet $(0;0)$ i robottens indre, er det ikke muligt, at foretage en måling dertil.

Derfor er punktet, *Robot* valgt ved robottens fod.

For *Box* er punktet ved kassens hjørne valgt.

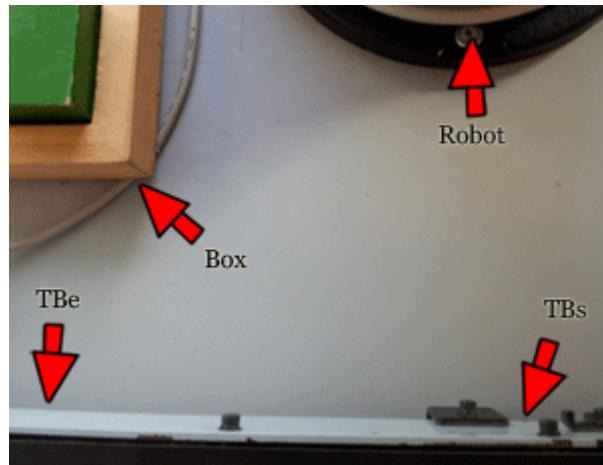
For *TransportBåndEnde* (*TBe*) er et punkt under et hul på transportbåndet valgt.

For *TransportBåndStart* (*TBs*) er et punkt ved siden af transportbåndets fod valgt.

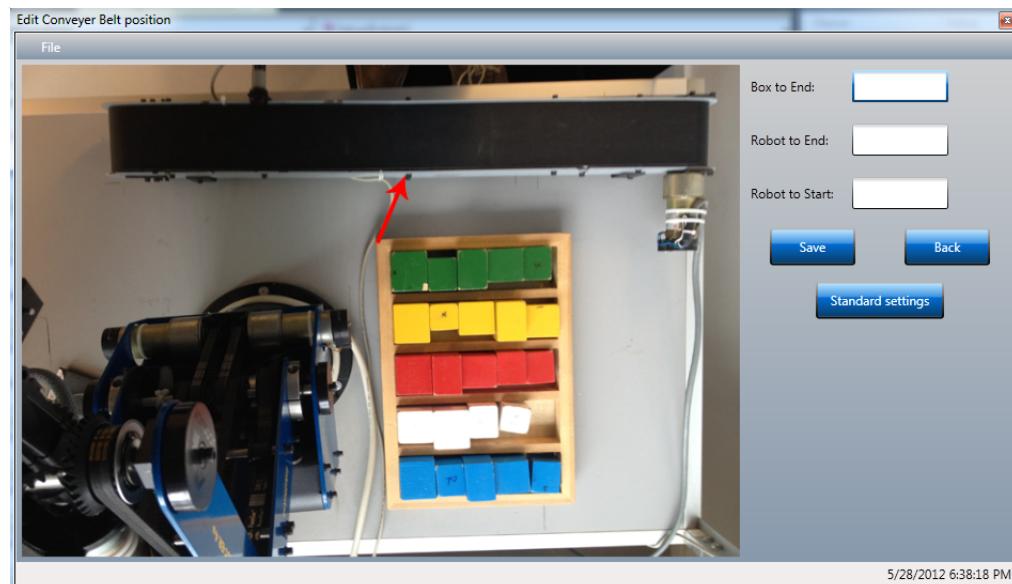
De følgende længder, $|BTBe|$, $|RTBe|$ og $|RTBs|$, skal måles og kan angives i vinduet, vist på figur 36.

For bedre overskuelighed vises målingerne i et koordinatsystem på figur 37.

De tre stiplede linjer på figur 37 er de tre sider der skal måles. De 2 hele linjer er længden mellem robotten og kassen, samt 2 punkter på transportbåndet.



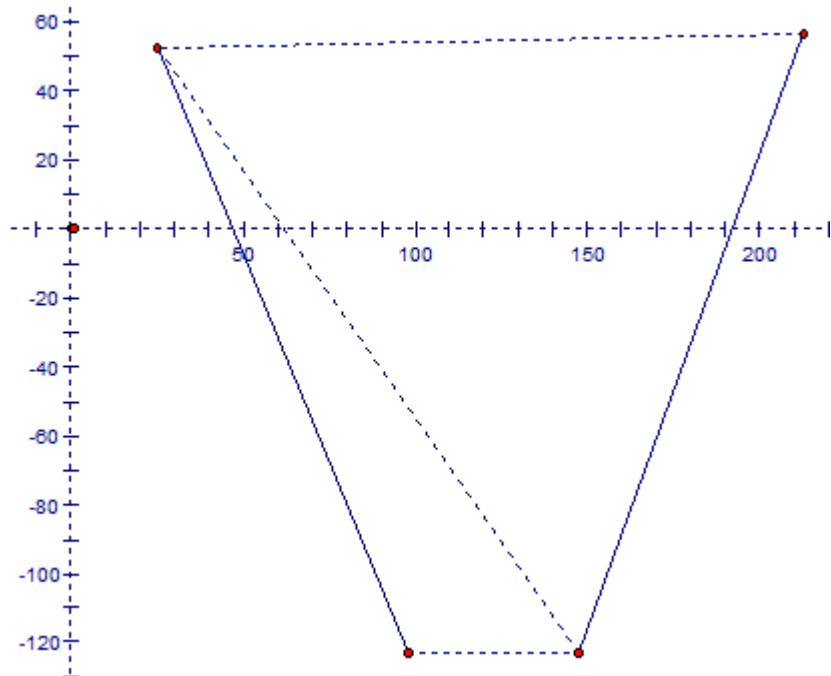
Figur 35: Punkter der måles imellem



Figur 36: Vindue måling skal tastes ind i

Eftersom transportbåndet kan placeres i en vilkårlig position, er der dog nogle begrænsninger:

1. Højden må ikke justeres på nogen måde.
2. Transportbåndet skal være placeret på den viste side, som vist på figur 35.
3. Transportbåndet må ikke roteres til siden.
 - (a) Det må ikke ligge på siden.
 - (b) Det må ikke tilte.



Figur 37: Oversigt over mulig opstilling for transportbåndet

4. Transportbåndet må ikke roteres rundt, således båndet kører fra robotten mod kassen.

BEREGNINGER

For at beregne hvor transportbåndet er, i forhold til robotten og kassen, samt hvor kloksen skal samles op og roteres, kræver en del beregninger. Disse vises herunder:

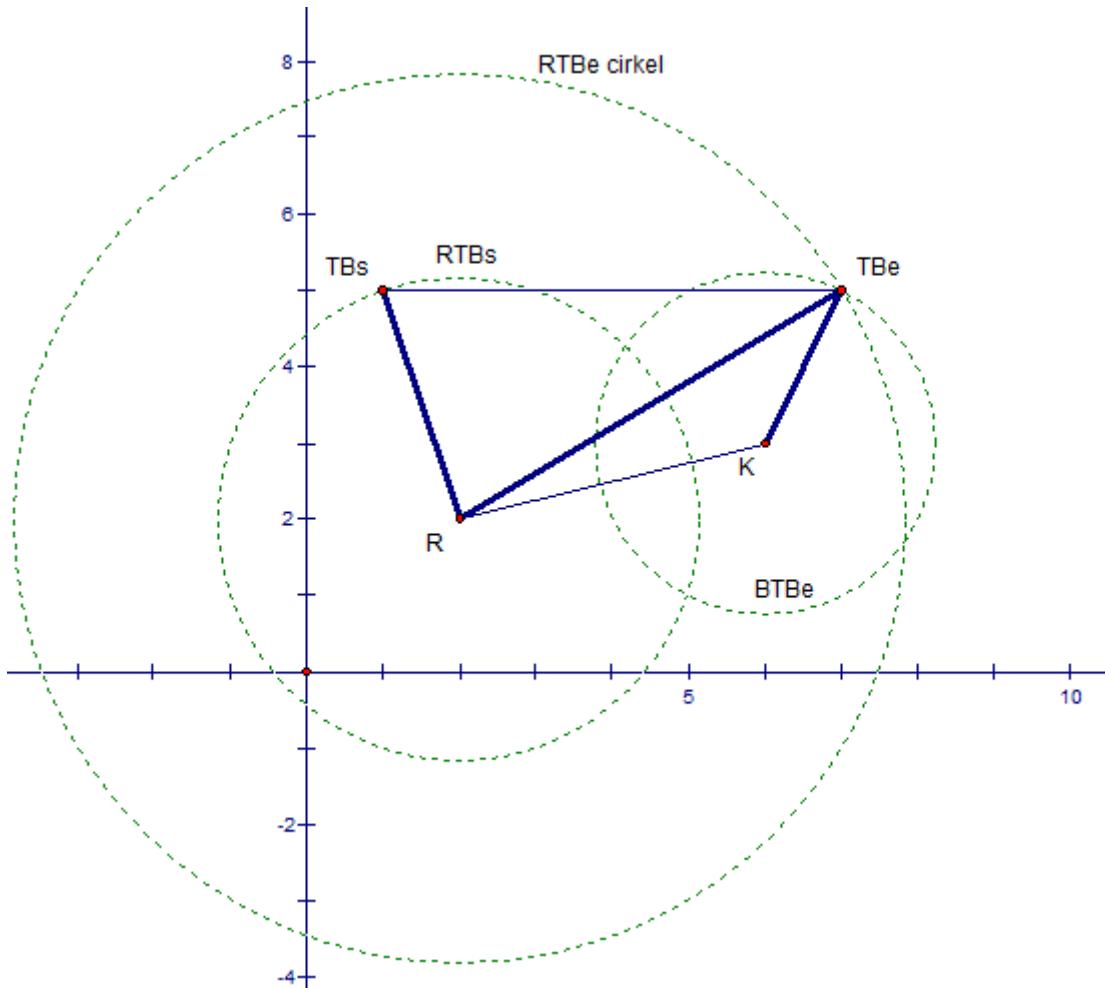
Når de 3 målinger er indtastet, eksekveres følgende metode:

Kodeudsnit 5: Funktionen CalculateTBCoords(...)

```

1 internal void CalculateTBCoords( out double xTBe , out double yTBe←
    , out double xTBs , out double yTBs )
2 {
3     double a1 ;
4
5     a1 = (yKasse - yRobot) / (xKasse - xRobot) ;
6     if (double.IsNaN(a1))
7     {
8         a1 = 120; //Hældning på 120 er meget stejlt og vil gå an
9     }
10

```



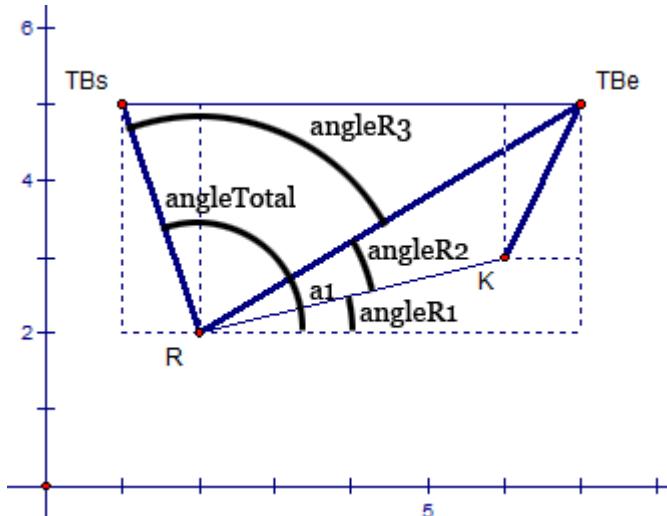
Figur 38: Beregninger for Transportbåndet. De fede streger viser målinger og cirklerne viser hvor længden kunne være

```

11  double angleR1 = Math.Atan(a1) * deg;
12
13  double angleR2 = CosEquation(KTBe, RTBe, RK);
14  var angleTotal = angleR1 + angleR2;
15
16  yTBe = yRobot + RTBe * Math.Sin(angleTotal*rad);
17  xTBe = xRobot + RTBe * Math.Sin((180 - 90 - angleTotal)*rad);
18
19  double angleR3 = CosEquation(TB, RTBe, RTBs);
20  angleTotal += angleR3;
21  yTBs = yRobot + RTBs * Math.Sin(angleTotal*rad);
22  xTBs = xRobot + RTBs * Math.Sin((180 - 90 - angleTotal) * rad);
23 }

```

Beregningerne forklares bedst med en tegning:



Figur 39: Illustration af beregninger

Først beregnes hældningen på mellem kassen og robotten (linje 5):

$$a = \frac{y_2 - y_1}{x_2 - x_1} \quad (2)$$

$$a1 = \frac{y_{kasse} - y_{Robot}}{x_{kasse} - x_{Robot}} \quad (3)$$

Det er muligt, at ændre koordinaterne for kassen eller robotten, således de står "lige over hinanden", altså deres x-værdi bliver ens. Hvis dette er tilfældet vil der blive divideret med nul.

