Forside

# Del I: Prolog

# 1.1 Synopsis

Denne rapport omhandler udvikling af Centrallagerstyringssystemet CSS, som er et system der skal være med til at effektivisere og automatisere arbejdsprocesser på et centrallager. Systemet kan modtage og oprette ordre direkte fra detailbutikker uden interaktion fra brugere i systemet. Administrator og manager på systemet kan efterfølgende se og ekspedere de ordrer, der er oprettet i systemet. Arbejdsprocesser i forbindelse med at sætte nye varer på lager og hente dem ud til afsending sker automatiseret ved hjælp af en robot.

I projektet er der anvendt PCMEF+ arkitektur samt brugsmønstrene *Composite,* Observer og *Adapter*. Implementeringen er forgået i Java og SQL ved hjælp af modellering med UML-artefakter. Kommunikationen mellem klient og server er foregået ved socket programmering i forbindelse med detailbutikkerne og RS232 Seriel kommunikation til robotten. Unified Process er den anvendte procesmodel.

# 1.2 Forord

Rapport er et resultat af 3. semesters projekt på Diplomingeniøruddannelsen i Informations- og Kommunikationsteknologi ved Det Tekniske Fakultet hos Syd Dansk Universitet.

I Projektforløbet har projektgruppen arbejdet med de områder, metoder og teknikker der er blevet undervist i på de tre første semestre på uddannelsen. Vi lægger, i dette 3. semester projekt, vægt på de kompetencemål, der er sat for de enkelte fag.

Målgruppen for denne rapport er projektvejleder Steffen Peter Skov, censor der skal foretage bedømmelse af rapporten og evt. medstuderende på samme uddannelse.

Projektgruppen vil gerne takke Steffen Peter Skov for konstruktiv kritik og vejledning i projektforløbet. Vi vil ligeledes gerne takke Lone Borgersen og Jørgen Jeppe Madsen for til afklaring af problemstillinger undervejs i forløbet.

Indholdsfortegnelse

[Del I: Prolog 2](#_Toc343552104)

[1.1 Synopsis 3](#_Toc343552105)

[1.2 Forord 3](#_Toc343552106)

[1.3 Læsevejledning 7](#_Toc343552107)

[1.4 Indledning 8](#_Toc343552108)

[1.4.1 Beskrivelse 8](#_Toc343552109)

[1.4.2 Formål 9](#_Toc343552110)

[1.4.3 Mål 9](#_Toc343552111)

[1.4.4 Inddragelse af fagområder 9](#_Toc343552112)

[Del II: Procesmodel 10](#_Toc343552113)

[2.1 Projektværktøjer 11](#_Toc343552114)

[2.1.1 Rapportskrivning 11](#_Toc343552115)

[2.1.2 Udviklingsmiljø 11](#_Toc343552116)

[2.1.3 UML Modellering 11](#_Toc343552117)

[2.1.4 Database 11](#_Toc343552118)

[2.1.5 Versionsstyring 11](#_Toc343552119)

[2.2 Projektstyring 11](#_Toc343552120)

[2.2.1 Unified Process 11](#_Toc343552121)

[2.2.2 SCRUM 12](#_Toc343552122)

[2.3 Den overordnede projektplan 13](#_Toc343552123)

[2.4 Det faktiske forløb i projektet 13](#_Toc343552124)

[Del III: Resultater 15](#_Toc343552125)

[3.1 Vision 16](#_Toc343552126)

[3.2 Krav til systemet 17](#_Toc343552127)

[3.3 Aktørliste 18](#_Toc343552128)

[3.4 Use Cases 18](#_Toc343552129)

[3.4.1 Prioritering af Use cases 19](#_Toc343552130)

[3.4.2 High-level Use Cases 20](#_Toc343552131)

[3.4.3 Expanded Use Cases 21](#_Toc343552132)

[3.5 Klassediagram 22](#_Toc343552133)

[3.6 Systemsekvensdiagrammer og Systemoperationskontrakter 23](#_Toc343552134)

[3.7 Softwarearkitektur 25](#_Toc343552135)

[3.7.1 Designmål 25](#_Toc343552136)

[3.7.2 Arkitektur - PCMEF+ framework 25](#_Toc343552137)

[3.8.3 Anvendte designmønstre 29](#_Toc343552138)

[3.8 Detaljeret Design 34](#_Toc343552139)

[3.8.1 Design sekvensdiagrammer 34](#_Toc343552140)

[3.8.2 Persistens framework 37](#_Toc343552141)

[3.8.3 Singleton pattern 39](#_Toc343552142)

[3.9 Implementering 40](#_Toc343552143)

[3.9.1 Pakker og klasser i Java 40](#_Toc343552144)

[3.9.2 Singleton 40](#_Toc343552145)

[3.9.3 Observer 41](#_Toc343552146)

[3.9.4 Persistens framework 41](#_Toc343552147)

[3.10.1 Forbindelse mellem DSS og CSS 46](#_Toc343552148)

[3.10 Test 47](#_Toc343552149)

[3.10.1 Testtyper 47](#_Toc343552150)

[3.10.2 JUnit 48](#_Toc343552151)

[3.10.3 Test af CSS 48](#_Toc343552152)

[3.10.4 Evaluering af designmål 48](#_Toc343552153)

[3.11 Datakommunikation 49](#_Toc343552154)

[3.12 Styring regulering og overvågning 50](#_Toc343552155)

[3.12.1 PLC’en 50](#_Toc343552156)

[3.12.2 Protokol 50](#_Toc343552157)

[3.12.3 Implementering på CSS siden 50](#_Toc343552158)

[3.12.4 Implementering på RCS siden 51](#_Toc343552159)

[3.12.5 Beskrivelse af kommunikationen 52](#_Toc343552160)

[3.12.4 Problem med kommunikation med PLC 52](#_Toc343552161)

[3.13 Deployering 52](#_Toc343552162)

[Del IV: Epilog 54](#_Toc343552163)

[4.1 Konklusion 55](#_Toc343552164)

[4.2 Perspektivering 55](#_Toc343552165)

[4.3 Refleksioner 55](#_Toc343552166)

[4.3.1 Projektetablering 55](#_Toc343552167)

[4.3.2 Planlægning 55](#_Toc343552168)

[4.3.3 Gennemførsel 56](#_Toc343552169)

[4.3.4 Afrapportering 56](#_Toc343552170)

[4.5 Litteraturliste 56](#_Toc343552171)

[Bilag 57](#_Toc343552172)

[Appendix A - Projektgrundlag 58](#_Toc343552173)

[1. Projektgrundlag 58](#_Toc343552174)

[1.1 Interessenter i projektet 58](#_Toc343552175)

[Projektvejleder 58](#_Toc343552176)

[1.2 Projektgruppen 58](#_Toc343552177)

[1.3 Beskrivelse 58](#_Toc343552178)

[1.4 Formål 59](#_Toc343552179)

[1.5 Mål 59](#_Toc343552180)

[1.5.1 Produktmål 59](#_Toc343552181)

[1.5.2 Procesmål 59](#_Toc343552182)

[1.6 Inddragelse af fagområder 59](#_Toc343552183)

[1.7 Samarbejdet 59](#_Toc343552184)

[1.8 Dokumenter 60](#_Toc343552185)

[1.9 Ressourcer 60](#_Toc343552186)

[1.10 Projektstyring 60](#_Toc343552187)

[1.11 Versionsstyring 60](#_Toc343552188)

[1.12 Den overordnede projektplan 60](#_Toc343552189)

[1.13 Plan for inceptionsfasen 61](#_Toc343552190)

[1.14 Use-case Diagram 62](#_Toc343552191)

[Appendix B - High-level Use Cases 63](#_Toc343552192)

[Appendix C - Expanded Use Cases 64](#_Toc343552193)

[Appendix D - Systemsekvensdiagrammer og Systemoperationskontrakter 65](#_Toc343552194)

[Appendix E - Design sekvensdiagrammer 67](#_Toc343552195)

# 1.3 Læsevejledning

Rapporten er skrevet i sammenhæng og vil give bedst mening, hvis den læses i den rigtige rækkefølge fra start til slut.

Rapporten er delt op i 4 dele: Prolog, Procesmodel, Resultater og Epilog.

#### Figurer, tabeller og kodeeksempler i rapporten

I rapporten findes billeder, tabeller og kodeeksempler og de vil blive betegnet med enten *Figur*, *Tabel* eller *Kode* og et tilhørende nummer, der henviser til afsnit det tilhører og et tal for nummer i dette afsnit fx *Tabel 3.10.1.2* hvilket betyder tabel nummer to i afsnit 3.10.1.

#### Henvisninger

MANGLER

#### Begreber i rapporten

Der vil blive brugt mange begreber og forkortelser i rapporten, så for at læseren har samme forståelse af begreberne, som vi har haft under udarbejdelse, så er de listet i *Tabel 1.3.1* med navn og tilhørende beskrivelse.

Vi har valgt at begreber skrives på engelsk i rapporten.

|  |  |
| --- | --- |
| **Begreb** | **Beskrivelse** |
| ***CSS*** | Central Lager System |
| ***DSS*** | Detail Store System. Det system der sender ordre til *DSS*. |
| ***RCS*** | Robot Central System findes på det fysiske lager. Robotten placerer Items på lageret ud fra instrukser fra *CSS*. |
| ***Stock*** | Lageret der er inddelt i mange *StockPositions*. |
| ***StockPosition*** | Er en given plads på lageret, der kan holde én Item (kan godt være en palle eller en kasse). |
| ***Scanner*** | Findes på lageret og bruges når en vare ankommer til lageret. |
| ***Stregkode (Barkode)*** | Den kode der sidder på en vare fra dens producent. |
| ***OrderRequest*** | Når en Detailbutik sender en ordre modtages denne af *CSS.* |
| ***Order*** | En ordre med tilhørende Items oprettes i *CSS* ud fra en *orderRequest*. |
| ***Confirmation*** | Er en bekræftelse der sendes til *DSS* når en ordre er modtaget i *CSS*. |
| ***ItemType*** | Er en varetype, som en Item tilhører. Den kan fx være Gevalia kaffe Økologisk. |
| ***Item*** | Er én vare i systemet. Kan fx være én sofa, men også én kasse kaffe med 10 stk i. Det er altså den mindste enhed af en *ItemType,* der kan bestilles. |
| ***Administrator*** | Den person der har fuld adgang til systemet og har mulighed for at oprette og slette fra systemet. |
| **Manger** | Ham der er ansat på *Stock* og har til opgave at ekspederer ordrer. |

**Tabel 1.3.0.1**: Begrebstabel

#### CD-indhold

MANGLER

Den medfølgende cd indeholder følgende:

* Rapport.pdf
* Mødereferater

#### Afprøvning af program

Der vil bagerst i rapporten være en lomme med en dvd, der indeholder programmet. (indstallatione?).

MANGLER

# 1.4 Indledning

Rapporten har til formål at dokumentere udviklingen af et system, samt redegøre for de metoder og teknikker, der anvendes i forbindelse med projektarbejdet.