Dette er normalt en ulovlig operation, men da $xKasse$ og $xRobot$ er defineret af typen *double*, vil værdien være *NaN* - uendelig stor.

Skulle det være tilfældet, kan der blot indsættes en meget stor hældning som erstatning, da "uendelig stor" ikke er til at regne med.

Herefter beregnes hældningen i grader (linje 11):

$$\theta = \tan^{-1}(a) \quad (4)$$

$$angleR1 = \tan^{-1}(a1) \cdot \frac{180}{\pi} \quad (5)$$

Der ganges med $\frac{180}{\pi}$, da værdien ønskes gemt i grader og C#'s *Math*-klasse bruger radianner.

Herefter beregnes vinklen $\angle BoxRobotTBe$ med cosinus-relationen (linje 13):

$$\angle A = \cos^{-1} \left(\frac{b^2 + c^2 - a^2}{b \cdot c \cdot 2} \right) \quad (6)$$

$$angleR2 = \cos^{-1} \left(\frac{|RTBe|^2 + |RK|^2 - |KTBe|^2}{|RTBe| \cdot |RK| \cdot 2} \right) \quad (7)$$

De to vinkler ligges herefter sammen, således den fulde vinkel til TBe findes (linje 14):

$$angleTotal = angleR1 + angleR2 \quad (8)$$

Nu kan y-koordinaten til TBe findes vha. sinus-relationen, samt *Robot*'s x- og y-koordinat (linje 16-17):

$$\frac{a}{\sin(A)} = \frac{b}{\sin(B)} \Leftrightarrow \quad (9)$$

$$a = \frac{b \cdot \sin(A)}{\sin(B)} \quad (10)$$

Da $B = 90$ og $\sin(90) = 0$ undlades nævneren

$$a = b \cdot \sin(A) \quad (12)$$

$$y_{TBe} = y_{Robot} + |RTBe| \cdot \sin(angleTotal) \quad (13)$$

$$x_{TBe} = x_{Robot} + |RTBe| \cdot \sin \left((180 - 90 - angleTotal) \cdot \frac{\pi}{180} \right) \quad (14)$$

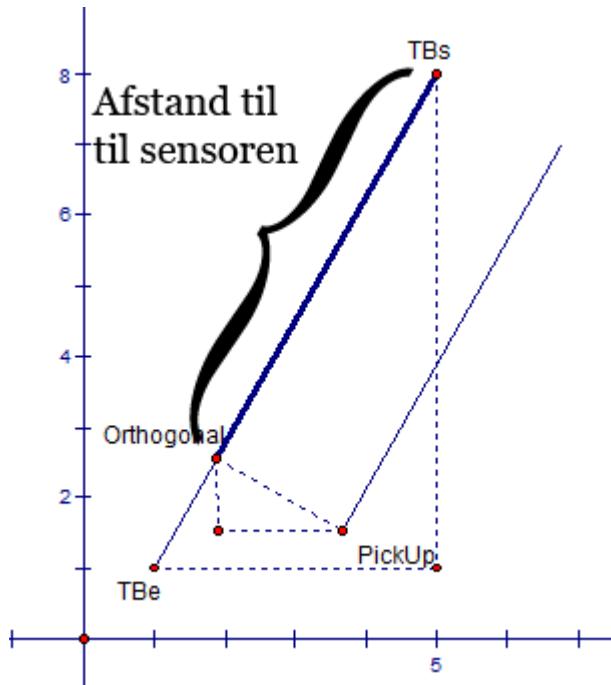
Her ganges der med $\frac{\pi}{180}$ for omskrivning fra grader til radianer.

De $180 - 90 - angleTotal$ er der sidste vinkel i trekanten.

Nu kan vinklen $\angle TBeRobotTBs$ findes ved cosinus-relationen (linje 19), hvorefter *TBs*'s y-koordinat (linje 21) findes vha. *angleTotal* (linje 20) og x-koordinat (linje 21) ligeledes.

Nu kan punkterne, hvor klodsen skal samles op beregnes:

Kodeudsnit 6: Funktionen 'CalculateSensorCoords()'



Figur 40: Illustration af beregning for SensorPickUp

```

1 internal void CalculateSensorCoords()
2 {
3     y4 = _TBe_y;
4     x4 = _TBe_x;
5     y3 = _TBS_y;
6     x3 = _TBS_x;
7
8     double a1;
9
10    a1 = (y4 - y3) / (x4 - x3);
11    if (double.IsNaN(a1))
12    {
13        a1 = 120;
14    }
15
16    double a2 = Math.Abs( -1 / a1);
17    double angleA1 = Math.Atan(a1);
18
19    double y0rtogonal;
20    if (angleA1 * deg < 0)
21        y0rtogonal = y3 + TBSensor * Math.Sin(angleA1);
22    else
23        y0rtogonal = y3 - TBSensor * Math.Sin(angleA1);
24
25    double x0rtogonal;
26

```

```

27     if( angleA1*deg < 0)
28         xOrthogonal = x3 + TBsSensor * Math.Sin((180 - 90 - ←
29             angleA1 * deg) * rad);
30     else
31         xOrthogonal = x3 - TBsSensor * Math.Sin((180 - 90 - ←
32             angleA1 * deg) * rad);
33
34     double GK = HalfBandSize * Math.Sin(Math.Atan(a2));
35
36     if(a1 > 0)
37         y5 = yOrthogonal - GK;
38     else
39         y5 = yOrthogonal + GK;
40
41     double GE = HalfBandSize * Math.Sin((180 - 90 - Math.Atan(a2)←
42             * deg) * rad);
43     x5 = xOrthogonal + GE;
44 }
```

De fundene punkter for transportbåndet læses ind i 4 lokale koordinater (linje 3-6).

Herefter beregnes hældningen på transportbåndet (linje 10) og det checkes om de skulle stå lige over hinanden (Ens x-koordinator). Hvis dette er tilfældet sættes hældningen til 120 (blot et højt tal) (linje 11-14).

Nu findes det punkt, hvor punktet *PickUp* står ortogonalt på (vinkelret):

$$a_2 = \frac{-1}{a_1} \quad (15)$$

Da denne skal bruges til at finde en længe, *skal* dette være positivt (linje 16).

Når hældningen, *a1*, er beregnet, findes den i grader (linje 17).

Først findes det ortogonale punkts y-koordinat:

Vha. Sinus-relationerne (ligning 12) findes forskellen på y-koordinatet.

Hvis *AngleA1* i grader er negativ lægges forskellen til *TBe*'s y-koordinat (linje 20-21).

Hvis *AngleA1* i grader er positiv trækkes forskellen fra *TBe*'s y-koordinat (linje 22-23).

Herefter findes det ortogonale punkts x-koordinatet på samme måde:

Ved at kombinere ligning 12, og reglen om, at en trekant er 180° , beregnes den sidste

vinkel i trekanten:

$$\text{Sidste vinkel} = 180^\circ - 90^\circ - \text{angleA1} \quad (16)$$

Hvis *AngleR1* i grader er negativ lægges forskellen til *TBe*'s x-koordinat (linje 27-28).

Hvis *AngleA1* i grader er positiv trækkes forskellen fra *TBe*'s x-koordinat (linje 29-30).

Når det ortogonale punkt er fundet, findes koordinatsættet til *PickUp* (angivet ved *x5* og *y5*):

Først findes forskellen på y-koordinatet (linje 32) med sinus-relationen. Hvis *a1* er positiv trækkes forskellen fra *yOrthogonal* (linje 34-35).

Hvis *a1* er negativ lægges forskellen til *yOrthogonal* (linje 34-35).

Herefter findes forskellen på x-koordinatet (linje 39) med sinus-relationen og formlen for den sidste vinkel (ligning 16).

Forskellen lægges til *xOrthogonal*.

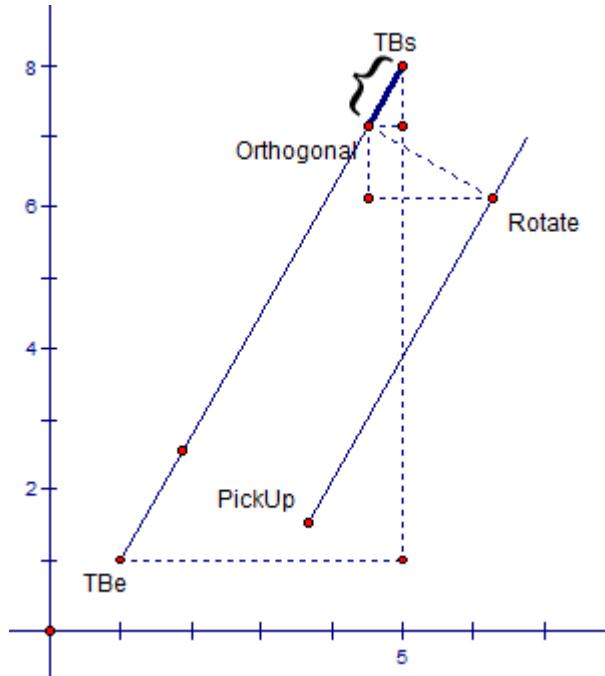
Kodeudsnit 7: Udsnit af funktionen 'CalculateRotateCoords()'

```

19  double yOrthogonal;
20  if (angleA1 * deg < 0)
21      yOrthogonal = y3 + (TBsSensor - TBsPickUp) * Math.Sin((←
22          angleA1));
23  else
24      yOrthogonal = y3 - (TBsSensor - TBsPickUp) * Math.Sin((←
25          angleA1));
26
27  double xOrthogonal;
28  if (angleA1 * deg < 0)
29      xOrthogonal = x3 + (TBsSensor - TBsPickUp) * Math.Sin((180 -←
30          90 - angleA1 * deg) * rad);
31  else
32      xOrthogonal = x3 - (TBsSensor - TBsPickUp) * Math.Sin((180 -←
33          90 - angleA1 * deg) * rad);

```

På kodeudsnit 7 ses et udsnit af funktionen *CalculateRotateCoords()*. Det viste er det eneste der afviger fra funktionen *CalculateSensorCoords()*:



Figur 41: Illustration af CalculateRotateCoords()

Som illustreret på figur 41, beregnes længden til punktet $y_{Orthogonal}$, hvor klosen skal roteres, ved at trække afstanden fra sensoren til punktet fra længden til sensoren og gange med sinus-relationen (ligning 12).

Hvis vinklen (*angleA1*) er negativ, lægges *y3* (*TBs*'s y-koordinat) til beregningen (linje 21).

Hvis den er positiv trækkes det fra $y3$ (linje 23).

$x_{\text{Orthogonal}}$ findes ved samme fremgangsmetode, hvor sinusrelationen (ligning 12) og ligning 16 kombineres.

Hvis vinklen (*angleA1*) er negativ, lægges beregningen til *y3* (*TBs*'s y-koordinat) (linje 30).

Hvis vinklen ($angleA1$) er positiv, trækkes beregningen fra $y3$ (TBs 's y-koordinat) (linje 30).

For at bruge beregningerne kaldes funktionen *Messure(...⁹)*, hvorefter 2 arrays med index hver i sær på 2.

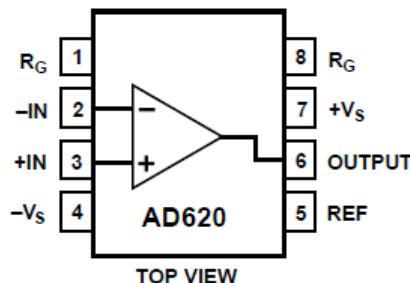
Herefter kaldes *PickUpCoords(out array1)* og *RotateCoords(out array2)* og de to koordinatsæt kan nu tilgås igennem de to arrays. Herefter

⁹Her mangler 3 parameter med forståelige navne

8.2.4 Komponent 4: Weight

Den udleverede Strain gauge-vægtcelle leverer en spænding på 1 mV/V. Da den anbefalede forsyningsspænding jævnfør databladet er 10V, er denne valgt, det betyder at cellen leverer 10 mV per Kg vægt. Der ønskes at måle i området 0 - 1000 gram. Dvs. den udsendte spænding er mellem 0 og 10 mV. STK500 udviklingskittets A/D-konverter operer med 0 - 5V (5,5V max). For at få vægtcellen til at passe med dets spændingsinterval skal signalet forstærkes. Det hjælper med til at få en god præcision/opløsning at forstærke signalet. Det betyder at en forstærkning på 500x vil være ideel. Altså ønskes forstærkning med et gain på maksimum 500. Da signalet fra en Strain gauge i sin natur er meget svag, er det også meget udsat for støj. Et andet krav til forstærkeren er derfor at den skal kunne håndtere støj. Derfor blev en instrumentationsforstærker valgt, frem for blot en enkelt operationsforstærker.