Rapporten vil udelukkende henvende sig til undervisere og censor, da opgavens problemformulering er fiktiv og ikke omhandler udvikling af et system til løsning af et konkret problem. Projektgruppen har sammen formuleret opgaven.

## 1.4.1 Beskrivelse

Central Stock System (CSS).

Et nyt centrallager skal implementere et system til styring af varer: Bestilling, Modtagelse, lager og udlevering samt grundlæggende bogholderi.

Bestilling af vare til lageret foretages hos eksterne leverandører og registreres, som ”bestilt” i systemet.

En vares rejse gennem centrallageret forløber som følger: Ved ankomst til varemodtagelsen (EUR paller) scannes varen ind med stregkode eller QR kode og køres i dybdereol. Et robotstyret lager sørger automatisk for at varen opbevares forsvarligt til den skal udleveres. Varer der skal kasseres skal scannes ud af systemet, som kasserede.

Når en butik ønsker at bestille varer fra lagret, sender de en bestilling til systemet.

Varer bestilt i systemet bliver hentet på lager og gjort klar til pakning. Ved udlevering bliver varen udskrevet fra lageret, og Systemet sørger for bogføring.

Systemet skal være indlysende at bruge for personalet(brugervenligt) og stabilt, det må bl.a. ikke gå ned på brugerfejl. Der skal laves backup en gang pr. døgn. Systemet skal være så uafhængig af platform som muligt.

## 1.4.2 Formål

At fremstille et System der kan varetage den grundlæggende funktionalitet for drift af et centrallager. Systemet skal medvirke til nedbringelse af driftsomkostninger gennem automatisering af arbejdsprocesser.

## 1.4.3 Mål

#### Produktmål

Målet for produktet er:

1. at få udviklet et system, der kan håndtere og administrere varer på et lager.
2. at få udviklet et distribueret system (f.eks. Client/Server)
3. at få udviklet et system der indbefatter automation (PLC)
4. at anvende netværksprotokoller i kommunikationen mellem de enkelte arbejdsstationer i systemet.
5. at få det beskrevet i en god rapport.

#### Procesmål

Ved projektets afslutning skal det enkelte gruppemedlem alene og i samarbejde med andre:

1. være blevet bedre til at bruge de udviklingsværktøjer vi lærer om (UP, Scrum) til systemudvikling.
2. kunne opbygge og forstå en netværksbaseret softwareløsning.
3. have forståelse for softwarearkitekturen og have fokus på genbrugeligt design.
4. kunne anvende versionsstyringssystem til versionering af dokumenter og kildetekster.
5. have opnået større erfaring med MySQL, Java, netværksprotokoller og automation.
6. kunne lave en holdbar projektplan så milepæle og artefakter bliver klar til tiden.
7. have fået gode erfaringer med udvikling af et **distribueret system.**

## 1.4.4 Inddragelse af fagområder

* SRO inddrages i forbindelse med lagerrobot og implementering i et distribueret system.
* SUD inddrages i forbindelse med udvikling af systemets forskellige dele og fastlæggelse af systemets arkitektur og design.
* KOM inddrages i forbindelse med socket programmering i det distribuerede system.

# Del II: Procesmodel

# 2.1 Projektværktøjer

I de følgende afsnit vil værktøjer benyttet i projektet blive beskrevet.

## 2.1.1 Rapportskrivning

Til rapport skrivning har vi brugt Microsoft word. Vi overvejede at bruge LaTex som nogle havde erfaring med, men denne ide blev droppet fordi vi mente det blev for tidskrævende.

## 2.1.2 Udviklingsmiljø

Programmet er programmeret ved hjælp af Eclipse, fordi det meste af gruppen havde erfaringer med dette program. Det er også meget let at integrere med GitHub, som vi brugte til versionsstyring. Netbeans er også blevet brugt, i forbindelse med programmerings opgaver som ikke krævede at flere gruppemedlemmer arbejdede med på samme tid. Netbeans kunne vi ikke få til at virke ordentligt med GitHub.

## 2.1.3 UML Modellering

Alle UML diagrammer er tegnet på online diagram tegnings hjemmesiden www.Cacoo.com. Denne hjemmeside har været brugt til at tegne diagrammer igennem flere semestre. Den har ikke så mange funktioner som for eksempel Microsoft Visio, men er hurtigere at bruge. Det er også muligt for flere brugere at redigere et diagram på samme tid, samt der er et simpelt versionsstyrings værktøj indbygget.

## 2.1.4 Database

Som database har vi brugt MySQL, fordi alle havde erfaring med dette open source program. Database undervisningen på 1. semester brugte også dette program, hvorfra vi har erfaringen. MySQL Workbench er blevet brugt til at teste databasen, og kontrollere sql syntaks.

## 2.1.5 Versionsstyring

Til versionsstyring er GitHub blevet brugt. GitHub er et frit distribueret versionsstyringssystem. I opstartsfasen brugte vi dropbox til udveksling af dokumenter. GitHub gjorde det muligt at linke til alle vores java filer ved hjælp af Eclipse, dette gjorde at muligt for flere medlemmer af gruppen at redigere i det samlede program på samme tid. Dropbox blev også brugt under rapport skrivningen, da GitHub og Microsoft word filer ikke arbejdede godt sammen.

# 2.2 Projektstyring

Der anvendes UP til den overordnede systemudvikling (iterationer, artefakter). Scrum er afprøvet som redskab under inceptionsfasen.

## 2.2.1 Unified Process

Vi har anvendt Unified Process (UP) under systemudviklingen. UP gør det lettere at få overblik over store systemer, ved at dele det op i mindre dele. UP der dele op til 4 fase:

* Inception – Første fase går ud på at danne en forståelse overfor systemet. Her bliver der lavet funktionelle og ikke-funktionelle krav til systemet, hvis man ikke har disse fra en kunde man arbejder for. Ud fra de funktionelle krav vil man lave use cases og skabe den første domain model. Før fasen er ovre, vil man prioritere sine use cases, og de mest essentielle vil man arbejde videre med i første elaboration.
* Elaboration – Denne fase kan gentages, og vil normalt blive det. I første elaboration vil man arbejde det de højst prioriterede use cases. Ud fra dem vil man gå dybere i detaljer med dem, ved at skrive en expanded use case som beskriver handlinger på systemet og systemets svar. Disse expanded use cases vil så blive viderearbejdet til et system sekvens diagram, hvor man grafisk viser kald og svar på systemet. Det er på disse diagrammer de første metodenavne komme på plads. Næste punkt er at beskrive system operations kontrakter for hver interaktion en aktør har på systemet, i hver use case. Dette er nødvendigt for at konstruere interaktions diagrammer. I interaktions diagrammer er alle metoder som skal til, for at use cases fungerer med. Det er beskrevet hvilke klasser som benytter dem, og hvad de skal returnere. Derefter kan klasse diagrammet laves, og det første kode kan skrives og testes. Herefter kan man så påbegynde en ny elaboration med nye use cases.
* Construction – Efter man har skabt overblik over systemet i elaboration fasen, kan construction fasen begynde. Det er her selve programmeringen foregår, selvom det første programmering starter i elaboration.
* Transition – Når hele systemet er færdigt og leveret til en kunde, begynder transition. Her bliver der arbejdet videre på det feedback kunder kommer tilbage med, sammen med videreudviklinger.

De vigtigste faser i vores projektforløb har været inceptions- og elaborations faserne. Ovenstående er en kort beskrivelse af nogle vigtige elementer og artefakter i de forskellige faser. Det vigtigste mål er arkitekturen i systemet. Det er når interaktionsdiagrammerne skal laves at den indre arkitektur kommer på plads.

## 2.2.2 SCRUM

Scrum er et agilt software udviklings værktøj til styring af software projekter. Det er et meget resultatbaseret værktøj, hvor et arbejde kun har værdi når det er færdigt. Et Projekt bliver delt op i små produkter, som hver har nogle krav som skal bruges til at vurdere produktet som færdig. Alle disse produkter bliver så givet en story point værdi, som er en værdi som symboliserer, hvor svær og tidskrævende den er. Disse produkter bliver så placeret i en produkt *backlog*.

Arbejdet forgår i sprints som typisk er i en periode fra en til fire uger, hvor der arbejdes med en eller flere produkter fra backloggen. Når sprintet er færdig vil hvert produkt blive vurderet, som hvis kravene er imødekommet vil produktet blive godkendt, ellers ryger opgaven tilbage på backloggen.

Et Scrum team består af følgende aktører:

* Projekt Owner – denne person står for at håndtere backloggen og vurdere produkterne. Samt at kommunikere med kunden.
* Scrum Master – denne person er en mellemmand mellem projekt owner og development team. Det er også personen opgave at sørge for scrum processen foregår som den skal.
* Development team – dette er de forskellige medlemmer af gruppen, som arbejder på produkterne.

Projektgruppen har deltaget i et kursus i Scrum, men har valgt ikke at følge Scrum metoden gennem projektet, men tage enkelte dele af metoden ud og anvende dem under inceptionsfasen.

#### Scrum Storypoints

Scrum storypoint bruges til at bestemme hvor tidkrævende Use Cases er. Måden at afgøre det på er at spille et spil Planning poker, også kaldet Scrum poker. Reglerne er simple, hver deltager har 11 kort med værdierne 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100. For hver use case der er, vælger hver deltager en kort værdi, som symboliserer Use Casens kompleksitet. Hvis alle deltagerere ikke har valgt det samme kort, så argumenterer de 2 personer, som har valgt henholdsvis den laveste og højeste værdi, hvad der ligger til grund for deres bud. Derefter vælges der kort igen, som spillet forsætter indtil der er enighed.

# 2.3 Den overordnede projektplan

Den overordnede planlægning for projektets forløb fra start til slut.

|  |  |  |
| --- | --- | --- |
| **Uge** | **Emne** | **Beskrivelse** |
| **40** | Projektgrundlag |  |
| **41** | Inception – 1. Iteration |  |
| **42** | **EFTERÅRSFERIE** |  |
| **43** | Inception – 1. Iteration |  |
| **44** | Elaboration – 1. Iteration |  |
| **45** | Elaboration – 1. Iteration |  |
| **46** | Elaboration – 2. Iteration |  |
| **47** | Elaboration – 2. Iteration |  |
| **48** | Konstruktion |  |
| **49** | Konstruktion |  |
| **50** | Rapport |  |
|  |  |  |
| **AFLEVERING** | | |

**Tabel 2.3.1:** Overordnet projektplan

# 2.4 Det faktiske forløb i projektet

Vi havde en forventning om at benytte dele af Scrum i en af faserne af vores projekt, dette blev i inceptions fasen. Grundet til at vi ikke valgte at benytte Scrum mere var på grund at antallet af medlemmer i gruppen. Hvis den 4 mand store projektgruppe skulle afgive en mand til at spille projekt owner, og en mand til at være scrum master, Så var vi kun 2 tilbage i development team’et. Vi overvejede også at folk skulle varetage flere roller og skiftes om positionerne, men den ide droppede vi hurtigt igen. Det ville blive for kompliceret, og sinke projektforløbet mere end det gavnede. Den del af Scrum, som vi valgte at benytte var story point systemet, som vi brugte til at prioritere vores use cases med.

I starten af projekt forløbet holdte vi et møde hver uge, hvor medlemmer kunne have projekt hjemmearbejde for. Det blev dog taget hensyn til alle medlemmer, hvis vi havde en portefølje der skulle afleveres. Allerede efter 2 ugers projekt arbejde, blev antallet af møder per uge hævet til 2. Vi havde ikke faste møder med vores vejleder, men tog først fat i ham når gruppen ikke kunne nå enighed i en diskussion. Ellers brugte vi vores andre undervisere, hvis diskussionsemnet havde relevans i forhold til deres undervisning.

Vi havde som mål at nå to elaborations faser, men vi nåede kun én. Dette skyldes dels et dårligt estimat af de prioriterede use cases, som blev estimeret mindre end de var og en lang opstarts periode, hvor projektgruppen fik sporet sig ind på en fælles forståelse og definering af opgavens problemstilling.

# Del III: Resultater

# 3.1 Vision

Projektgruppens vision for projektet er at lave et system, som kan hjælpe med håndteringen af varer på et centralt lager system (Central Storage System). Vi vil gøre det muligt for kunder, som i vores tilfælde er detailbutikker, at bestille varer i vores system. Detailbutikker bruger deres eget system til at bestiller varer, dette system vil så indeholde en tynd klient til centralt lager systemet. En detailbutik kan sende en ordre til centralt lager system. Når en ordre blev behandlet i centralt lager system, vil de bestilte varer blive reserveret på lageret, og være klar til at blive plukket på lageret. Til den fysiske plukning vil vi benytte en robot, som både kan sætte varer på lager, og hente dem igen.

En lager manager står for den overordnede styring at robotten, det er denne person som står for at sætte varer på lager, og hente dem igen. Måden dette skal ske på er at, manageren sætter varen på et samle bånd. Derefter får systemet besked på at sætte varen på lager, derefter vil robotten overtage håndteringen af varen. Når en varer så skal hentes igen, vil det ske i forbindelse med pakning af en ordre. Manageren vil bede systemet om at hente den hele ordre, hvorefter alle varer på denne ordre vil blive leveret til et opsamlingssted. Manageren behøver derefter kun at transportere varerne op på en ventende lastbil. Denne del af system kunne fx laves på en mobil enhed, som manageren havde på sin gaffeltruck. Herfra vil han så kunne styrer de funktioner han har brug for, uden at forlade trucken.

Til hele systemet vil der også være knyttet en administrator, som vil kunne oprette nye varer og se hvilke aktive ordre systemet arbejder med.

Hele systemet skal laves på en måde, så det mindsker behovet for medarbejdere til at håndtere varer. Systemet skal være let at bruge, og ikke kræve særlig uddannelse.



**Figur 3.1.1:** Vision for systemet

# 3.2 Krav til systemet

I *Tabel 3.2.1* er de funktionelle krav til systemet listet. Kravene er fastlagt af projektgruppen.

|  |  |
| --- | --- |
|  | **Funktionelle krav** |
| **Krav** | **Beskrivelse** |
| **001** | Systemet skal kunne håndtere Vare som er inddelt efter Varetype. |
| **002** | Der skal kunne oprettes og slettes Vare, Varetyper og brugere i systemet. |
| **003** | Der skal kunne redigeres i eksisterende Vare, Varetype og Brugere i systemet. |
| **004** | En Vare skal have tilknyttet en Lagerplacering. |
| **005** | Der skal være en søgefunktion i systemet, der gør det muligt at finde informationer om en Vare, Varetype, Bruger og Ordre i systemet. |
| **006** | Brugere af systemet skal verificeres af systemet. Evt. via log ind. |
| **007** | Systemet skal kunne modtage en Ordre fra en Detailbutik. |
| **008** | Systemet skal kunne give besked til Detailbutik når en Ordre er modtaget. |
| **009** | En ny Vare skal indscannes og registreres i systemet inden den bringes til lageret. |
| **010** | Systemet skal give den korrekte Lagerplacering til Lagerrobot. |
| **011** | Systemet skal kunne bede om at få hentet en Vare med en given Lagerplacering fra lager vha. Lagerrobot. |
| **012** | Systemet skal kunne udskrive en Følgeseddel med Detailbutikkens informationer når en vare hentes fra Lageret. |
| **013** | En Detailbutik skal kunne se Vare i systemet. |
| **014** | En modtaget Ordre skal indeholde informationer om Detailbutik. |
| **015** | Systemet skal kunne sende en ordrebekræftelse til Detailbutik. |

**Tabel 3.2.1.1:** De funktionelle krav til systemet

*Tabel 3.2.1.2* indeholder alle de ikke-funktionelle krav til systemet. Lige som de funktionelle er disse bestemt af projektgruppen.

|  |  |  |  |
| --- | --- | --- | --- |
| **Ikke-Funktionelle krav** | | | |
| **Krav** | **Titel** | **Beskrivelse** | **Prioritet** |
| **201** | **Robusthed** | Systemet må ikke gå ned pga. brugerfejl |  |
| **202** | **Brugervenligt** | Systemet skal kunne bruges uden efteruddannelse |  |
| **203** | **Vareinformation** | Systemet skal kunne levere en dækkende beskrivelse af en vare. |  |
| **204** | **Dokumentation** | Systemet skal sørge for at lageret er ajourført. |  |
| **205** | **Backup** | Systemet skal lave backup en gang pr. døgn. |  |
| **206** | **Hardware** | Systemet skal være platformsuafhængigt (pc windows/linux etc.). Hardware skal kunne udskiftes uden at påvirke systemet. |  |
| **207** | **Sikkerhed** | Brugeradgang til Systemet skal begrænses |  |

**Tabel 3.2.1.2:** De ikke-funktionelle krav til systemet

# 3.3 Aktørliste

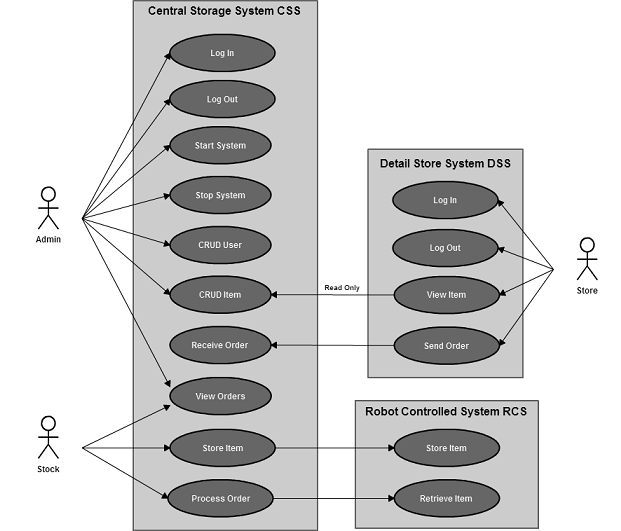
*Tabel 3.3.1* indeholder de aktører, der findes på CSS. En aktør kan enten være en person eller et andet system.

|  |  |  |
| --- | --- | --- |
| **Aktør** | **Beskrivelse** | **Type** |
| **Administrator** | Administrator som har til ansvar at starte systemet op samt søge oplysninger ordre i systemet. | Person |
| **Manager** | Manager fungerer på lageret og har til ansvar at ekspedere ordre i systemet. Han har også til opgave at lægge nye vare ind på lageret. | Person |
| **DSS** | Detail Store System som sender ordre til CSS. | system |
| **RCS** | RCS som kommunikerer med CSS når vare skal bringes til og fra lageret. | system |

**Tabel 3.3.1**: Aktørliste

# 3.4 Use Cases

Ud fra de fundne krav til systemet er følgende Use Cases defineret af projektgruppen. Use Cases vises i en Use Case model i *Figur 3.4.1*. Hvert Use Case i figuren er overskrift for et hændelsesforløb mellem en aktør og systemet. Hvert Use Case vil blive beskrevet mere detaljeret i de følgende afsnit.



**Figur 3.4.1**: Use Case model

## 3.4.1 Prioritering af Use cases

For at dele systemudviklingen op i flere iterationer prioriterede vi Use Cases i systemet for at finde ud af hvilke der skulle med i først iteration. *Tabel 3.4.1.2* viser en liste af de use cases, der findes i systemet og den prioritet vi har vælge at give dem.

#### MoSCoW prioritering

Prioriteringen foregik ud fra *MoSCoW* metoden, der går ud på at prioritere efter begreberne: **M**ust, **S**hould, **C**ould og **W**ont.

|  |  |
| --- | --- |
| **Must** | Skal med i den endelige version |
| **Should** | Burde være med i den endelige version og har derfor høj prioritet |
| **Could** | En feature som kunne være god at have med hvis tiden tillader det |
| **Wont** | En funktionalitet som ikke kommer med nu, men som evt. kommer med i næste version |

**Tabel 3.4.1.1:** Begreber i MoSCow metoden

#### SCRUM Story points

Efter *MoSCoW* prioriteringen opstillede projektgruppen alle *Must* Use Casesog brugte derefter Storypoints fra Scrum metoden til at pointgive Use Casene. I *Tabel 3.4.1.2* ses resultatet af både *MoSCow-* og *Story point* prioriteringen.

#### Resultat af prioriteringen

Tabel 3.4.1.2viserresultatet af prioriteringen af Use Cases. Der arbejdes med alle *Must* Use Cases i første iteration. Hvert Use Case er listet med reference til de tilhørende krav fra *Tabel 3.2.1.1: Funktionelle krav.*

|  |  |  |
| --- | --- | --- |
| **Use Cases med prioritering** | | |
| **Use case** | **Krav ref.** | **MoSCow prioritet** |
| **Log in** | 006 | C |
| **Log out** | 006 | C |
| **Start system** | - | S |
| **Stop system** | - | S |
| **CRUD User** | 005, 002, 003 | C |
| **Receive order** | 007, 008, 014, 015 | M (P30) |
| **View orders** | 005 | M (P8) |
| **Store item** | 010, 009 | M (P10) |
| **Process order** | 010, 011, 012 | M (P10) |

**Tabel 3.4.1.2:** Tabel over use cases i systemet og deres tilhørende prioritering

Det er derfor Use casene *Receive order, View orders, Store Item* og *Process order* der bliver behandlet i første iteration.

## 3.4.2 High-level Use Cases

High-level Use Cases beskriver hvad der sker i de enkelte situationer i systemet. En Use Case er beskrevet med aktør, formål og en beskrivelse af situationen.