CONNECTION DIAGRAM
**8-Lead Plastic Mini-DIP (N), Cerdip (Q)
and SOIC (R) Packages**



Figur 42: AD620 IC kredsen

Kredsen AD620 lever op til overstående krav, og den er derfor valgt til formålet. Kredsen kræver kun en modstand, R-gain, til at styre forstærkningen, som kan beregnes ud fra nederste formel på Figur 43:

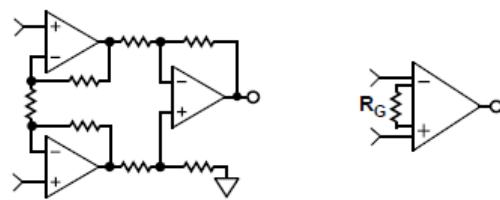
The gain equation is then

$$G = \frac{49.4 \text{ k}\Omega}{R_G} + 1$$

so that

$$R_G = \frac{49.4 \text{ k}\Omega}{G - 1}$$

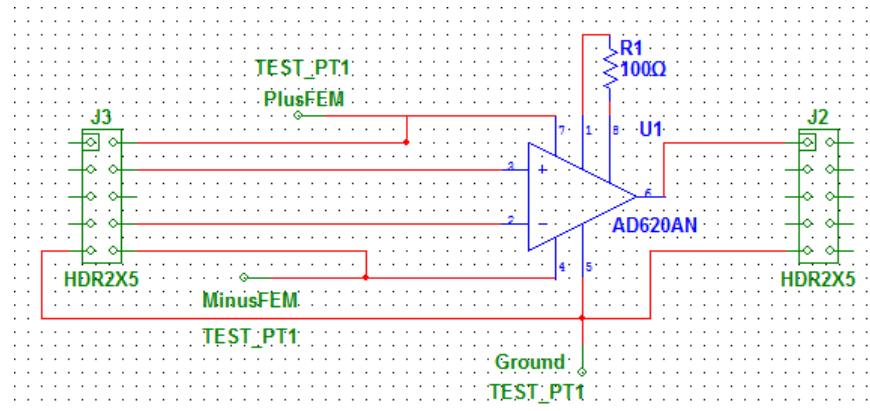
Figur 43: AD620 formel



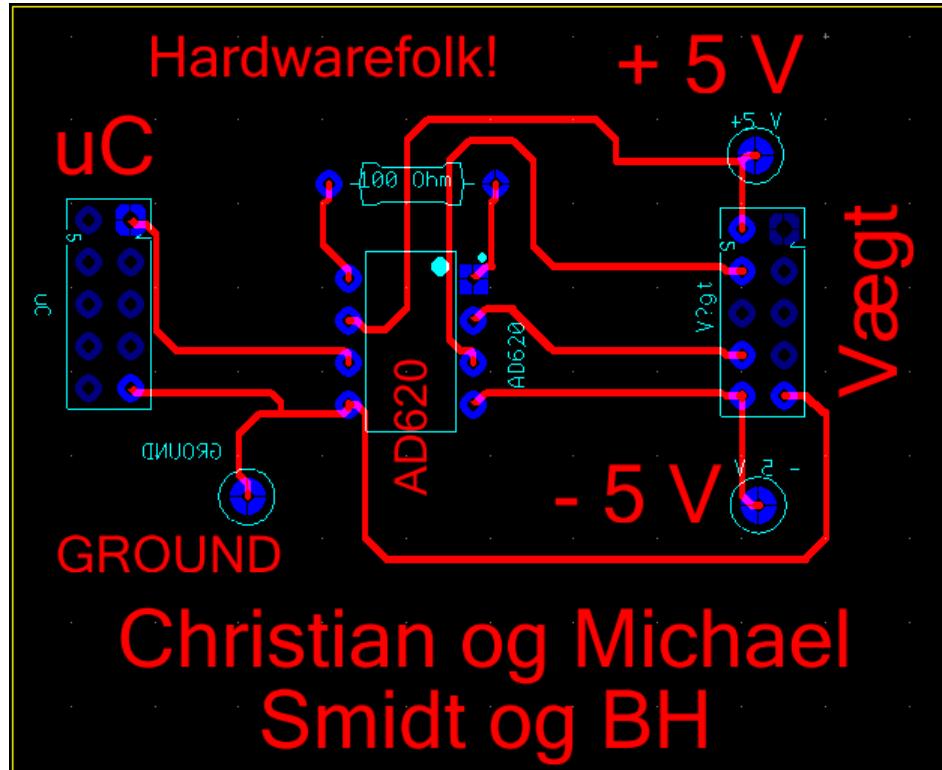
Figur 44: AD620 IC

Der er valgt en R-gain på 100 Ohm, hvilket giver en forstærkning på 495 gange, hvilket er tilstrækkeligt, selvom forstærkningen ikke er den ønskede på 500x. Det betyder at udgangsspændingen bliver 0-4.95V, og med en ADC-opløsning på 10Bit mulighed for at sample på helt ned til <1g præcision.

AD620 arbejder på forsyningsspænding på -5V og +5V, hvilket også gives til vægtcelen, som altså får 10V spændingspotentiale. Det færdige kredsløb blev som følger:

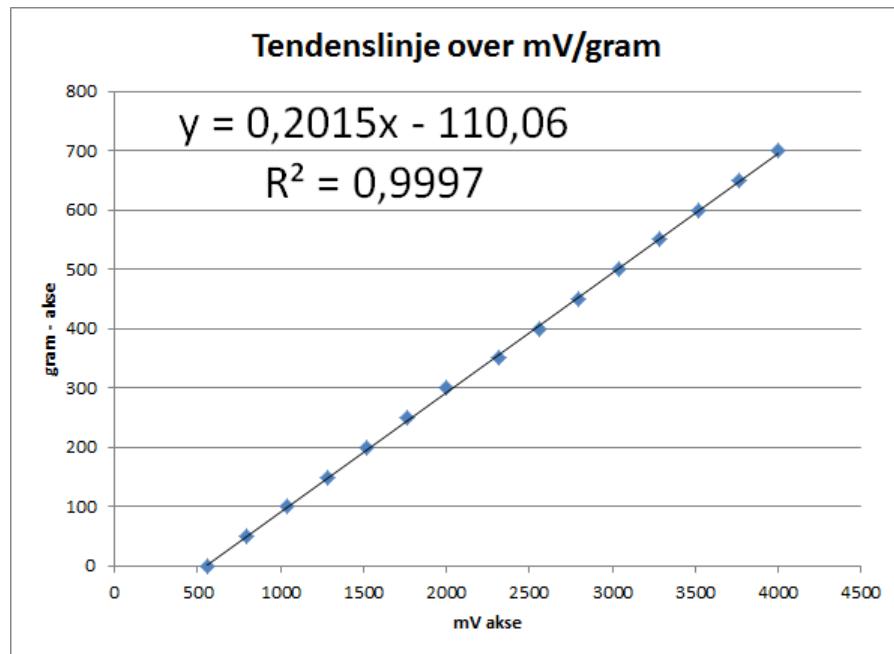


Figur 45: Forstærker kredsløb, Multisim



Figur 46: Forstærker kredsløb, Ultiboard

På figur 47 ses en graf af udgangsspændinger fra AD620 ved en given vægt.

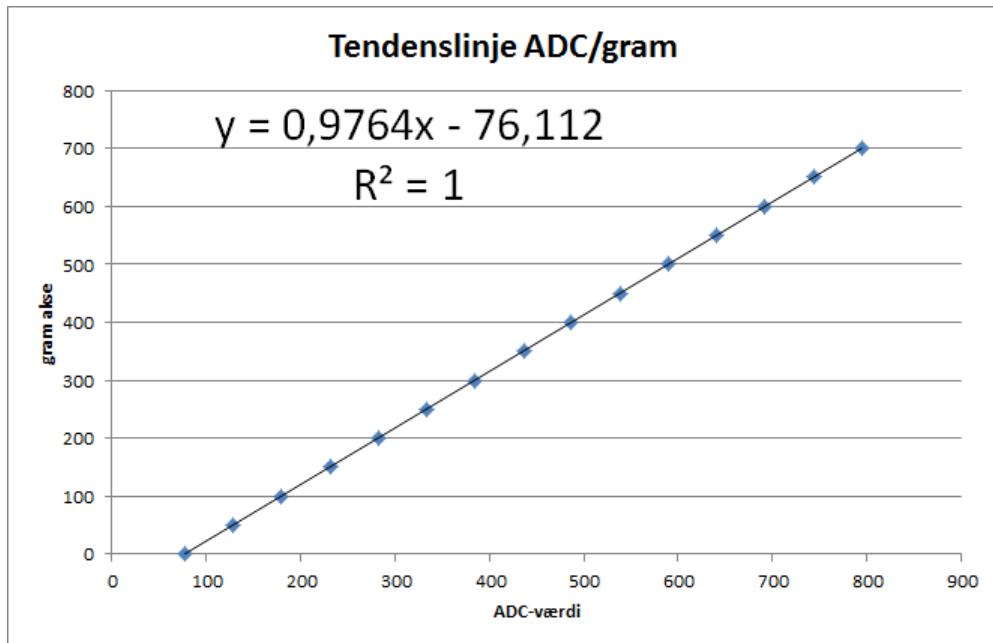


Figur 47: Udgangsspænding mV / Vægt gram - graf

gram	0	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750
mV	560	800	1040	1280	1520	1760	2000	2320	2560	2800	3040	3280	3520	3760	4000	4320

Figur 48: Udgangsspænding mV / Vægt gram - datasæt

På figur 49 ses en graf for ADC værdien ved en given vægt.



Figur 49: ADC værdi / Vægt gram - graf

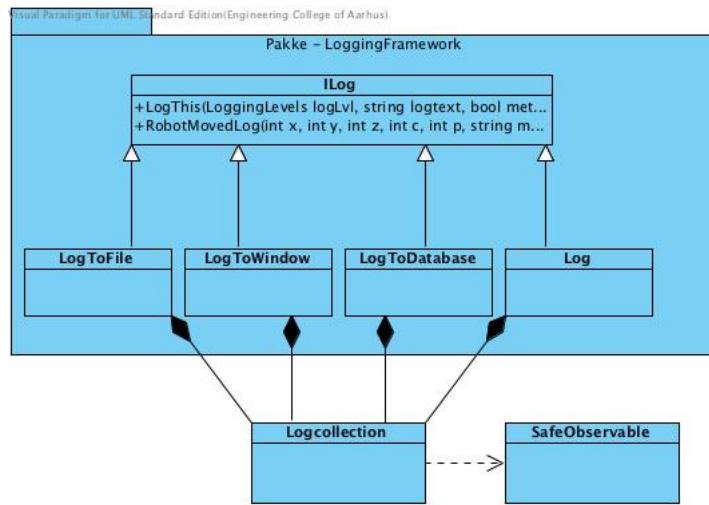
gram	0	50	100	150	200	250	300	350	400	450	500	550	600	650	700
ADC mV	78	129	180	232	283	334	385	437	487	539	590	641	692	744	795

Figur 50: ADC værdi / Vægt gram - datasæt

8.2.5 Komponent 5: Log

Denne komponent fungerer som et fælles loggingframework for hele systemet. Komponenten er implementeret via et singleton designpattern, hvilket sikrer, at det er den samme log der benyttes på tværs af systemet. Den vigtigste funktion i komponenten er, at indskrive loggen i systemets database, men indeholder også mulighed for at logge til fil, logge til et WPF-vindue og en log der bruges i simuleringsøjemed.

Loggen er baseret på et centralt interface, ILog, som indeholder funktionen logThis() og RobotMoved. Dette interface implementeres af forskellige klasser, der henholdsvis har til opgave at logge til forskellige medier. Dette betyder at logtypen kan udskiftes uden større problemer, endda på runtime. Strukturen er fremvist på billedet herunder:



Figur 51: Sturktur over Log

Implementeringen af LogThis varierer fra logtype til logtype. Mens der altid logges et tidspunkt, et loglevel og en besked, er det ikke altid der logges f.eks. metoden eller filen hvorfra kaldet kommer. Metoden RobotMoved er en version af LogThis, som er specialiseret til at indsætte beskeder om robottens bevægelse. Loggen er baseret på et loglevel, defineret som en global enum, der indeholder alle loglevels.