Nedenfor kan ses *High-level Use Cases* for 1. Iteration. Vi vælger at indsætte High-level Use Cases for *Receive order* og *Process order* herunder og vedlægge de to andre som bilag under *Appendix B.*

|  |  |
| --- | --- |
| Receive Order | |
| Use case: | Receive Order |
| Actors: | DSS (Detail Storage System) |
| Purpose: | Receive an order and handle it accordingly. |
| Description: | System receives an order from another system. The attached encrypted user identification is validated. Each item on the order-list is cross-referenced with the amount of the specific item in stock. If an item is not in stock the lack is noted and attached to the order-confirmation, which is returned then to the user. The items on the order-list are marked as reserved until they are checked out of the System. |

**Tabel 3.4.2.1: Receive order**

|  |  |
| --- | --- |
| Process Order | |
| Use case: | Process Order |
| Actors: | Stock |
| Purpose: | Retrieve all items in a specific order from storage. |
| Description: | All items are continuously retrieved from storage. Every time an Item is retrieved it is checked out of the System. When all Items have been retrieved, the order status is changed. |

**Tabel 3.4.2.2: Process order**

## 3.4.3 Expanded Use Cases

I *expanded Use Cases* beskrives hvert Use Case mere detaljeret med *pre- og postconditions*, et typiske hændelsesforløb og et eventuelt alternativt hændelsesforløb.

Nedenfor er Expanded Use Cases for *Receive Order* og *Process order*. De resterende *Expanded Use Cases* er vedlagt som bilag under Appendix C.

|  |  |  |  |
| --- | --- | --- | --- |
| Receive Order | | | |
| Use case: | | **Receive Order** | |
| Actors: | | DSS(Detail Store System) | |
| Purpose: | | Receive Order from DSS and save it in the system. | |
| Overview: | | CSS receives an order from a DSS over a secure socket connection. CSS confirms the identity of the detail store and converts the received order to an active order in CSS and stores it in the system. The system goes through the order and updates the stock by reserving each item in the order and associates it to the order. As the system goes through the order; reserving items, it puts together a Confirmation to send back to the detail store. The Confirmation holds information about which items can be delivered and which is in backorder. | |
| Type: | | Essential | |
| Preconditions: | | The DSS is known by the System. | |
| Postconditions: | | An Order is created in the system and the stock is updated. | |
| Special Req.: | | No special requirements are needed. | |
|  | |  | |
| Flow of Events | | | |
| Actor Action | | | **System Response** |
| 1. This use case starts when a Detail store system sends a request to CSS containing an order and store information. | | | 1. CSS responds to the request and a connection is established. |
|  | | | 1. CSS verifies the Detail store identity. |
|  | | | 1. CSS verifies that the Order is not a duplicate. |
|  | | | 1. CSS convert the incoming order to an order in CSS and store it in the system. |
|  | | | 1. CSS goes through the stock reserving and associating items to the Order. |
|  | | | 1. CSS creates a Confirmation and sends it to the DSS |
| 1. The DSS sends a “Close connection” message to CSS. | | |  |
|  | | | 1. CSS receives the “close connection” message, and closes the connection. |
|  | | |  |
| Alternative Flow of Events | | | |
| Line 2: | CSS rejects the request (return to Line 1). | | |
| Line 3: | CSS cannot confirm the Detail store identity and rejects the order (**return to line 1**). | | |
| Line 4: | CSS find that the order is a duplicate (**go to line 9**) | | |
| Line 7: | The Detail store do not receive a Confirmation from CSS (**return to line 7 until timeout - on timeout go to line 9**) | | |

**Tabel 3.4.3.1: Receive order**

|  |  |  |  |
| --- | --- | --- | --- |
| Process Order | | | |
| Use case: | | **Process Order** | |
| Actors: | | Manager | |
| Purpose: | | Manager processes an Order, retrieve the ordered Items from RCS (Stock) and ready the Items for shipment. | |
| Overview: | | All items are continuously retrieved from storage. Every time an Item is retrieved it is checked out of the System. When all Items have been retrieved, the order status is changed. | |
| Type: | | Essential | |
| Preconditions: | | Stock positions of Items are known to the System. | |
| Postconditions: | | Order has its State changed to Processed. Items are removed from Stock | |
| Special Req.: | | No special requirements are needed. | |
|  | |  | |
| Flow of Events | | | |
| Actor Action | | | **System Response** |
| 1. This Use Case begins when the Manager wants to process an Order | | |  |
| 1. The Manager requests the Order to be processed. | | | 1. For each Item CSS looks up the Item’s Stockposition and sends a ’Retrieve Item’ with that Stockposition to RCS. RCS retrieves the Item and responds with a ’retrieved’. CSS removes the Item from Stock. On last Item the State of the Order is changed to Processed. |
|  | | |  |
| Alternative Flow of Events | | | |
| Line 3: | If RCS responds with ’failure’, CSS indicates a ’failure’ for that Stockposition. CSS then looks up another ’not reserved’ Item of same Itemtype and retrieves it instead. If no available Item CSS indicates a ’failure’ for retrieving that Item. | | |

**Tabel 3.4.3.1: Process order**

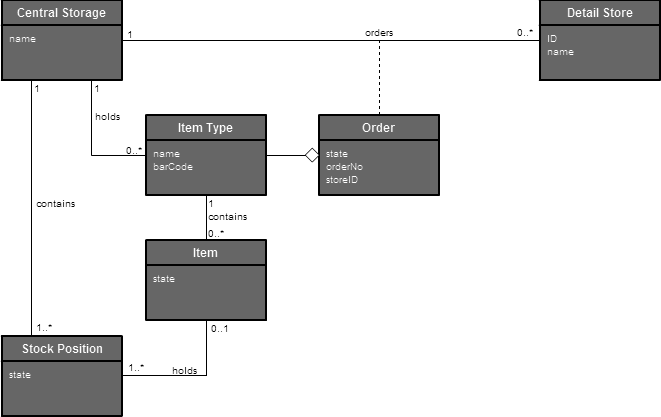
# 3.5 Klassediagram

*Tabel 3.5.1* indeholder en liste af klasserne i modellen listet med tilhørende beskrivelse.

|  |  |
| --- | --- |
| **Klasser** | **Beskrivelse** |
| Central Storage | Det centrale lager |
| Detail Store | Detail store systemet der sender ordre til Central Storage. |
| Order | Den ordre der oprettes i systemet når en Detail Store sender en ordre til Central Storage. |
| Item | Den vare der enten ligger på lager og enten er ledig eller reserveret til en given ordre. |
| Item Type | En beskrivelse af de Items der knytter sig til den givne Item type. |
| Stock Position | En given position på lageret. En position kan enten være optaget eller fri |

**Tabel 3.5.1:** Klasserne i modellen

*Figur 3.5.1* Viser klassediagram med klasserne og deres relationer til hinanden.



**Figur 3.5.1**: Klassediagram

# 3.6 Systemsekvensdiagrammer og Systemoperationskontrakter

Før vi påbegynder designarbejdet beskriver vi ”normal hændelser” og aktør-påvirkninger i systemet vha. System sekvensdiagrammerne. Hvert kald fra aktør til system beskrives i en systemoperationskontrakt.

Nedenfor er systemskvensdiagrammer og de tilhørende systemoperationskontrakter for Use Casene Recieve order og Proces order vist og de resterende er ved lagt som bilag under *Appendix D*.

#### Recieve Order

CSS modtager en bestilling fra DSS og CSS kvitterer for bestillingen.

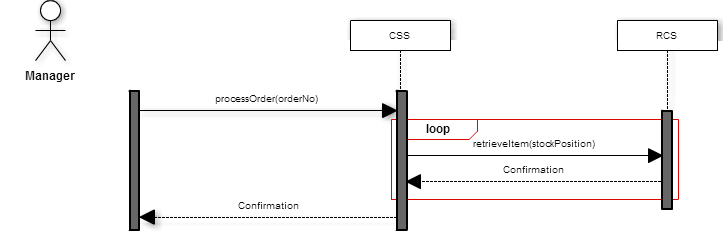


**Figur 3.6.1:** Recieve Order

|  |  |
| --- | --- |
| UC - Receive Order | |
| **Contract Name:** | receiveOrder(orderRequest) |
| **Responsibilities:** | To receive an OrderRequest from a Detail store, make it in to an Order and reserve the requested items in the system. |
| **Type:** | System |
| **Exceptions:** | Not a valid OrderRequest format. The system cannot validate the StoreInfo. The Order is a duplicate of a privies received OrderRequest. |
| **Output:** | A confirmation to send to the Detail Store. |
| **Preconditions:** | No preconditions for this contract. |
| **Postconditions:** | An OrderRequest is created by the CSS. An Order is created and associated with the OrderRequest. Items is reserved and associated with the Order. |

**Tabel 3.6.1:** Recieve Order

Manageren har udført handlingen *View order* og vælgt en *åben* order i systemet derefter vælger ham *Process Order* på den ønskede order. CSS giver RCS besked om at hente hver Item på ordren ud fra lageret.



**Figur 3.6.2:** Process order

|  |  |
| --- | --- |
| UC: Process Order | |
| Contract Name: | processOrder(orderNo) |
| Responsibilities: | Retrieve all Items in an Order from the stock and check it out of the System. |
| Type: | System |
| Exceptions: | No exceptions for this contract. |
| Output: | A confirmation of either success or failure is returned. |
| Preconditions: | An Order has to exist in the System. |
| Postconditions: | When an Item is retrieved from the Stock, the state of the StockPosition is changed to ‘free’ and the item removed from the System. When all Items have been processed, change the state of the Order to ‘processed’. |

**Tabel 3.6.2:** Process Order

# 3.7 Softwarearkitektur

I dette kapitel vil designmål for systemet blive opstillet og systemets arkitektur blive beskrevet.

## 3.7.1 Designmål

Ud fra de *ikke-funktionelle krav* bestemmes designmål for systemet.

**Driftssikkerhed:**

* Systemet skal være robust så det ikke går ned ved f.eks. ved fejlagtigt brugerinput.
* Systemet skal være pålideligt. Overensstemmelse mellem specificeret og observeret adfærd.
* Systemet skal være tilgængeligt. Den del af tiden hvor systemet ikke kan bruges til udførelse af normale opgaver skal være minimal.

**Anvendelighed:**

* Systemet skal være brugervenligt og intuitivt.
* Systemet skal støtte brugeren i dennes arbejde.

## 3.7.2 Arkitektur - PCMEF+ framework

Projektgruppen har valgt at bruge PCMEF+ arkitekturen. PCMEF+ er det software framework der er blevet undervist i, i system udvikling og design i 3. semester og med dette valg kunne gruppen forbedre forståelsen af dette framework. I de følgende afsnit vil principperne bag PCMEF+ og brugen af framework’et i vores projekt blive beskrevet.

#### Principper i PCMEF+

PCMEF+ tilbyder en lagdelt arkitektur, som foreskriver nogle principper, der gør den velegnede til distribuerede systemer. En væsentlig grund til at bruge dette framework er dets anvendelighed ved software genbrug, det vil sige at frameworket er velegnet til videreudvikling og udbygning af et givent software system.