LogFile

Der gemmes her persistent til en fil på disken. Filen gemmes som udgangspunkt i programmappen med et filnavn bestående af logtidspunktet.

LogToWindow

En version, der logger til en collection, der implementerer INotifyPropertyChanged og derfor kan bindes til fra viewet

LogToDB

Den log der bruges under et normalt programforløb, gemmer til databasen. Dette viste sig problemfyldt, da brugergrænsefladen ikke kunne binde dertil. Derfor blev det valgt også at opretholde en lokal kopi af loggen. Denne kopi skulle ydermere tilgåes af en tråd anden end Dispatcher, hvilket ikke er muligt med en ObservableCollection, da denne ikke er trådsikker. Dette problem blev løst ved at lave en klasse SafeObservable, som både implementerer INotifyPropertyChanged interfacet og indeholder trådbeskyttelse.

8.2.6 Komponent 6: Beskedkø, observer pattern og database tilgang

Komponenten består af tre dele; Beskeden, en beskedkø og en observer. Denne del af dokumentet vil beskrive hvordan databasen tilgås ved hjælp af disse tre dele.

Besked:

Klassen "Message" består udelukkende af overloadede constructorer. Alt efter hvilke funktionaliteter man vil tilgå på databasen, laves en specifik Message. Hvis brugeren f.eks. vil lave et Select statement uden en where clause, laves en message med tre parametre; En operation, en tabel, og de kolonner der skal hentes. Hvis brugeren gerne vil have en where clause på beskeden, skal han lave en besked med fem parametre; De tre forrige samt en kolonne der skal søges i og en værdi der skal søges efter. Inde i denne klasse ligger der også flere enumerations; En der indeholder hvilken operation der skal laves, en der indeholder de forskellige tabeller man kan tilgå og en for hver tabel med de kolonner der eksister i tabellen. På kodeudsnit 8 ses hvordan en select statement uden where clause bliver oprettet.

Kodeudsnit 8: Eksempel på en Message constructor

```

1 public Message(Operations operation, Tables table, string column)
2 {
3
4     Op = operation;
5     Tb = table;
6     Column = column;
7 }
```

Disse beskeder kan tilgås frit fra hele systemet. Når beskeden er oprettet bliver de lagt i beskedkøen.

Beskedkø:

Klassen MsgQueue bruges til samle beskeder fra hele systemet og sørge for, at de beskeder der kom først bliver håndteret først. Selv hvis forbindelsen til databasen forsvinder, så er det muligt for systemet at fortsætte, indtil der skal hentes noget fra databasen. Når der skal hentes noget fra databasen, bliver systemet nødt til at vente, da en select statement ofte er kritisk for systemets videre kørsel. Klassen består af en privat Queue der

der typedefineret til objekter af typen Message. Til at tilgå køen er der blevet lavet tre funktioner; En til at tage den forreste Message fra køen, en til at ligge en message på køen og en til at kigge på den forreste Message i køen. Beskedkøen er bygget op som Singleton designmønsteret, dvs. uanset hvor du tilgår beskedkøen fra, er det den samme kø. Det er også Publisher delen af Observer design mønsteret. Dertil har den en funktion, NotifyAll, der fortæller Subscriberen, DatabaseObserver klassen, at en besked er klar til at blive håndteret, og venter ellers bare hvis køen er tom. På kodeudsnit 9 kan det ses hvordan NotifyAll funktionen er implementeret

Kodeudsnit 9: Eksempel på en Message constructor

```

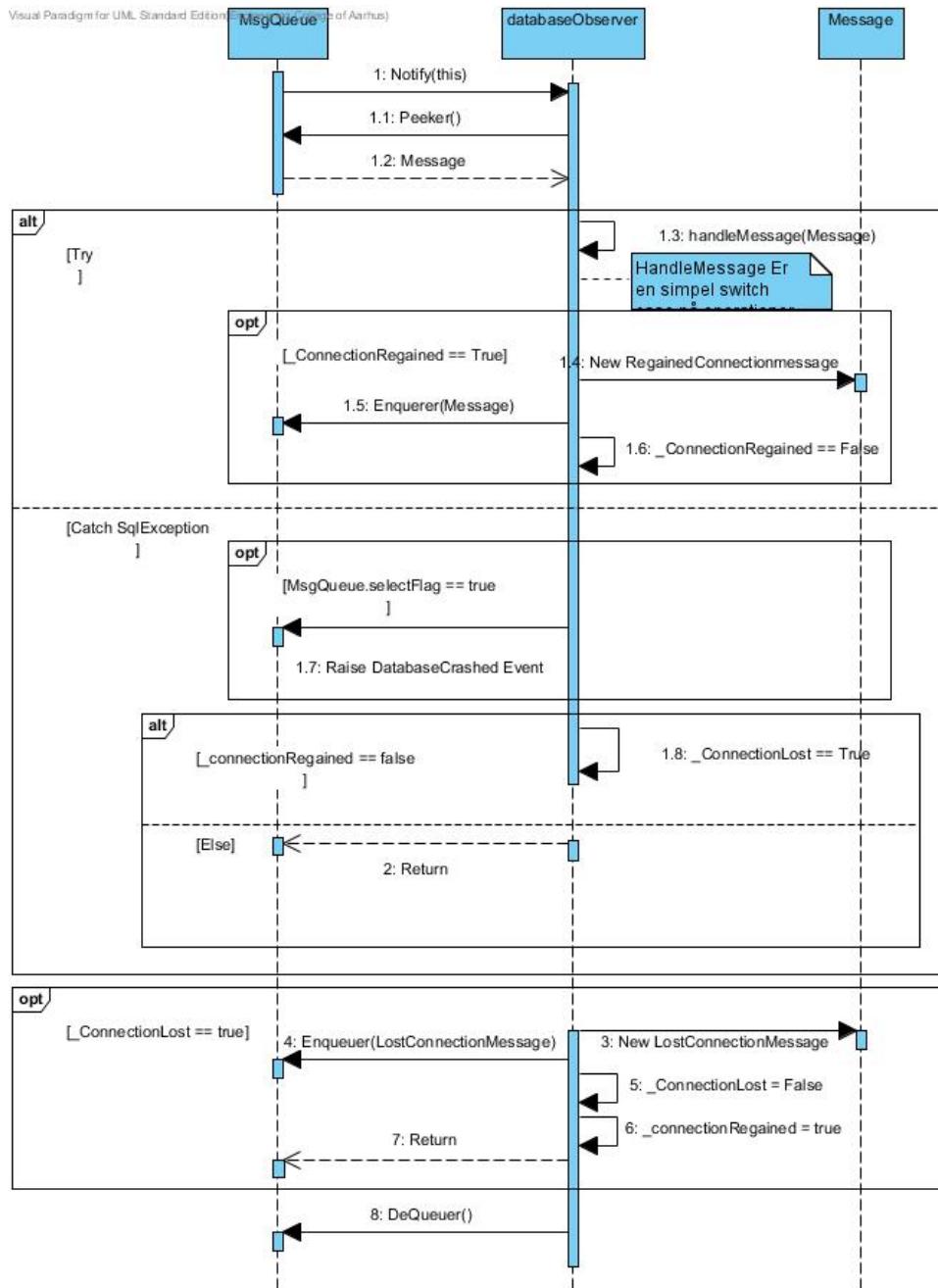
1 public void NotifyObservers()
2 {
3     while (true)
4     {
5         if (_dbQueue.Count != 0)
6         {
7             selectFlag = false;
8             foreach (Message m in _dbQueue)
9             {
10                 if (
11                     m.Op == Operations.Select
12                     || m.Op == Operations.SelectDistinct
13                     || m.Op == Operations.SelectMax
14                     || m.Op == Operations.SelectBetween
15                 )
16                     selectFlag = true;
17                 }
18             _dbObserver.Notify(SingletonMsgQueue);
19         }
20
21         else
22         {
23             Thread.Sleep(35);
24         }
25
26     }
27 }
```

Hvis køen er tom ventes der i 35 millisekunder, ellers undersøges der om der ligger en besked, der skal hente fra databasen i køen. Hvis der gør sættes der et flag, som bruges i

DatabaseObserver klassen. Herefter kaldes notify på observeren, med køen som parameter. Hvis forbindelsen forsvinder vil der i DatabaseObserver klassen kaste en event, som gribes i MsgQueue klassen. Denne spørger brugeren om han vil prøve at skabe forbindelse igen, eller om han vil lukke systemet ned.

Observer:

Klassen DatabaseObserver er subscriber delen af Observer design mønsteret og har til opgave at håndtere beskederne til databasen, samt at vedligeholde forbindelsen. Klassen består af to hoved funktioner og to trådfunktioner som er beskrevet nærmere i afsnit 6.3 Kommunikation og synkronisering. Den første hovedfunktion er Notify som kaldes fra klassen MsgQueue. Funktionens vigtigste opgave er, at hente en besked fra køen og kalde den næste hovedfunktion, handleMessage, med den hentede besked som parameter. Ud over dette skal den også gøre den exception der kastes, hvis forbindelsen fejler og ligge en besked i køen, der beskriver at forbindelsen er tabt. Når forbindelsen reestablishes skal den også ligge en besked i køen der beskriver at forbindelsen er reestablished. Desuden skal den event der bliver brugt i beskedkø delen også kastes, hvis der er en select besked i køen. På figur 52 kan man se et mere detaljeret hændelsesforløb over Notify funktionen. Sekvensdiagrammet kan ses i bilag under Diagrammer/Detaljerede sekvensdiagrammer



Figur 52: dSSD over Notify i DatabaseObserver klassen

I den anden hovedfunktion, handleMessage, bliver database facaden tilgået, alt efter hvilke attributer der er sat i beskeden. Den består i realiteten af en switch/case på den operation der er sat i beskeden. I nogle af casene er det muligt at to forskellige funktioner kan tilgås, alt efter om en specifik attribut er null eller ej. På kode eksempel 10 ses det hvordan facaden tilgås ved en switch case og hvordan det besluttes om funktionen skal tilgås med eller uden en where clause.

Kodeudsnit 10: Eksempel på en Message constructor

```

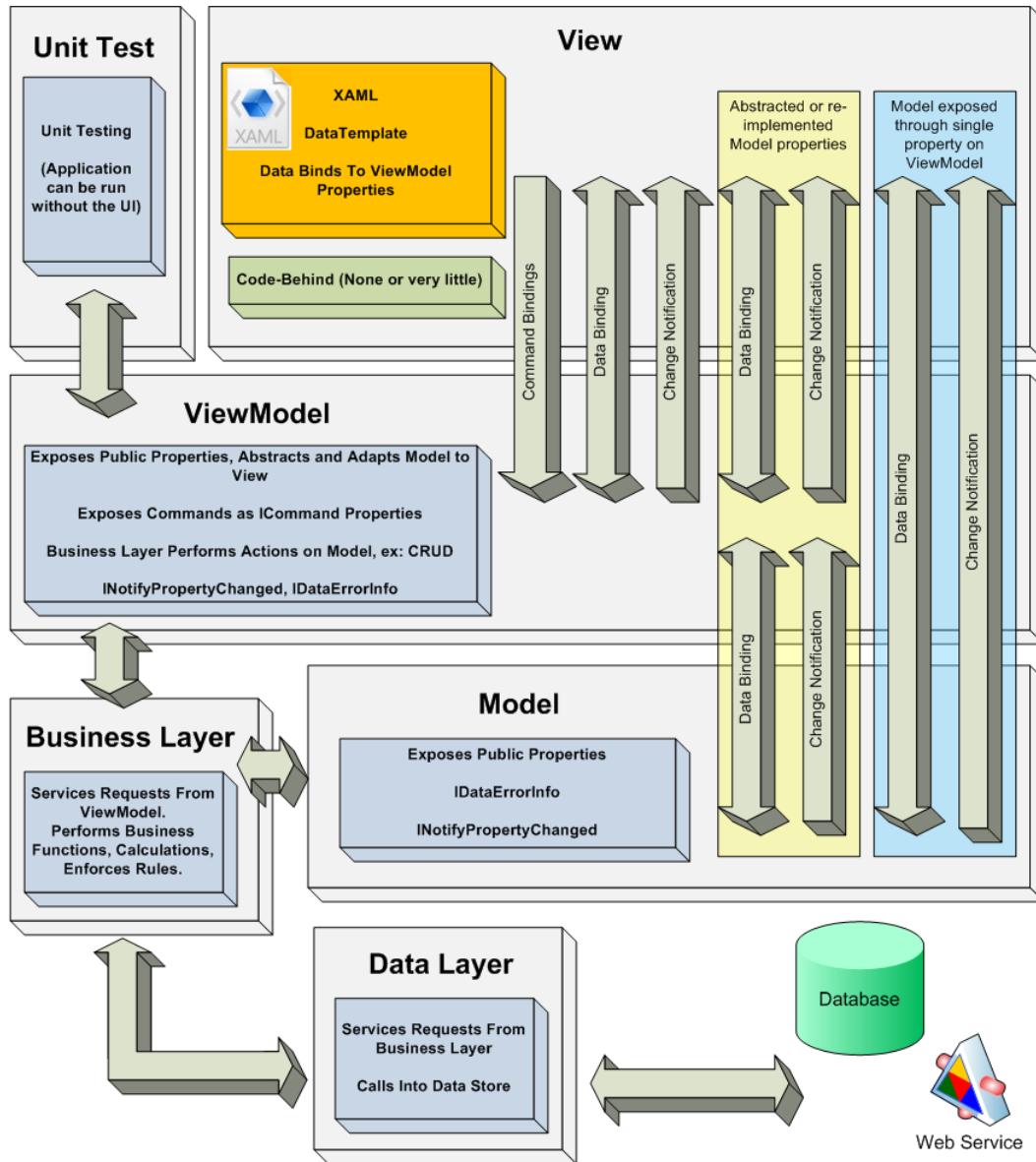
1  .....
2 switch(msg.Op)
3 {
4     case Operations.Select:
5         if (whereColumn == null)
6         {
7             RetrievedItemsList =
8                 _dbFacade.SelectFrom(
9                     column, table
10                    );
11             MsgQueue.HandlingSelect = false;
12             SemPool.Release();
13             break;
14         }
15     else
16     {
17         RetrievedItemsList =
18             _dbFacade.SelectFrom(
19                 column, table,
20                 whereColumn, value.ToString()
21                 );
22         MsgQueue.HandlingSelect = false;
23         SemPool.Release();
24         break;
25     }
26 }

```

Det forrige eksempel viser håndteringen af select statement. Hvis whereColumn ikke er sat, betyder det, at der ikke vil laves en select statement med en where clause. Herefter hentes der fra databasen og ligges over i RetrievedItemsList. HandlingSelect bruges i tilfælde af, at to forskellige tråde laver en select besked på samme tid. Release() bruges til at signalere til den tråd der venter, at den må fortsætte. Der kan læses mere om dette i Process/Task view unde afsnit 6.3 Kommunikation og synkronisering

8.2.7 Komponent 7: GUI/MVVM

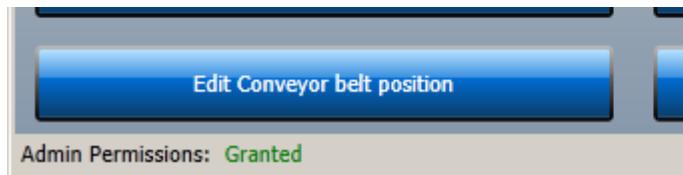
Denne komponent er en implementering af Præsentationspakken beskrevet i afsnit 5. Komponenten består af 3 dele: Viewet, ViewModellen og tildels Modellen. Den er baseret på MVVM, beskrevet under afsnit 10.2. Den grundlæggende struktur er som følger:



Figur 53: Opbygningen af MVVM. Både view,.viewmodel og model afspejles direkte i koden

View Selve view fremgår i koden som XAML-dokumenterne der beskriver de forskellige vinduers udseende. Altså definerer view kun vinduers grafiske repræsentation, og ikke

den bagvedliggende logik. View'et er bundet med binding via properties og commands til viewmodellen. Nedenstående eksempel viser hvordan en del af viewet er lavet, de to billeder viser en del af GUI'en alt efter man er logget ind som "programmer" eller "operator". Det ses at der står henholdsvis 'Admin Permissions: ' 'Granted' eller 'Not Granted'. Ligeledes er det heller ikke muligt at trykke på knappen 'Edit Conveyor belt position'.



Figur 54: MenuWindow udsnit fra programmer rettigheder



Figur 55: MenuWindow udsnit fra operator rettigheder

Kodeudsnit 11 viser hvordan koden for 'Admin Permissions' er lavet i view'ets XAML kode. Her er TextBlock'ens text opkoblet 'bindet' til en TextBlockken 'AdminPermissionsItem' i viewmodellen.

Kodeudsnit 11: ViewAdminPermissions

```

1  <TextBlock  Text="Admin Permissions: " />
2  <TextBlock  Text="{Binding Path=AdminPermissionsItem.Text}"
3      Foreground="{Binding Path=AdminPermissionsItem.Foreground<-
 }"/>

```

Kodeudsnit 12 viser hvordan decideret metoder bliver kaldt oppe fra præsentationslags view og ned af i viewmodellen, her er viewet bundet til en Command i viewmodellen.

Kodeudsnit 12: ViewConveyerBelt

```

1  <Button
2      Command="{Binding Path=ConveyerBelt}"
3      Content="Edit Conveyor belt position" >

```

```

4     <Button.ToolTip>
5         <TextBlock Text="Here the user is able to change the ←
          position of the elements in the system" />
6     </Button.ToolTip>
7 </Button>

```

Viewmodel Viewmodellens opgave er at omforme modellen så den passer til viewet. Ambitionen er at man umiddelbart kan skifte viewet udelukkende ved at fortage ændringer i viewmodellen. Implementeringsmæssigt er der lavet en.viewmodel for hvert vindue, denne indeholder en række properties og relaycommands, som kalder ned i modellen. Viewmodellerne bliver oprettet og nedlagt som der skiftes view. Alle controls fra View bindes til viewmodellen. Dog indeholder viewmodellen ingen reel data, men fungerer kun som en slags adapter, der returnere public properties fra modellen. Implementeringen af INotifyPropertyChanged/RaisePropertyChanged og Relaycommands er hentet fra galasoft MVVM-light toolkittet.

Kodeudsnit 13 viser kort hvordan der kaldes videre ned i modellen. Her sættes Acess variablen, som senere bliver set på om hvorvidt man har rettighederne til diverse ekstra funktioner. Kodeudsnit 14 viser hvordan TextBlocken fra før bliver sat alt efter om brugeren har rettigheder, ligeledes bestemmes det om brugeren kan trykke på ConyeverBelt knappen eller ej.

Kodeudsnit 13: ViewModelConstructoren

```

1 public MenuViewModel() // Constructoren
2 {
3     model = new MenuModel();
4     Acess = model.Privileges(); // Model funktionen kalder ned i ←
          databasen .
5     AdminRights();
6 }

```

Kodeudsnit 14: ViewModelAdminRights

```

1 private void AdminRights()
2 {

```

```

3  if (Acess == "A") // Admin/Programmer Permissions
4  {
5      ConyeverBeltButtonIsEnabled = true;
6      AdminPermissionsItem.Text = "Granted";
7      AdminPermissionsItem.Foreground = Brushes.Green;
8  }
9  else if (Acess == "U") // User/Operator Permissions
10 {
11     ConyeverBeltButtonIsEnabled = false;
12     AdminPermissionsItem.Text = "Not Granted";
13     AdminPermissionsItem.Foreground = Brushes.Red;
14 }
15 }
```

Her, linje 15, er selve propertien viewet binder til i viewmodellen, når denne ændres kaldes som sagt en RaisePropertyChanged der får viewet til at opdatere ændringerne.

Kodeudsnit 15: ViewModelAdminPermissionsItem

```

1 private TextBlock _AdminPermissionsItem;
2 public TextBlock AdminPermissionsItem
3 {
4     get { return _AdminPermissionsItem; }
5     set
6     {
7         _AdminPermissionsItem = value;
8         RaisePropertyChanged("AdminPermissionsItem");
9     }
10 }
```

Nedenstående kodeudsnit 16 er den Command som view knappen 'ConyeverBelt' er bundet til. Det er kun muligt at trykke på denne knap hvis man har administratorrettigheder. Dette ses i funktionen ConyeverBeltCanExecute() der tjekker på om ConyeverBeltButtonIsEnabled er sand, denne variable sættes jo som vist i det foregående kodeudsnit

14

Kodeudsnit 16: ViewModel ICommandConyeverBelt

```

1 void ConyeverBeltExecute()
2 {
3     Messenger.Default.Send(new NotificationMessage("←
```

```

        ConveyorBeltWindow" ) );
4 }
5
6 bool ConyeverBeltCanExecute()
7 {
8     if (ConyeverBeltButtonIsEnabled)
9         return true;
10    return false; // Else
11 }
12
13 public ICommand ConveyerBelt
14 {
15     get{return new RelayCommand(ConyeverBeltExecute,←
16         ConyeverBeltCanExecute);}
16 }

```

For at få åbnet det nye vindue 'ConveyorBeltWindow' når der trykkes på knappen 'Edit Conveyor belt position' er det nødvendigt, at sende en besked op til View-lagets code-behind. Det sker i funktionen ConyeverBeltExecute(), ses i kodeudsnit 16. Det er det såkaldte Mediator mønster. Udsnittet 17 viser Code-behind fra menuvinduet, hvor NotificationMessage modtages, hvorpå der det nye vindue åbnes, samt det gamle vindue lukkes.

Kodeudsnit 17: ViewNotification

```

1 Messenger.Default.Register<NotificationMessage>(this, ←
    ConveyorBeltPositionNotficationMessageReceived);
2 private void ConveyorBeltPositionNotficationMessageReceived(←
    NotificationMessage msg)
3 {
4     if (msg.Notification == "ConveyorBeltWindow")
5     {
6         var conveyorbeltwindow = new ConveyerBeltWindow();
7         conveyorbeltwindow.Show(); // Åbner conveyorbeltwindow
8         Close(); // Lukker menuVinduet
9         Messenger.Default.Unregister(this); // Sletter beskeden.
10    }
11 }

```

Model Modellen er den egentlige dataklasse. Der er lavet en modelklasse for hvert view. På den måde er data grupperet efter hvilket vindue de skal vises i. Dog er der data der skal optræde i hvert view, og derfor er der lavet en overliggende modelklasse, placeret i

App-klassen, som alle viewmodeller kan tilgå. Dette sikrer, at redundant data undgåes. Der er en væsentlig forskel fra illustrationen og til vores implementering. Modellen indeholder nemlig også alt GUI-nær logik. Det betyder at commands i viewmodellen kalder gennem modellen til Business Layer. Altså har viewmodellen ikke direkte fat Business Layer (Dette gælder med henblik på figuren. i forhold til hele systemet kan modellen betragtes som en del af Business layer).

Model-laget for Menu-vinduet er ikke synderligt stort. Det kalder blot ned i forretningslogikken i den store hoved 'Model' klasse der binder alle de forskellige komponenter der hører til projektet. Her ses et kodeudsnit af Menumodellen, 18, der tjekker på om en property er 'Admin' eller 'User' ned i hoved model-klassen.