PCMEF arkitekturen er lagdelt og benytter sig af fire lag indeholdende fem foruddefinerede pakker. Her beskrevt oppefra og ned:

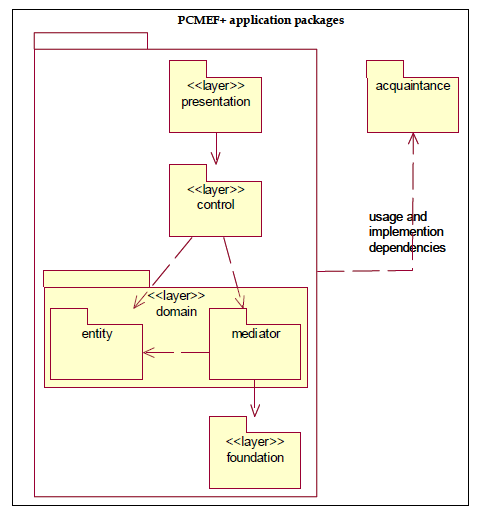
* Presentation Layer: presentation (GUI og interface til System aktører)
* Control layer: control (styring af domain og interface til presentation)
* Domain layer: Entity ( den del af systemet hvor instanser af businesmodel klasser opbevares og interagerer) samt Mediator (formidler kontakt mellem Entity og nedre lag)
* Foundation layer: foundation (Persistens lag. Styrer database og adgang til subsystemer)

Den eneste forskel på PCMEF og PCMEF+ er Acquaintance pakken, en pakke der er global adgang til og som bruges i de tilfælde hvor det ikke kan lade sig gøre at overholde alle principperne. Det kan være hvis der er brug for opadgående kommunikation eller kommunikation mellem ikke-nabo klasser.

**Principperne:**

* **Downward Dependency Principle:** Afhængigheder findes oppefra og ned i arkitekturen, så øvre lag kender til laget lige under sig.
* **Upward Notification Principle:** Vha. Designmønstret Observer. Objekter på øvre lag (*Subscriber) kan abonnere på* objekter fra nedre lag (*publisher*). Når publisher skifter tilstand sendes en notifikation til dets *abonnenter*.
* **Neighbor Communication Principle:** Pakker kan kun kommunikere direkte med sine nabopakker. Ikke-nabo kommunikation kan gøres gennem en bekendtskabsklasse (Acquaintance).
* **Explicit Association Principle:** Objekter har kun kendskab til andre objekter igennem en reference oprettet lokalt.
* **Cycle Elimination Principle:** princip for at undgå cirkulære afhængigheder. Hvis der er en gensidig afhængighed mellem to instanser A og B oprettes en ny klasse C med den del af A som B er afhængig af. Dermed er A afhængig af B, og A og B afhængig eaf C. Dermed er den cirkulære afhængighed fjernet.
* **Class Naming Principle:** Angiver første bogstav i pakkenavnet i klasserne, så man kan se hvilken pakke klassen tilhører.
* **Acquaintance Package Principle:** Bruges til opadgående eller ikke-nabo kommunikation for at kunne overholde principper i PCMEF+. Dermed kan pakker fra nedre lag offentligøre information som pakker der ikke har direkte adgang til dem kan bruge for at opnå adgang til dele af deres funktionalitet.

Den efterfølgende figur viser opbygningen af PCMEF+ og de afhængigheder den indeholder. Det vigtige ved dette framework er at de enkelte lag kun er direkte afhængige af ændringer i laget direkte under dem. Det betyder naturligvis at større ændringer i de nederste lag i værste fald kan give anledning til rettelser hele vejen op.

**

**Figur 3.7.2.1**: Diagram over PCMEF+

#### PCMEF+ i CSS

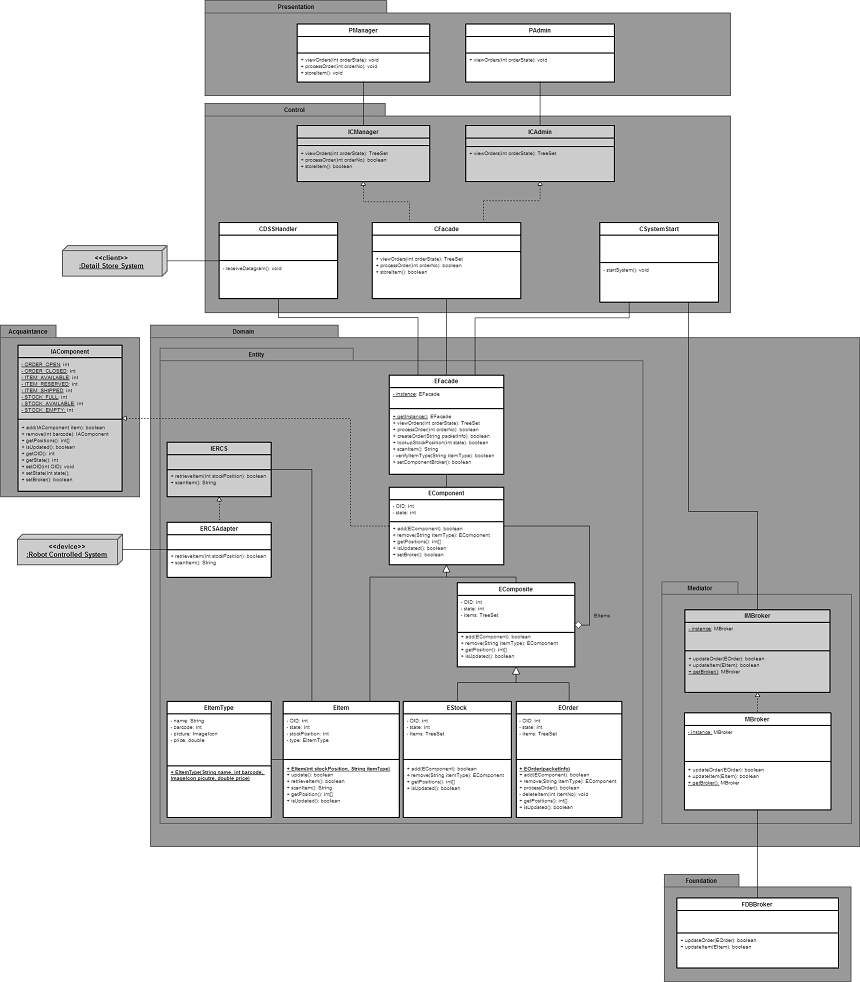
Selve CSS med de 4 Use Cases der blev udvalgt til 1. Iteration er ikke så omfattende et system at det berettiger til at bruge PCMEF+ men det er jo heller ikke pointen. Pointen er at modellere CSS på dette framework som om det var et stor og kompliceret system hvor hver enkelt del bliver udviklet og vedligeholdt af hver sit team. De klasser CSS bliver opdelt I afspejler dels dette, dels at vise vi kan bruge en række bestemte designmønstre I implementeringen af systemet. Forløbet af designudviklingen er beskrevet andetsteds så her ses kun på det endelige resultat.

Som en væsentlig tilføjelse skal nævnes at det er ‘tilladt’ at lave et interface direkte ud gennem entity hvis aktøren er sekundær (passiv).

Der tages udgangspunkt I ‘entity’. Det er her selve forretnings objekter er. I CSS er det objekterne: EStock, EOrder, EItem, EItemType og ERCSAdapter. De fire første bruges til ordrehåndtering og den sidste er et interface til RCS (Robot Control System) der her er et Allen-Bradley PLC system. RCS er passiv dvs. Den igangsætter ikke nogen use cases den reagerer kun på CSS beskeder. Derfor er det tilladt at have den som interface direkte ud af entity.

Der er I CSS kun brugt en SQL database som persistenslag, derfor udgør mediator og foundation pakkerne et selvstændigt persistens framework på hvilket resten af CSS hviler. Persistens delen beskrives senere.

Presentation og control lagene sikrer brugeradgang for indtil videre tre typer af brugere. Admin der blandt andet forestår opstart af CSS. Manager der logger på systemet og og udfører ordrerhåndtering samt DSS (Detail Store System) der via TCP protokollen sender nye ordrer til CSS. Klasse diagrammet for CSS ser ved deadline således ud:



**Figur 3.7.2.2**: CSS klassediagram

## 3.8.3 Anvendte designmønstre

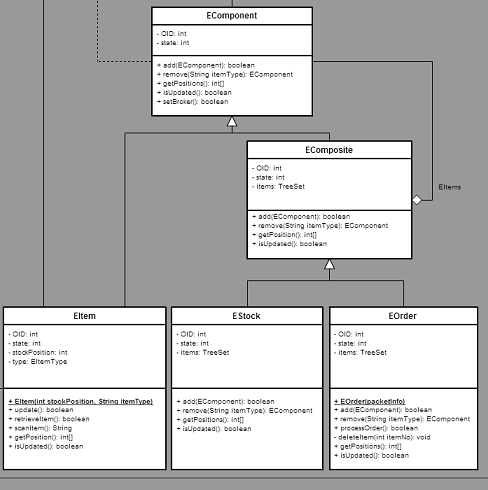
En væsentlig del af projektet er forståelse og anvendelse af designmønstre. Designmønstre er skabeloner til løsning af programopgaver. Igen for at sikre genanvendelig software. I det følgende afsnit beskrives de designmønstre der er anvendt i CSS.

#### Composite pattern

Bruges når vi har behov for at sammensætte objekter i træstrukturer for at beskrive hierarkier bestående af helheder og dele, og når vi ønsker at behandle individuelle objekter og sammensatte grupper af objekter på en ensartet måde. Et hierarki opbygges af en grundlæggende abstrakt component klasser hvorfra andre klasser arver. På den ene side er der diverse containerklasser (Composites) der kan indeholde andre containere eller leafs) på den anden side nodeklasser (Leafs). En væsentlig fordel er rekursive metoder til indsamling af information i en samling af objekter. Derudover er der ’standard’ metoderne add og remove til at flytte objekter rundt.

I CSS benyttes Kompositionsmønstret i forbindelse med klasserne E*Stock*, E*Order* og E*Item. EStock* og E*Order* er containerklasser der indeholder et givent antal EItems (leaf). ’E’et i navnene skyldes PCMEFs navneregel.

#### CSS anvendelse af Composite Pattern



**Figur 3.7.3.1**: CSS anvendelse af Composite Pattern

Som eksempel på anvendelse af Composite i CSS ses på metoden getPositions() der returnerer et array af typen int[]. Her er implementation af metoderne vist for de enkelte java klasser. Bemærk at EOrder og EStock arver deres metode fra EComposite:

Abstract EComponent {

getPositions(){return null;}

}

EItem{

Private int stockPosition=some\_position;

getPositions(){int[] p=this.stockPosition;return p;}

}

Abstract EComposite{

private ArrayList<IAComponent> items = new ArrayList<IAComponent>();

getPositions(){

IntBuffer iBuf = IntBuffer.allocate(someMaxValue);

for(IAComponent comp:items) {iBuf.put(comp.getPositions());}

int[] pos = new int[iBuf.position());

iBuf.clear();

iBuf.get(pos);

return pos;

}

}

EOrder{

getPositions

}

**Kode 3.7.3.1**: EComposite

#### Adapter pattern

Da vi skulle designe en måde at kommunikere med vores robot, overvejede vi 3 muligheder. Den første var at opfatte robotten som en del af vores system, og på den måde kommunikere med den gennem vores foundation lag i PFMEF+ modellen. Efter vi have fået bedre overblik over hele systemet, kunne vi se at robotten ikke direkte var en del af vores system, men mere en sekundær aktør. Dette åbnede 2 nye muligheder for kommunikation, en systemgrænseflade i presentation laget eller en adapter i entity laget.