Kodeudsnit 18: MenuModel

```

1 public class MenuModel
2 {
3     Model model = (Application.Current as GUI.App).model;
4
5     public string Privileges()
6     {
7         if (model.isAdminRightsGranted == "A")
8             return "A";
9         if (model.isAdminRightsGranted == "U")
10            return "U";
11         return null;
12     }
13 }
```

Hoved Model-klassens property sættes i Login skærmens model-klasse. Så for en ordens skyld er koden for login her, 19:

Kodeudsnit 19: Login

```

1 public bool Login()
2 {
3     if (UserName != null || Password != null)
4     {
5         if (UserName.Length >= 1 && Password.Length >= 1)
6         {
7             bool isTrue = VerifyPassword(UserName, Password);
```

```

8     if (isTrue) // At man har rettigheder
9     { // Her bestemmes så HVILKE rettigheder man har.
10        m_model.isAdminRightsGranted = VerifyAdminRights(←
11            UserName);
12        return true;
13    }
14 }
15 return false;
16 }
```

Kodeudsnit 20: VerifyPassword

```

1 public bool VerifyPassword(string userName, string password)
2 {
3     // Password hentes fra DB ved at slå op under brugernavnet
4     Message getPassMsg = new Message(Operations.Select, Tables.User←
5         , UserColumns.PK_UserName.ToString(), userName, UserColumns.←
6         Password.ToString()); // message oprettes
7     MsgQueue.SingletonMsgQueue.Enqueue(getPassMsg); // Message ←
8         indsættes i køen
9     m_model.observer.WaitSem(); // Der ventes
10 }
11 if (m_model.observer.RetrievedItemList.Count == 0)
12     return false;
13 if (password == m_model.observer.RetrievedItemList.First())
14     return true;
15 return false;
16 }
```

Kodeudsnit 21: VerifyAdminRights

```

1 public string VerifyAdminRights(string userName)
2 {
3     // Password hentes fra DB ved at slå op under brugernavnet
4     var getPassMsg = new Message(Operations.Select, Tables.User, ←
5         UserColumns.PK_UserName.ToString(), userName, UserColumns.←
6         User_Privileges.ToString()); // message oprettes
7     MsgQueue.SingletonMsgQueue.Enqueue(getPassMsg); // Message ←
8         indsættes i køen
9     m_model.observer.WaitSem(); // Der ventes
10 }
```

```

9  if ("A" == m_model.observer.RetrievedItemsList.First())
10  return "A";
11  if ("U" == m_model.observer.RetrievedItemsList.First())
12  return "U";
13  else
14  return null;
15 }

```

8.2.8 Komponent 8: IDE

Denne komponent gør det muligt at skrive, gemme, hente og slette sekvenser til robotten ved hjælp af en række foruddefinerede funktioner, ydermere gør den det muligt at tjekke den skrevet sekvens for syntaksfejl.

Design

IDE'en er bygget op ved brug af IronPython og microsoft.scripting. For at kunne køre en skrevet sekvens, bruges der en IronPython script engine. Denne er blevet implementeret via singleton pattern, da den samme engine skal bruges flere steder i systemet. Denne engine benytter en string, fra en RichTextBox i ProgramsWindow klassen, og denne oversættes fra python kode til C# kode der kan eksekveres på robotten. Til IDE'en er der implementeret en SyntaxisOk() funktion som tjekker det fornævnte script for python syntaks fejl. Denne bliver brugt både når der trykkes på Compile knappen, samt når programmet bliver eksekveret. Der er blevet implementeret en række funktioner som er mulig at kalde igennem IDE'en, disse funktioner er implementeret i InterfaceRobotClass klassen. funktionerne bliver brugt til styring af robotten, bla. OpenClaw(), GetWeight() og MoveByAxis() for mere information om disse funktioner, se *SBS_IDE-funktioner* documentet. For at kunne kalde disse funktioner bliver der brugt en Dictionary<string, object>. Dette giver mulighed for IronPython enginen at oversætte en string, i dette tilfælde "Robot/ "robot" til et givet objekt, igen i dette tilfælde et objekt af typen InterfaceRobotClass. Når IronPython enginen scripter den givne string, vil den oversætte "Robot", fra en string til et objekt af typen InterfaceRobotClass. Dette giver mulighed for, at i den overnævnte string, at skrive f.eks Robot.OpenClaw, dette vil kalde OpenClaw funktionen implementeret i InterfaceRobotClass

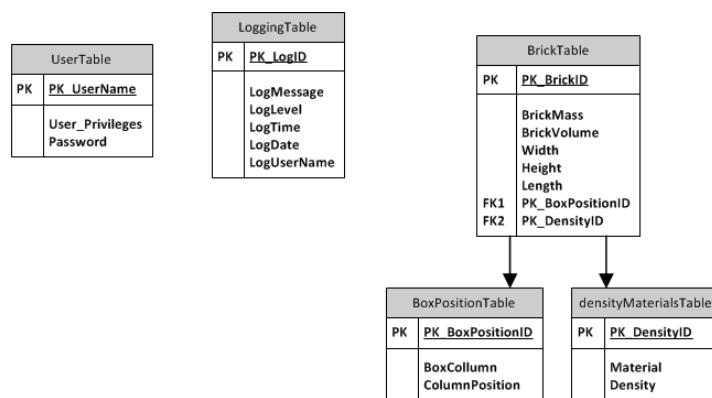
9 DATA VIEW

9.1 Data model

Til dette projekt er der blevet gjort brug af en relationel database til brug af persistent data lagring. Selve databasen er lavet med MS SQL Server og oprettet på webhotel10.ihb.dk. Diagrammer kan findes i fuld størrelse under Diagrammer/Database Diagrammer

9.1.1 Design af database

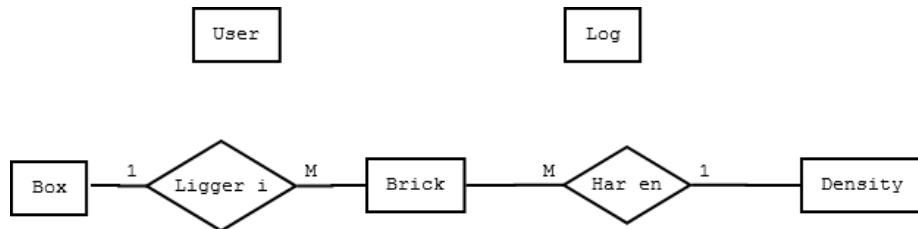
Hoved komponenten i datalagringen i dette projekt består i, at gemme data om en klods, og linke den til en position i en boks samt en densitet/materiale. Hertil består persisteringen også af brugernavne og loggingindlæg. På figur 56 ses et UML OO diagram over databasen



Figur 56: UML OO diagram over databasen

Som man kan se, har et indlæg i BrickTable (Hvor klodsernes beskrivelse ligger) en foreign key til BoxPositionTable (Hvor klodserne ligger) samt DensityMaterialTable(Hvor klodsernes densitet er beskrevet). UserTable(Hvor information om brugere af systemet ligger) og LoggingTable(Hvor alle system events gemmes) havde en relation i de første sprints, hvor LoggingTable havde en foreign key til UserTable, men dette blev kasseret da vi indså, at man i så fald ikke kunne fjerne en bruger fra systemet uden at slette alle brugerens logging indlæg. På figur 57 ses et ER diagram over databasen

Vi har udelukkende en-til-mange forhold i systemet, da flere klodser kan ligge i en kasse, men en klods kan ikke ligge i flere kasser. En Klods har desuden kun 1 densitet, men flere klodser kan godt have samme densitet.



Figur 57: ER Diagram over databasen

Databasen er på anden normalform da vi ikke har nogen composite keys. Der eksisterer dog transitive afhængigheder så den ikke er på tredje normalform. I BrickTable er Volume transitivt afhængig af Height, Length og Width. Det er dog blevet valgt, at databasen kun skal være på anden normalform, og databasen overholder derfor kravende.

9.1.2 Triggers og stored procedures

Til implementeringen blev der lavet flere triggers og en funktion.

Triggers:

Flere tabeller er blevet lavet ved hjælp af en identity column som primary key. Dette skaber visse problemer i tabellen, navnligt når man prøver at slette en række fra databasen. En delete statement vil lave et hul i primary key kolonnen som ikke er synderligt kønt, og det er ikke muligt at lave en update statement på kolonnen, da det ikke kan gøres på identity kolonne. Der må insertes hvis man midlertidigt slår identity insert til, så ved hjælp af en on delete trigger blev det gjort muligt at lave en update on delete trigger. På kodeudsnit 22 set et kodeudsnit for en update on delete trigger på LoggingTable

Kodeudsnit 22: SQL trigger for update af en identity kolonne

```

1 Create trigger [dbo].[LoggingTrigger]
2 ON [dbo].[LoggingTable]
3 FOR Delete
4 as
5
6 Declare @DeletedID bigint
7 Declare @TempTable Table (ID bigint, msg varchar(500), lvl nchar←
    (50), ltime nchar(50), ldate nchar(50), luser nchar(50))

```

```

8 Declare @Reseed int
9 Set @DeletedID = (Select MIN(PK_LogID) From deleted)
10
11 Insert into @TempTable select * from LoggingTable where PK_LogID <
    > @DeletedID
12 update @TempTable set ID = ID - 1
13 delete from LoggingTable where PK_LogID > @DeletedID
14
15 SET IDENTITY_INSERT LoggingTable ON
16 insert into LoggingTable (PK_LogID, LogMessage, LogLevel, LogTime, <
    LogDate, LogUserName) select * from @TempTable
17
18 SET IDENTITY_INSERT LoggingTable OFF
19 set @Reseed = (Select MAX(PK_LogID) from LoggingTable)
20 if @Reseed >= 0
21 dbcc checkident(LoggingTable, reseed, @Reseed)
22 else
23 dbcc checkident(LoggingTable, reseed, 0)
24 GO

```

Det første der sker efter erklæringen af variable er at DeletedID bliver sat til den mindste PKLogID af deleted. Deleted er en tabel der indeholder det slettede data. Herefter bliver alle rækker der har et større PKLogID end DeletedID, dvs alle rækker efter de slettede, sat ind i en midlertidig tabel. Så inkrementeres PKLogID i den midlertidige tabel med en, hvorefter alle rækker med større indlæg end de slettede, slettes fra tabellen. Så enables indsætning på en identity kolonne og den midlertidige tabel indsættes i LoggingTable. Identity reseedes til sidst til den største værdi af PKLogID i tabellen. Hvis tabellen er tom reseedes der til nul. Med dette er der blevet opnået en update statement på en identity kolonne. Dette sker ved samtlige tabeller der har en identity kolonne.

Triggeren for BrickTable har dog et par tilføjelser. Når man sletter en klods så er det ikke nok bare at slette kloden og opdatere primary key'en. Den række i BoxPositionTable som BrickTable refererer til skal også slettes og opdateres. Dette gøres ved at gemme foreign keyen over til BoxPositionTable fra det slettede, indsætte et dummy indlæg i BoxPositionTable og derefter slette den række som den slettede BrickTable række refererede til. Herefter slettes den række der har en primary key der stemmer overens med den slettede foreign key fra BrickTable, hvilket vil køre BoxPositionTable update on delete trigger. BrickTables update on delete fortsætter, og til sidst trækkes der en fra BoxPositionTables foreign key i BrickTable og dummy rækken i BoxPositionTable slettes.

Functions:

Den måde databasentilgangen er lavet på gør, at generel tilgang kan laves med funktioner på C# siden, men der er et enkelt sted hvor det er nødvendigt at bruge en lang join funktion. Denne funktion er specifik for denne join, dvs. tabeller og kolonner er hardcodet ind i join funktionen på sql-siden. På kodeudsnit 23 ses koden bag funktionen. Den skal bruges fordi data om kloden (ie. Densitet og placering i boksen) gemmes i andre tabeller end klodstabellen.

Kodeudsnit 23: SQL funktion for inner join for brick data

```

1 create function [dbo].[GetTotalBrickData]()
2 returns @ReturnTable table
3 (
4     BoxColumn bigint, ColumnPosition bigint, Length float,
5     Height float, Width float, BrickVolume float,
6     BrickMass float, Material nchar(50), Density float
7 )
8 as
9 begin
10    insert @ReturnTable
11    SELECT
12        BoxPositionTable.BoxColumn, BoxPositionTable.ColumnPosition,
13        BrickTable.Length, BrickTable.Height, BrickTable.Width,
14        BrickTable.BrickVolume, BrickTable.BrickMass,
15        DensityMaterialsTable.Material, DensityMaterialsTable.Density
16
17    FROM BoxPositionTable
18    INNER JOIN BrickTable ON
19        BoxPositionTable.PK_BoxPositionID = BrickTable.FK_BoxPositionID
20    INNER JOIN DensityMaterialsTable ON
21        BrickTable.FK_DensityID = DensityMaterialsTable.PK_DensityID
22
23    Return
24 end
25 GO

```

Denne funktion skal bruges ved visning af data om en klods på systemets GUI. Der skal altså bruges data fra flere tabeller end en, og frem for at lave flere select statements i

træk, blev det valgt at lave en specifik funktion til at håndtere denne datahentning.

9.2 Implementering af persistens

Persisteringen af data består af to hoveddele. En facade til den direkte tilgang og en besked kø der lavet efter observer design mønsteret som bliver beskrevet i implementeringsviewet, afsnit 8.2.7-Observer Pattern

For at tilgå databasen på C# siden er der blevet brugt facade designmønsteret. Tilgangen til databasen sker altså igennem én klasse, så vi bevarer lav kobling. Til selve forbindelsen blev der brugt SQLCommands. Dette er en del af .Net frameworkt. Forbindelsen ligger hardcodet i facaden, dvs. at det ikke er op til brugeren af systemet, at definere hvilken database man tilgår. Denne forbindelse består af en streng med bruger id, password, server, database og diverse optionelle valgmuligheder som f.eks. Connection Timeout som i dette tilfælde bestemmer, hvor lang tid systemet maksimalt må bruge på at skabe forbindelse. På kodeudsnit 24 kan man se hvordan forbindelsen bliver defineret og oprettet.