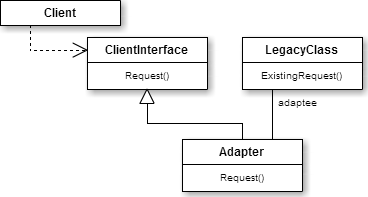


**Figur 3.7.3.2**: Adapter

1. metode - Foundation: Metoden igennem foundation ville kun give mening hvis RCS var en del af systemet.

2. metode - Systemgrænseflade: Denne metode ville være bedst hvis vi skulle modtage information fra RCS, men i vores tilfælde skal RCS kun returnere information når vi beder den om det. Dette vil kun ske når vi skal gemme en vare (storeItem() ) der forventer vi at RCS scanner item’et og returnere den scannede information.

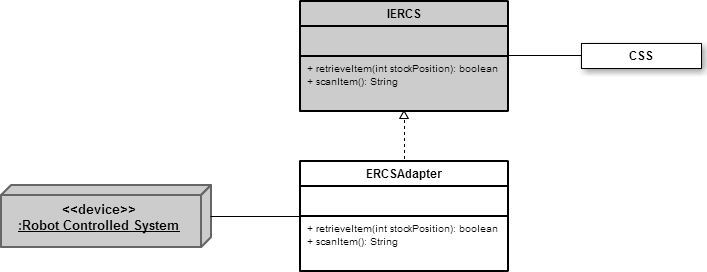
3. metode: Adapter: Ved at bruge en adapter kan vi skabe en meget direkte forbindelse mellem RCS og CSS, i denne pakke, som indeholder item klasse, som skal arbejde tæt sammen med RCS.



**Figur 3.7.3.3**: Adapter pattern

Udgangspunktet for adapter mønstret er dette, her har vi en LegacyClass, et færdigt system der i vores tilfælde er RCS. Vi ønsker at tilgå RCS fra en client nemlig CSS. Dette sker igennem et client interface og en adapter klasse der overholder dette interface.

CRS kan modtage forskellige input fra vores Adapter klasse. I IERCS(client interface) skrives de metoder vi skal kunne benytte, de vil blive nedarvet i ERCSAdapter (Adapter). Adapter klassen skal derfor indeholde alle de metoder, vi vil benytte i LegacyClass. Adapter skal derfor omskrive den metode den modtager til en metode LegacyClass forstår.

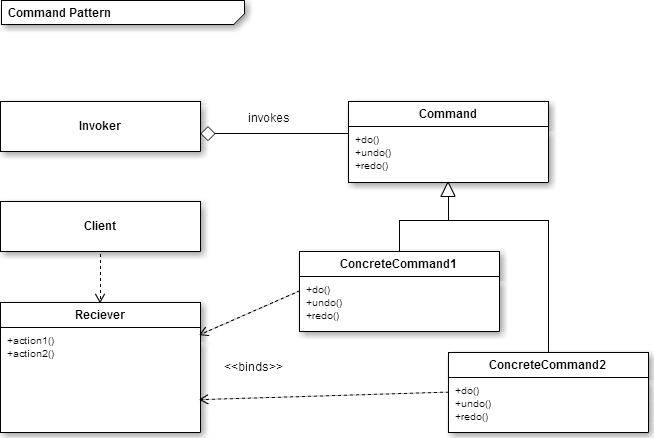


**Figur 3.7.3.4**: Adapter i CSS

I vores system har vi kun 2 metoder som RCS skal kunne udføre, retrieveItem og scanItem. Den sidste bruges også til at gemme varen på lager (storeItem). Disse 2 metoder skal så omskrive til metoder som RCS vil kunne forstå.

#### Command Pattern

Command pattern har til opgave at indkapsle bruger kommandoer (do) så systemet kan udføre evt. rollback (undo) eller genvisning af en kommando sekvens (redo), gennem indkapsling af kommandoerne i konkrete instanser af klasser. Det har yderligere den fordel at ekstra kommandoer kan tilføjes uden at ændre de allerede implementerede kommandoer. Det sker ved at brugeren (invoker) udfører en konkret kommando der bygger på en grundkommando som systemet så videresender til den korrekte modtagerklasse.



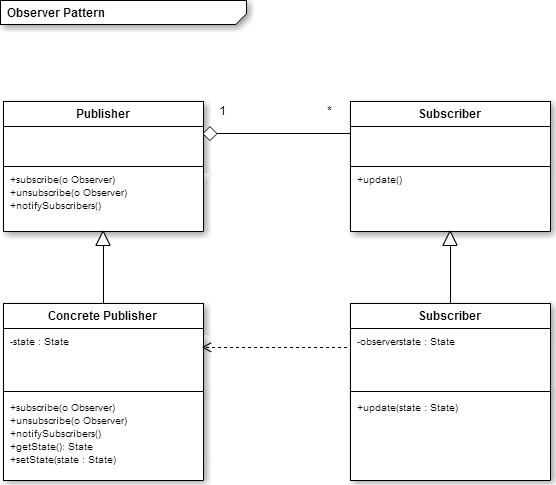
**Figur 3.7.3.5:** Command Pattern

Der er to områder hvor anvendelse af dette pattern er oplagt i CSS. For det første til at implementere use cases. F.eks. gennem CreateOrder kommando der igen bruger en SelectItem kommando. Det andet oplagte sted er i persistensframework’et hvor et hierarki af transaktions kommandoer vil gøre systemet mere fleksibelt.

Desværre har der ikke været tid til at implementere dette design pattern i praksis.

#### Observer Pattern

Som den anvendte PCMEF+ arkitektur foreskriver, foregår al kommunikation nedad i lagene, mens opadgående kommunikation ikke er tilladt da det giver opadgående afhængighed. For at løse dette kan man enten gøre brug af Acquaintance pakken eller bruge notifikation til opadgående beskeder. Notifikation kan løses ved at anvende Observer pattern. Det vil typisk være i forbindelse med opdatering af view’et i præsentationslaget, der er brug for anvendelse af Observer pattern og dette er vist ved implementeringen af CSS.   
Klassen hvori der kan forekomme ændringer, kaldes for publisher og skal implementere et publisher-interface. Klassen der skal have ændringerne at vide, kaldes for subscriber, og skal implementere et subscriber -interface. Publisher klassen tilbyder en subscribe/unsubscribe metode samt en notifySubscriber metode. Subscriber tilbyder en update metode der bliver kaldt af notify metoden.



**Figur 3.7.3.6:** Observer Pattern

I Java sker implementeringen gennem en Observable klasse (publisher) og et Observer interface (subscriber). Observable tilbyder den nødvendige funktionalitet som publisher. Da det er en klasse man nedarver fra kan den med fordel skilles ud således at ejerklassen der holder ’state’ attributen formidler ændringer i denne gennem en publisher. I Observer interfacet i java kaldes update med Observable klassen selv. Det har den fordel at funktionalitet til at håndtere state attributen kan ligge på publisher og derigennem udnyttes af subscriber.

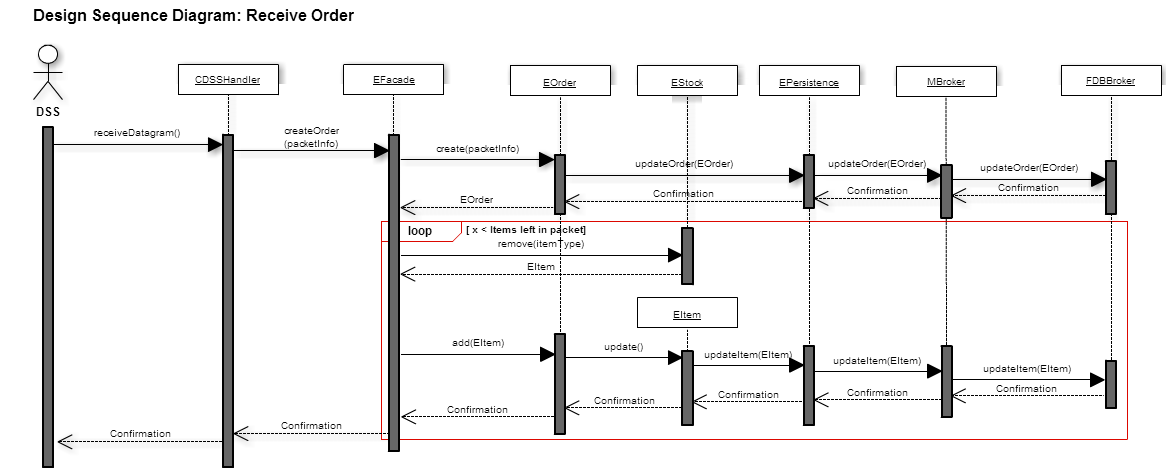
Der er i CSS benyttet Observer pattern til håndtering af ’tilstanden’ af en ordreliste mellem presentation og control lagene.

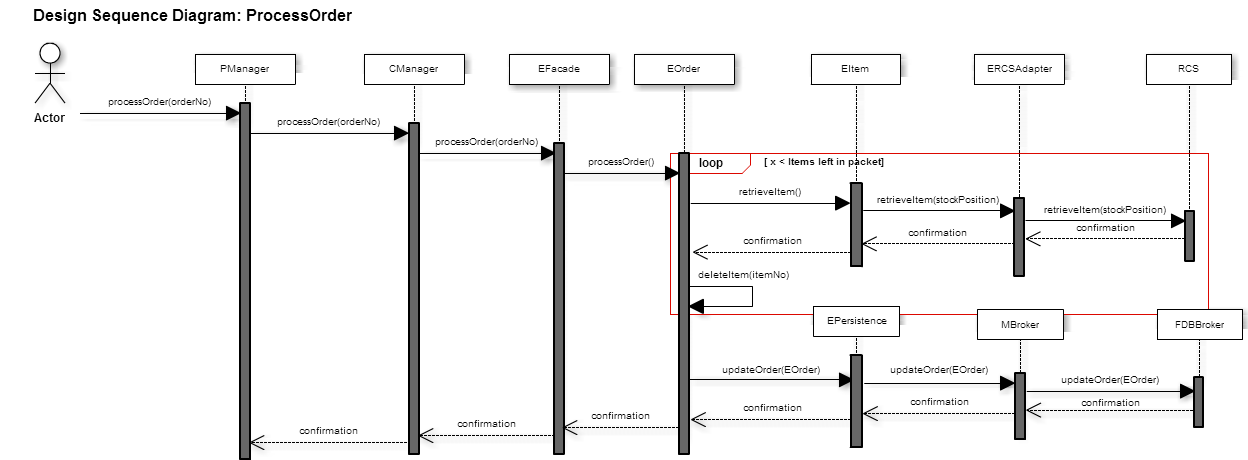
# 3.8 Detaljeret Design

## 3.8.1 Design sekvensdiagrammer

Design sekvensdiagrammerne viser alle de kald, der foretages internt i systemet efter en aktør har foretaget en handling. I dette afsnit viser vi sekvensdiagram for Use casene *Process Order og Receive Order* de resterende vedlægges som bilag under*Appendix E.*

*Figur 3.8.1.1* Viser Sekvensdiagram for *Receive Order* og *Figur 3.8.1.2* Viser Sekvensdiagram for *Process Order*.