Kodeudsnit 24: Definition and opening of connection to database

```

1 private const string UserID      = "user id=F12I4PRJ4Gr5;" ;
2 private const string Password   = "password=F12I4PRJ4Gr5;" ;
3 private const string Server     = "server=webhotel10.ihb.dk;" ;
4 private const string Database   = "database=F12I4PRJ4Gr5;" ;
5 private const string ConnectionTimeOut = "Connection TimeOut=2" ;
6
7 private readonly SqlConnection _myConnection = new SqlConnection(
8
9
10
11
12
13
14
15
16 if (_myConnection.State != ConnectionState.Open)
17     _myConnection.Open();

```

Forbindelsen bliver brugt ved at eksekvere en SQL streng fra C# siden, altså en SQL statement skrevet i en string. Det er blevet forsøgt at gøre disse commands generelle,

så alt efter hvad man vil på en vilkårlig tabel, kan man tilgå en generel funktion, som specificeres via parametre. Når en funktion, der henter noget fra databasen køres, vil data returneres via en SQLDatareader som funktionen, ExecuteReader(), returnerer. Nedunder, på kodeudsnit 25 kan man se, hvordan en simpel SELECT statement bliver brugt, kørt og hvordan den returnerer data.

Kodeudsnit 25: Definition of a Select statement with a where clause

```

1 public List<string> SelectFrom(string columnName,
2                                 string tableName,
3                                 string whereColumnName,
4                                 string target)
5 {
6     if (_myConnection.State != ConnectionState.Open)
7         _myConnection.Open();
8
9     var back = new List<string>();
10    var command = new SqlCommand(string.Format(
11        "SELECT {0} FROM {1} WHERE {2}='{3}'",
12        columnName, tableName,
13        whereColumnName,
14        target),
15        _myConnection);
16
17    var read = command.ExecuteReader();
18    while (read.Read())
19    {
20        back.Add(read[columnName].ToString().TrimEnd(' '));
21
22    }
23    read.Close();
24    _myConnection.Close();
25    return back;
26 }

```

Når man kalder Execute reader bliver den specificerede streng kørt på databasen. Hvis operationen på databasen returnerer noget vil dette blive hentet via read[columnName] hvor columnName er den kolonne man vil hente fra. Hvis der hentes flere kolonne fra databasen vil disse blive gemt i den kolonne man specificerer. Database tilgangen sker på samme måde ved samtlige funktioner, med den undtagelse, at der ikke returneres noget på de operationer på databasen, der ikke skal returnere noget f.eks. insert- eller

update statements. Disse operationer er de eneste der forekommer på databasen, udover de triggers og functions der også kan køres.

10 GENERELLE DESIGNBESLUTNINGER

10.1 Arkitektur mål og begrænsninger

Der er ikke mange funktionelle krav at indhente fra produktoplæget, og derfor har disse heller ikke haft den store indflydelse på arkitekturen. Dog er systemet implementeret til brug på windows pc, og brugergrænsefladen er baseret på WPF, og dette har en vis betydning for arkitekturen. Brugen af ScoreBot-robotarmen og USBC.dll biblioteket betyder også at arkitekturen er udformet omkring disse ydre omstændigheder. For gruppens sider er der fastsat en række udviklingskrav. F.eks. bruges Scrum som den primære udviklingsprocess og alle diagrammer er lavet i visual paradigm.

10.2 Arkitektur mønstre

Nedenstående liste er en opremsning af de standard arkitekturmønstre, der er anvendt i systemet. Under listen er de forskellige mønstre dokumenteret.

- Model View ViewModel (MVVM)
- Singleton
- Adapter
- Strategy
- Facade
- Publish-Subscribe
- Mediator

Model View ViewModel (MVVM)

MVVM-mønstret bruges til at opretholde en overskuelig arkitektur i systemets implementering. Det sørger bl.a. for, at det er muligt at udføre softwaretest på alt kode. Koden opdeles i fire lag; *View*, *ViewModel*, *Model* og *DAL*¹⁰.

Se http://en.wikipedia.org/wiki/Model_View_Model for mere dokumentation.

¹⁰Data acces Layer

Singleton

Singleton bruges til at oprette ét objekt af en instance. Når først objektet er oprettet vil andre, der forsøger at oprette det, få returneret det samme objekt, som der blev oprettet først.

Se afsnit 26.5 i *Applying UML and Patterns* for yderlige information.

Adapter

Adapter-mønstret bruges til at oprette et interface og en klasse til en anden klasse, således den anden klasses funktioner kan tilgås fra interfacet.

Se afsnit 26.1 i *Applying UML and Patterns* for yderlig information.

Strategy

Strategy-mønstret bruges til at udskifte funktionalitet på run-time, således den samme funktion kan kaldes, men hentes fra forskellige klasser.

Se afsnit 26.7 i *Applying UML and Patterns* for yderlig information.

Facade

Facade-mønstret bruges til at pakke flere sammenhængende klasser ind i en .dll-fil, således det kan tilgås af vilkårligt andre klasser med samme funktionalitet.

Se afsnit 26.9 i *Applying UML and Patterns* for yderlig information.

Publish-Subscribe

Publish-Subscribe-mønstret bruges til at få flere klasser til at videregive information, uden de skal vente på en returkald. På denne måde kan andre klasser tilgå informationen, uden klassen der afgav den informationen kender de(n) der modtager.

Se afsnit 26.9 i *Applying UML and Patterns* for yderlig information.

Mediator

Mediator-mønstret opretter en klasse til at sende information imellem to andre klasser, således kaldet kan testes.

Se http://en.wikipedia.org/wiki/Mediator_pattern

10.3 Generelle brugergrænsefladeregler

Kravene til brugergrænsefladen kan deles op i to kategorier:

- Arkitekturspecifikke
- Udseendesspecifikke

Begge er beskrevet herunder

10.3.1 Arkitekturspecifikke

Brugergrænsefalden er baseret på MVVM designmønsteret, som beskrevet i afsnit 10.2 - *Arkitektur mønstre*. Dette er valgt af to grunde; Det betyder lavere kobling mellem model og view, og brugergrænsefalden kan derfor udskriftes uden større problemer. Desuden betyder det, at det er markant nemmere at teste kode der ligger tæt op af brugergrænsefladen. Viewet er lavet udelukkende i XAML, og i viewmodellen ligger en række properties, som omformer data fra modellen til viewet, samt en række commands der kalder ned i modellens logik. Det var ideologen at der absolut ikke måtte placeres kode i den såkaldte "Code-behind"fil. Dette var dog ikke helt muligt når der f.eks. skulle åbnes nye vinduer. Derfor blev det besluttet, at sådanne operation skulle baseres på Mediator Pattern (nærmere beskrevet i afsnit 10.2 - *Arkitektur mønstre*). Dette betød at der stadig blev holdt en forholdsvis lav kobling.

10.3.2 Udseendesspecifikke

For at holde et strengt ens udseende på tværs af hele brugergrænsefladen blev det besluttet at bruge WPF-themes fra codeplex. Dette påvirker at alle kontroller altid har samme udseende.

10.4 Exception og fejlhåndtering

10.4.1 Exception i database tilgangen

Fejlhåndtering er meget kritisk i databasetilgangen, da det er plausibelt at tilgangen til internet, og dermed tilgangen til databasen, kan forsvinde i løbet af systemets forløb. Når en besked til databasen skal håndteres sker dette i en try block. Hvis internet er forsvindet vil database tilgangen kaste SQLException. Denne vil blive grebet af en catch blok, som vil lave en besked om at forbindelsen er tabt. Dette sker dog kun hvis det er første gang denne exception bliver kastet, siden forbindelsen er gået tabt. Når forbindelsen reetableres ligges der en besked i køen om at forbindelsen er reetableret. Hvis tråden er gået ind i sin kritiske sektion, det vil sige når der ligger en besked i køen, som vil hente noget fra databasen, så kræves det at der er forbindelse til databasen. Dvs. hvis der kastes en SQLException vil brugeren have mulighed for at forsøge at skabe forbindelse igen, eller lukke programmet ned. Hvis brugeren vælger at lukke programmet ned vil samtlige beskeder til databasen gå tabt. Det antages at brugeren selv kan genoprette forbindelsen til internettet, eller ved hvad der skal gøres i et sådan tilfælde.

10.4.2 Exception i serial communication

Weight kommunikerer med systemet over en seriell forbindelse. Denne forbindelse kan være meget ustabil og det har derfor været nødsaget at håndtere forbindelsen i en try blok. Hvis en exception kastes vil denne gribes og de funktioner der returnerer en bool vil returnere false og den funktion der returnerer en string vil returnere false i en string. Så længe bare en af bool-funktionerne returnerer false vil hele kommunikationen starte forfra. Hvis string-funktionen returnerer false sættes result til 1 og det returneres til vægten.

10.4.3 Exception i Robot funktionerne

I robot funktionerne sker der en hel del exception handling. RobotProgramFunctions klassen er bygget op af tilgange til mange forskellige eksterne hardware dele. Hvis nogen af dem returnere false vil det altid resultere i., at kloden smides ud, og robotten returnerer til startposition. Dette kan f.eks. ske hvis vægten returnerer false.

10.5 Implementeringssprog og værktøjer

Herunder beskrives udviklingsværktøjer. Afsnittet er ikke ment som et udtømmende afsnit, og derfor vil alle værktøjer der har været brugt under projektet, ikke nødvendigvis beskrives.

10.5.1 C

Kode på ATmega16 er skrevet i det funktionsbaserede programmeringssprog C. Koden er simpel og hardwarenær.

10.5.2 C++

I projektet er der ikke skrevet noget egentlig kode i C++, dog er det udleverede USBC-bibliotek skrevet i unmanaged C++.

10.5.3 C#

Selve programkoden er udført i C#, ligesom der er brugt .NET-biblioteker og microsoft WPF.

10.5.4 Microsoft Visual Studio 2010

Den egentlige kode er blevet lavet og vedligeholdt i visual studio.

10.5.5 Visual paradigm

De fleste UML-baserede diagrammer er forsøgt lavet i Visual paradigm. Dog er enkelte digrammer lavet i Enterprice Architect.

10.5.6 AnkhSVN og tortoiseSVN

Til versionsstyring er der brugt SVN. AnkhSVN er brugt som plugin til visual studio, mens tortoiseSVN er brugt som shell-extention.

10.5.7 TexMaker

Alt dokumentation er skrevet i L^AT_EX og som editerings-værktøj er der brugt TexMaker.

10.5.8 MSSQL server management studio

Dette program er brugt til at opsætte og ændre databasen

10.6 Implementeringsbiblioteker

Da systemet er baseret på .NET, og derfor er der selvfølgelig brugt adskellige .NET biblioteker. Det er langt fra alle der vil blive beskrevet i dette afsnit, hvor kun de vigtigste er forklaret.

- System.IO.Ports - Bibliotek, som indeholder funktioner der bruges til kommunikation over den serielle port. Se klassen RobotProjekt.Weights.Weight.
- WPF.Themes - Bibliotek der indeholder brugergrænsefladens tema.
- Microsoft.Scripting - Er blevet brugt til at lave en scripting-engine til brug i IDEen. Se komponentbeskrivelsen af IDEen under afsnit 8.
- Galasoft.MvvmLight - Bibliotek fra Galasoft, som bruges til MVVM funktionalitet. Fx indeholder det en implementering af RelayCommand, og et Message System.
- NUnit.Framework - NUnit frameworket, som bruges til automatiserede test.

11 STØRRELSE OG YDELSE

I dette afsnit er der angivet de kritiske størrelser og ydelsesparametre for systemet. En begrænsning for det aktuelle system er sorteringen, der maksimum kan håndtere fem klodselementer i hvert materialetyperum. Det er nødvendigt at elementerne bliver placeret på den mest hensigtsmæssige og optimale måde, ellers kan der kun være fire elementer. Dog kan denne begrænsning altid videreudvikles samt håndteres i en fremtidig version af systemet.

Før systemet kan udføre en programsekvens i "run-vinduet", skal der kaldes en "home"-funktion på robotten der kalibrerer dens akser. Denne funktion er lagt i en baggrundstråd for sig selv, for at så vidt muligt ikke at sløve brugergrænsefladen. Dog trækker funktionen mange ressourcer, så GUI'en bliver en anelse langsom i betrækket. Hvis systemet bliver eksekveret på den udstedte computer, kan man forvente at afsætte hele eftermiddagen på at få kørt et program.