**Figur 3.8.1.1** Receive Order. 

**Figur 3.8.1.2:** Process Order. 

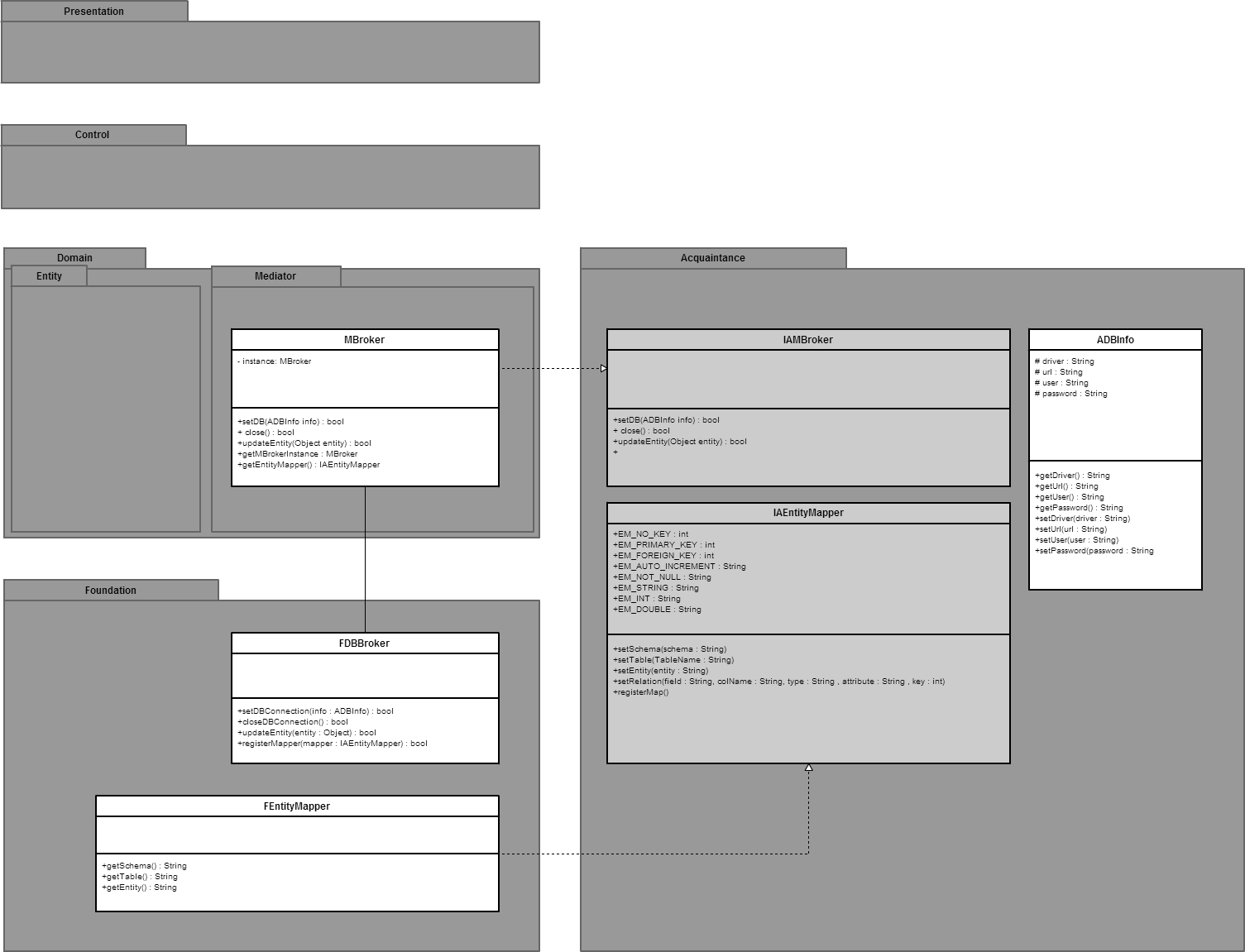
## 3.8.2 Persistens framework

Det primære formål med et persistens framework er at skabe et system der kan sikre at objekter kan hentes (materialize) og gemmes (update) persistent. Systemet skal være sådan opbygget at det kan levere denne funktionalitet til enhver klient der måtte ønske det. Systemet betegnes som et ORM System – ORM står for Object Relation Management – og handler om oversættelsen mellem objektorienteret programmering og relationsorienterede databaser.

Der er en række vigtige funktioner persistens framework’et skal opfylde. De to vigtigste er naturligvis at kunne hente og gemme data fra en persistent form typisk, og også i CSS, en database. Derudover skal det forbedre performance af systemet ved såkaldt ’lazy initialization’ det vil sige ved at have en cache i hukommelsen af de objekter der findes i databasen. Sidst men ikke mindst skal persistens framework overholde PCMEF+ generelle regler specielt om uafhængighed af højereliggende lag. Da database delen befinder sig i foundation betyder dette at frameworket skal organiseres således at denne opadgående uafhængighed overholdes!.

’Lazy initialization’ er ikke implementeret i CSS nuværende version og er derfor et ’senere udviklings emne’. Kravene om overholdelse af PCMEF+ og hente/gemme funktionaliteten er dog implementeret.

Det persistens framework der er udviklet til brug i CSS ser således ud:



**Figur 3.8.2.1:** Persistens Framework

Udgangspunktet er klassen FDBBroker. FDBBroker leverer to ydelser.

* Den leverer adgang til og brug af databasen. Brugen af acquaintance klassen ADBInfo er blot en bekvemmelighed, den klasse kunne skrives ud af systemet ved at ændre getDBConnection metoden i FDBBroker.
* Den leverer ORM Mapping facilitet gennem IAEntityMapper interfacet. Dette implementeres med FEntityMapper klassen i foundation.

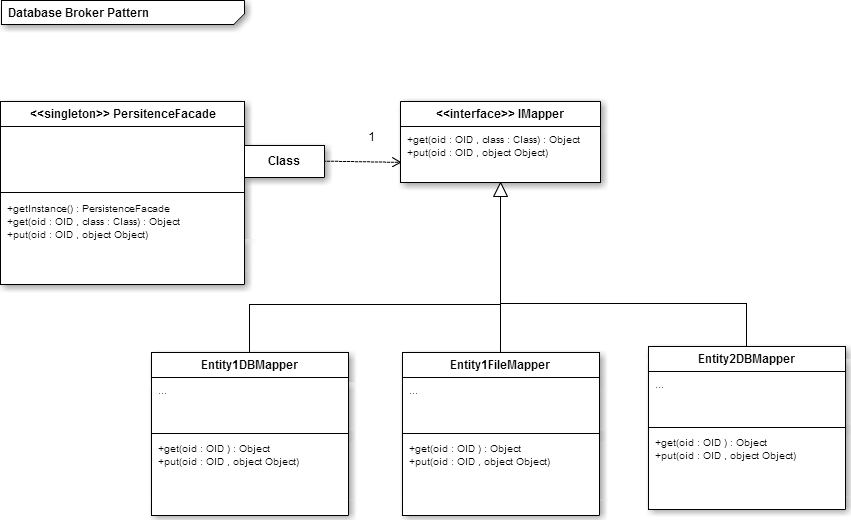
MBroker er formidler af FDBBrokers funktionalitet. Den initialiserer FDBBroker (singleton, delegation) og via IAMBroker interfacet og metoden getMBroker() (singleton) kan control laget levere adgang til MBroker klassen. På denne måde leveres persistensfunktionalitet til hele CSS uden at persistens frameworket er afhængig af nogle klasser i højereliggende lag.

#### ORM Mapping

Den ene funktionalitet der skal leveres er ORM Mapping. Denne mapping sørger for at de felter i en klasse der ønskes gemt i databasen skal mappes til en konkret tabel og en konkret kolonne i denne tabel. Der er flere måder at gøre dette på

#### Database Broker Pattern

Database Broker Pattern beskriver hvordan der for hver Persistens klasse (klasse der skal gemme data) laves en tilsvarende mapperklasse der implementerer et Mapper interface. Disse mapper klasser registreres i persistensfacaden. Når put eller get metoden kaldes bruger facaden den pågældende entitys klassetype til at finde den rigtige map. Det er så mapperen der sørger for at oversætte klassens attributter til sql kommandoer (eller filformat hvis det er en datafil man vil gemme i).



**Figur 3.8.2.2:** DataBase Broker Pattern

Som sagt kræver metoden at man i entity implementerer en mapperklasse for hver entity klasse. Der findes en mere elegant måde at gøre det på nemlig MetaData baserede mappere

#### MetaData Mapping

I stedet for at definere en mapper for hver persistens klasse leverer persistens facaden en mapper der overholder et mapper interface. Denne mapper leverer funktionalitet der binder de enkelte attributter i klassen til f.eks. tabeller i relations databaser. På den måde kan relationerne sættes ’runtime’ dvs. at hver gang der laves en ny persistensklasse skal den blot initialisere og registrere en standard mapper. Konkrete eksempler på dette kommer under implementering.

#### Data Modelling Profile

Som en del af persistens beskrivelsen af et system udarbejdes et UML artifakt kaldet Data Modelling Profile. Det angiver de forskellige tabeller i systemet samt deres indbyrdes relationer.

#### 

**Figur 3.8.2.3:** Data Modelling Profile

## 3.8.3 Singleton pattern

Vi har gjort brug af designmønstret Singleton. Ideen ved et singleton er at skabe kun én instans af klassen, således det altid er det samme man får fat i, ligegyldigt hvor i systemet den kaldes fra. Singleton bruges i de tilfælde, hvor instansen skal være global. Singleton klasser bruges f.eks. i klasser der skaber adgang og kommunikere med databasen eller klasser der er facader mellem to lag i modellen.

|  |  |
| --- | --- |
| **Singleton klasser i CSS** | |
| **Singleton klasse navn** | **Beskrivelse** |
| **EFacade** | Facadeklasse i Entity pakken i domænelaget. |
| **MBroker** | Klasse der skaber forbindelse til MySQL databasen. |
| **PAdmin** | Brugerinterface for Administrator i systemet |
| **PManager** | Brugerinterface for manager i systemet |

**Tabel 3.8.3.1**: Singleton klasser i CSS

# 3.9 Implementering

I dette kapitel vil hele systemet blive beskrevet overordnet, mens udvalgt dele vil blive mere detaljeret beskrevet med eksempler.