Performance for systemet:

Lagerforbruget på disken omkring 20 megabyte.

Der benyttes 32700 byte kb memory og 50% cpu ved et stressende while-loop. I idlemode er CPU'en 0%. Middelsvartiden for programmet ligger på et acceptabelt og normalt niveau.

12 KVALITET

12.1 Brugervenlighed

Det er tilstræbt at holde systemet simpelt og brugervenligt som muligt. Dette er gjort ved at lave forholdsvis store knapper med sigende tekst.

12.2 Pålidelighed

Der er lidt problemer mht. systemets pålidelighed, da der ofte opstår problemer mht. at home robotten. Men da dette enten skyldes funktioner i .dll-filen eller selve hardwaren, har det ikke været muligt fuldstændig at udrette dette problem. Til gengæld er systemet lavet således, at brugeren løbende har mulighed for at home robotten, hvis der skulle opstå fejl. Således er det ikke nødvendigt at genstarte hele programmet, hvis der skulle opstå fejl på denne front. Desuden at der lavet en del fejlhåndtering, beskrevet nærmere under afsnit *10.4 Exceptions og fejlhåndtering*.

12.3 Integritet

Den fysiske nødstopknap afbryder strømmen til systemet, og dette resulterer naturligvis i at robotten stopper i sin respektive position. Desuden er det også muligt at stoppe en igaangværende sorteringsmekaniske på pause, og starte denne igen.

13 OVERSÆTTELSE

I dette afsnit beskrives hvordan man kommer fra kildeteksten til objektkoden, altså til et program, der er klart til at blive kørt. Programmet er pre-compiled og kræver derfor kun en installering - se næste punkt.

13.1 Oversættelses-hardware

Der er ikke behov for megen hardware, for at få oversættet programmet. En computer som programmet kan køres på. Hvis man ønsker at benytte sig af den fysiske vægt, er det nødvendigt at have en Seriel COM port i sin computer. Der kan hvis man ikke har en COM port, anvendes en serielcom port til usb converter. Dette bruges naturligvis til at lægge den oversatte kode over på microcontrolleren.

13.2 Oversættelses-software

Den software, der er nødvendig for at oversætte programmet er compileren, assembleren, linkeren i Microsoft Visual Studio samt CodeVision.

13.3 Installation

Den arbejdsgang, der skal følges, for at det oversatte program kan bringes til at køre i de rette omgivelser er at køre programinstallationen. Derefter køres programmet fra skrivebordet, hvor en programgenvej er blevet oprettet.

14 KØRSEL

I dette program beskrives hvordan programmet køres. Når programmet skal køres, er der visse hardware og software forudsætninger, der er gældende. Dette er beskrevet i følgende afsnit.

14.1 Kørsels-hardware

For at udnytte systemets fulde funktionalitet skal programmet være opkoblet til Scorebot-Er 4u og en vægt. Dog skal det siges, at man kan simulere disse to elementer, så man er dem foruden, selvfølgelig uden at der kan sorteres klodser i det virkelige problemdomæne. Det er ydermere nødvendigt at være forbundet til en database, hvor brugerdata ligger. Hvis det ikke er muligt kan der ikke logges ind på systemet. Databasen kan godt simuleres efter login.

14.1.1 Opstilling

Vægtenprintet forbindes over med det 10-polede parallelkabel til STK500-kittets port A. 5-pin DIN stikket forbindes til vægten. Mellem STK-kittets Spare-serialport og computerens USB forbindes prolific Seriel-to-usb-converter og denne indstilles som COM20. Desuden skal vægtprint tilsluttes +5V, -5V og ground på de respektive bananstik. PCen skal have netværks forbindelse, og er der ikke tilsluttet til IHA's net skal der være forbindelse til skolens VPN. USB-stikket fra Scorebots USB-controller skal være tilsluttet en af PCens USB-porte.

Er transportbåndets position ikke indstillet, skal transportbåndet placeres i yderste venstre hjørne af bordet.

14.2 Kørsels-software

For at køre softwaren kræves installationen af programmet. Denne startes ved at køre filen *SilverBulletSort.msi*, der er udleveret på den medfølgende CD-ROM.

Herefter udpakkes diverse .dll-filer, samt programfiler, i mappen *Programmer/Awesome Sort/RobotInstaller/SilverBullet/*

For softwarekrav til Client PC henvises til afsnit 7.3.1: *Node 1 beskrivelse* hvor dette er beskrevet.

14.3 Start, genstart og stop

Programmet startes ved double klik på at skrivebordsikonet, hvorefter Silver Bullet Sort programmet åbnes, og giver brugeren mulighed for at logge ind.

Programmet lukkes som ethvert andet program, nemlig ved at trykke på det røde kryds i højre hjørne, hvorefter brugeren bliver spurgt, om han er sikker på, at programmet skal lukkes ned. Programmet kan naturligvis også lukkes ved at afbryde strømmen til computeren, men dette frarådes.

14.4 Fejludskrifter

Systemets fejludskrifter består hovedsagligt af logging indlæg til databasen, hvor logging niveauet er "ERROR" og i form af beskedbokse der fremkommer på skærmen. Nedenfor følger en liste af fejl. Der beskrives kort hvordan de forekommer og hvordan de kan udbedres. Hvis der i overskriften for fejlen står LOG til sidst er det en log besked til databasen, hvis det står MSG forekommer fejlen i en beskedboks.

- **Initialization of the robot failed! LOG**

- Fejlen forekommer når robotten ikke kan ikke findes eller ikke vil operere når den initialiseres. Brugeren kan ikke udbedre det direkte fra programmet, da det er en hardware fejl.

- **Homing failed! LOG**

- Fejlen forekommer når robotten under homing proceduren fejler. Denne fejl forekommer tilfældigt af og til og kan ikke ubedres fra programmet da det er en hardware fejl. Brugeren kan forsøge at kører homing igen indtil dette lykkes.

- **Robot was not able to move! LOG**

- Fejlen forekommer når robotten er blokeret eller af anden årsag ikke kan bevæge sig. For at udbedre fejlen kan brugeren fjerne hvad end der blokkerer robottens vej.

- **Something went wrong with the weight. Weight not obtained. LOG**

– Fejlen forekommer når vægten malfunktionerer, og der fra den serielle port modtager "failed". Brugeren kan udbedre dette ved at konfirmere at forbindelser til vægten er sluttet rigtigt til, samt kontrollere at spændings- og ground-forbindelser er tilsluttet rigtigt.

- **Could not find the materialtype. LOG**

– Fejlen forekommer når den udregnede densitet ikke kan parres med en materialtype på databasen. Dette kan ske hvis der sker en fejllæsning fra en siderne eller på kloksen masse. Det kan forsøges at placerer kloksen på transportbåndet og starte sorteringen forfra.

- **Could not find the total bricklength. LOG**

– Fejlen forekommer hvis den samlede længde af alle kloserne af den givne type i databasen, ikke kan hentes. Brugeren kan ikke udbedre dette. Kloksen udsmides og sorteringen fortsætter med en ny klods.

- **No room for Brick in the box! LOG**

– Fejlen forekommer når der ikke er plads til den klods der skal sorteres i det specifikke rum. Brugeren kan udbedre dette ved at fjerne kloser fra det gældende rum.

- **Could not insert the box position into the database. LOG**

– Fejlen forekommer når kloksens position i boksen ikke kan skrives til databasen. Kloksen må sorteres igen. Brugeren kan verificerer at der er forbindelse til databasen.

- **Could not insert the brick into the database. LOG**

– Fejlen forekommer når kloksen og dens tilhørende data ikke kan skrives til databasen. Brugeren kan ikke udbedre dette. Kloksen må sorteres igen.

- **Lost connection to database LOG**

- Fejlen forekommer hvis forbindelsen til databasen forsvinder. Brugeren kan udbedre dette ved at sørge for, at der forbindelse til internettet, og manuelt undersøge om databasen kan tilgås.

- **Database Connection can not be established. To Retry Press 'Yes'. To Close the program press 'No',, MSG**

- Fejlen forekommer når forbindelsen til databasen er forsvundet og der er en besked i køen der skal hente fra databasen. Brugeren kan udbedre dette ved at sørge for at der er forbindelse til internettet, og manuelt undersøge om databasen kan tilgås.

- **Compile failed. MSG**

- Fejlen forekommer når der er syntax fejl i koden skrevet i IDE'en. Brugeren kan udbedre dette ved at rette fejlene i programmet.

- **The robot is not online, or has not been homed! MSG**

- Fejlen forekommer hvis robottens initialiseringen eller homingen fejler på robotten. Brugeren kan ikke udbedre dette. Det kan dog forsøges at home igen.

- **Please input a value in all fields. MSG**

- Fejlen forekommer når man vil indstille transportbåndets position og alle felter ikke er udfyldt. Brugeren kan udbedre dette ved at udfylde alle felter

- **All fields must contain a real number. MSG**

- Fejlen forekommer når man vil indstille transportbåndets position og nogle felter indeholder andet end tal. Brugeren kan ubedre dette ved udelukkende at skrive tal i felterne.

- **The data input is wrong. Only real numbers. MSG**

- Fejlen forekommer når man vil indstille transportbåndets position og tallet bliver for stort. Brugeren kan udbedre dette ved at skrive realistiske tal i felterne

- **Program is in progress! MSG**

- Fejlen forekommer når man vil starte et program, hvor et andet program allerede kører. Brugeren kan udbedre dette ved at vente til programmet er kørt færdigt, eller stoppe det kørende program.

- **Please choose a program first! MSG**

- Fejlen forekommer når man trykker på start uden at have valgt et program. Brugeren kan udbedre dette ved at vælge et program fra listen.

- **Can't load a file, that is being used by the robot! MSG**

- Fejlen forekommer når man vil redigere i et program som robotten er igang med at eksekvere. Brugeren kan udbedre dette ved at vente til programmet er kørt færdigt, eller stoppe det kørende program.

- **Cannot load a Standardprogram. MSG**

- Fejlen forekommer når standardprogrammet prøves at indlæses. Kan ikke udbedres da standardprogrammet ikke kan indlæses.

- **Cannot load file! MSG**

- Fejlen forekommer når den specificerede fil ikke kan findes på den givne sti. Brugeren kan udbedre fejlen ved at undersøge om han har specificeret den rigtige sti og om filen rent faktisk eksisterer.

- **Can't overwrite standardprogram, sorry. MSG**

- Fejlen forekommer når brugeren prøver at overskrive standard programmet fra IDE'en. Denne fejl kan udbedres ved at vælge et andet filnavn.

- **Please name the program. MSG**

- Fejlen forekommer når brugeren prøver at gemme et program skrevet i IDE'en uden at specificere et navn på filen. Brugeren kan udbedre dette ved at skrive et filnavn.

- **Can't delete standardprogram. MSG**

- Fejlen forekommer når brugeren prøver at slette standard programmet fra IDE'en. Denne fejl kan udbedres ved at vælge en anden fil.

- **Can't delete this file! MSG**

- Fejlen forekommer når brugeren prøver at slette et program skrevet i IDE'en uden at specificere et navn på filen. Brugeren kan udbedre dette ved at vælge en eksisterende fil.

- **Invalid username or password! MSG**

- Fejlen forekommer på loginskærmen hvis det skrevne brugernavn og password ikke passer sammen. Dette kan udbedres ved at undersøge om man har skrevet det rigtige brugernavn eller password.

- **Can't delete default material! MSG**

- Fejlen forekommer når brugeren prøver at slette en materialetyper, der er indskrevet som default. Denne fejl kan udbedres ved at slette en anden materiale type end de fem, der er default materialetyper.

- **Robot have not been homed! MSG**

- Fejlen forekommer når brugeren prøver at starte et program uden at robotten er blevet homet. Denne fejl kan udbedres ved at home robotten før et program startes.

Bilag

Nedenfor er mappestrukturen i bilagsmappen listet.

DLL filer

1. USBC.dll

Diagrammer

1. Database diagrammer
2. Domænemodel
3. Generelt
4. Klasser og lavdelings-diagrammer
5. Sekvensdiagrammer
 - (a) UseCase Realiseringer
 - (b) ProcessTask view
6. Detaljerede sekvensdiagrammer
 - (a) UseCase Realiseringer
 - (b) RobotProgramFunctions
7. UseCase Diagrammer

Billeder

1. Brugergrænsefladen
2. Placering af transportbånd
3. Scrum
4. Weight

Reference dokumenter