## 3.9.1 Pakker og klasser i Java

*Tabel 3.9.1.1* indeholder en oversigt over de pakker og deres tilhørende java-klasser, der findes i systemet.

|  |  |
| --- | --- |
| **CSS** | |
| **Pakkenavn** | **Java-klasser** |
| **Presentation** | PManger, PAdmin |
| **Control** | ICManager, ICAdmin, CDSSHandler, CFacade CSystemStart |
| **Entity** | EFacede, EItemType, IERCS, ERCSAdapter, EComponent, EComposite, EItem, EStock, EOrder |
| **Mediator** | IMBroker, MBroker |
| **Foundation** | FDBBroker |
| **Acqaintance** | IAComponent |

**Tabel 3.9.1.1:** Pakker og java-klasser

## 3.9.2 Singleton

I dette afsnit vises et eksempel på implementering af designmønstret Singleton i CSS. Eksemplet er taget fra facadeklassen EFacade, som er facadeklasse i Entity pakken i Domain laget. Når getInstance() kaldes tjekkes der først for om der allerede findes en instansen af klassen ellers oprettes der en.

**private** **static** EFacade *instance*== **null**;

..

..

**public** **static** EFacade getInstance()

{

**if**(*instance* == **null**){*instance* = **new** EFacade();}

**return** *instance*;

}

**Kode 3.9.2.1**: Singleton eksempel fra EFacade

## 3.9.3 Observer

Som tidligere nævnt er Observer patterne anvendt for at overholde PCMEF+ arkitekturen. Dette afsnit viser implementeringen af Observer Patterne i CSS.

MANGLER

## 3.9.4 Persistens framework

Implementeringen af persistens frameworket (herefter kaldet PFW) har været en proces. Dels fordi det var 2. valg da Hibernate var for problematisk pga. nylige ændringer i implementeringen. Dels fordi selve forståelsen af de beskrevne ideer først kom undervejs og der er meget at forstå endnu. Følgende ideer og funktionalitet er implementeret.

* Ideer:
  + OID – unikt Objekt ID
  + Metamapping vha. standardmapper
* Funktionalitet
* IAEntityMapper – Interface til mapping
* IAMBroker – Interface til broker
* IAEntity – Interface til entities
* getEntity(IAEntity entity)
* putEntity(IAEntity entity)
* queryTable(String entity)

Klassediagrammet for PFW er tidligere vist under detaljeret design. Grundlaget for modellen er at alle persistensklasser implementerer et globalt interface fra acquaintance IAEntity:

**package** acquaintance;

**public** **interface** IAEntity {

**public** **int** getOID();

**public** **void** setOID(**int** OID);

}

**Kode 3.9.4.1**: IAEntity klasse fra package acquantance

Objekt ID’et OID er et unikt ID for hver enkelt instans på tværs af alle persistensklasser. Det initialiseres i constructoren til -1 og tildeles endeligt af PFW.

#### Initialisering af PFW

PFW initialiseres fra control (System Start) via interfacet IAMBroker der implementeres af klassen MBroker:

**package** acquaintance;

**public** **interface** IAMBroker {

**public** **boolean** setDB(ADBInfo info);

**public** **boolean** close();

**public** **boolean** putEntity(IAEntity entity);

**public** **boolean** getEntity(IAEntity entity);

**public** IAEntityMapper getEntityMapper();

**public** IAMBroker getBroker();

}

**Kode 3.9.4.2:** IAMBroker fra package acquantance

MBroker kalder FDBBroker (Skulle strengt taget ske via en FFacade klasse da foundation kan have anden funktionalitet end lige FDBBroker). Først oprettes forbindelse til en MySQL database med setDB() metoden, derefter intialiseres mappers gennem IAEntityMapper interfacet der implementeres af klassen FEntityMapper.

**package** acquaintance;

**public** **interface** IAEntityMapper {

**public** **static** **int** *EM\_NO\_KEY* = 0;

**public** **static** **int** *EM\_PRIMARY\_KEY* = 1;

**public** **static** **int** *EM\_FOREIGN\_KEY* = 1;

**public** **static** String *EM\_SECONDARY\_KEY* = "foreign key ";

**public** **static** String *EM\_AUTO\_INCREMENT* = "auto\_increment ";

**public** **static** String *EM\_NOT\_NULL* = "not null ";

**public** **static** String *EM\_STRING* = "String ";

**public** **static** String *EM\_INT* = "int ";

**public** **static** String *EM\_DOUBLE* = "Double ";

**public** **boolean** registerMap();

**public** **void** setSchema(String schema);

**public** **void** setTable(String tableName);

**public** **void** setEntity(String entity);

**public** **void** setRelation(String field,String colName,String type,String attributes, **int** keys);

}

**Kode 3.9.4.3:** IAEntityMapper fra package acquantance

Metoderne setSchema() og setTable() sætter hhv. database og table navne for klassen. setEntity() sættes med EMyEntity.getClass().getCanonicalName() og bruges til at identificere hvilken klasse der er knyttet til denne mapper instans. Metoden setRelation() er den mest interessante. De 5 parametre knytter en attribut i persistensklassen til en tabel i databasen:

* String field: navnet på klassens attribut (Case sensitiv f.eks. ”OID”)
* String column: navnet på den kolonne i tabellen hvor værdien af field gemmes
* String type: datatypen på field f.eks ”int” (Case sensitiv)
* String attributes attributter til dette felt i tabellen
* Int keys Primary foreign eller no key

Til sidst kaldes registerMap() der registrer mapperen med PFW. Der oprettes en mapper instans for hver klasse der skal kunne gemmes i PFW.

**public** **void** setRelation(String field, String columnName, String dataType, String attributes,**int** keys) {

**if**(relaCount<relaMax){

// ingen kontrol af de 4 felters syntaks

String vtype = dataType;**if**(vtype.contains("String")){vtype = "text";}

relations[relaCount] = **new** Relation(field,columnName,vtype,attributes);

**if**((keys==IAEntityMapper.*EM\_PRIMARY\_KEY*)){

primaryKey = "primary key (`"+columnName+"`)";

pkField = field;

pkColumn = columnName;

}

**if**((keys==IAEntityMapper.*EM\_FOREIGN\_KEY*)){/\* Not implemented\*/}

relaCount++;

}

}

**Kode 3.9.4.4:** setRelation metode

Relation er en intern klasse i FEntityMapper der holder listen af relationer for denne mappers klasse. Mapperen registreres af FDBBroker der kontrollerer og holder en liste af registrerede mappere. Ved kontrollen undersøges om databasen og tabellen eksisterer, elles dannes de i henhold til mapperens relationer. I databasen oprettes/aflæses en tabel ’oid’ for at FDBBroker kan opretholde næste ledige OID.

#### putEntity()

Herefter er PFW klar til at håndtere put og get af entity instanser. Der er lidt forskel på tilgangen til opgaven i de to metoder. Først ses på FDBBrokers putEntity() metode:

**public** **boolean** putEntity(IAEntity entity) {

// er entitys OID<0 er den ikke i Databasen endnu og skal indsættes ellers updates

FEntityMapper map = **null**;

**for**(FEntityMapper m:entities){ **if**(m.getEntity().equals(entity.getClass().getCanonicalName())){map=m;}}

**if**(!(map==**null**)){

**try** {

String s = **null**;

Field f = getField(entity.getClass(),map.getPKField());

f.setAccessible(**true**);

**if**(entity.getOID()<0){ // entity er uinitialiseret!!! entity.setOID(1+getLastOID(map.getSchema()));

s = "INSERT INTO "+map.getSchema()+".`oid`(`oid`,`entity`) VALUES ("+entity.getOID()+",'"+entity.getClass().getCanonicalName()+"');";

executeSQLLine(s);

s = "INSERT INTO "+map.getSchema()+"."+map.getTableName()+map.getColumns()+" values "+map.getValues(entity)+";";

}

**else**{s=map.getUpdateString(entity);}

executeSQLLine(s); //

}

**catch** (SecurityException e) {e.printStackTrace();}

**catch** (NoSuchFieldException e) {e.printStackTrace();}

**catch** (IllegalArgumentException e) {e.printStackTrace();}

}

**else** **return** **false**; // pågældende entity findes ikke blandt maps!

**return** **true**;

}

**Kode 3.9.4.5:** putEntity metode

Først løbes listen af Mappere igennem til den rigtige er fundet. Når det er sket undersøges om entity er nyoprettet (OID = -1) og skal insertes i sql, eller om den tidligere er tildelt en OID og skal updates i sql. Dannelsen af sql kommandoerne sker i et samarbejde mellem mapperen og FDBBroker da noget af funktionaliteten bedst ligger på mapperen idet den kender relationerne for klassen.

Der er et lille eksempel på hvordan koden har udviklet sig da putEntity fra start af blev kaldt med et Object da IAEntity ikke var defineret endnu. Dette gav anledning til mapper metoden getPKField() der efter indførelsen af IAEntity kan erstattes af ”OID”.

Kort sagt, første gang putEntity() kaldes tildeles entity en OID og insertes i tabellen, ved efterfølgende kald updates den.

#### getEntity()

Persistens klasser skal også kunne hentes fra databasen. Dette gøres med getEntity.

**public** **boolean** getEntity(IAEntity entity){

**if**(!(entity==**null**))**if**(entity.getOID()>-1){

ResultSet result = **null**;

FEntityMapper map = **null**;

**for**(FEntityMapper m:entities){ **if**(m.getEntity().equals(entity.getClass().getCanonicalName())){map=m;}}

**if**(!(map==**null**)){

**try** {

Class c = Class.*forName*(map.getEntity());

Field[] fields = c.getDeclaredFields();

result = dbstat.executeQuery(map.getQueryString(entity));

ResultSetMetaData meta = result.getMetaData();

result.first();

**for**(**int** k=1;k<= meta.getColumnCount();k++){

**for**(**int** kk=0;kk<fields.length;kk++){

**if**(map.getFieldFromColumn(meta.getColumnName(k)).equals( fields[kk].getName())){

String sValue = result.getString(k);

String sType = map.getTypeFromColumn(meta.getColumnName(k));

fields[kk].setAccessible(**true**);

**if**(sType.equals("int")) { fields[kk].setInt(entity,(**int**)Integer.*parseInt*(sValue));}

**if**(sType.equals("double")) { fields[kk].setDouble(entity,(**double**)Double.*parseDouble*(sValue));}

**if**(sType.equals("String")) { fields[kk].set(entity,sValue);}

}

}

}

**return** **true**;

}

**catch** (ClassNotFoundException e1) {e1.printStackTrace();}

**catch** (SQLException e) {e.printStackTrace();}

**catch** (IllegalArgumentException e) {e.printStackTrace();}

**catch** (IllegalAccessException e) {e.printStackTrace();}

}

**return** **false**;

}

Ende’

**Kode 3.9.4.6:** getEntity metode

Først kontrolleres at entity findes i databasen. Igen findes den rigtige mapper frem for entity og mapperen leverer en sql query streng. Kolonnerne i det hentede ResultSet løbes igennem og mapperen finder de tilhørende attributter i entity og sætter værdien vha. den tilhørende type. Hertil bruges funktionerne getFieldFromColumn() og getTypeFromColumn i mapperen.

**public** String getFieldFromColumn(String column){

**for**(**int** k=0;k<relaCount;k++){

**if**(relations[k].columnName.equals(column)) **return** relations[k].field;

}

**return** **null**;

}

**Kode 3.9.4.7:** getFieldFromColumn metode

#### queryTable()

En sidste vigtig funktionalitet der er lagt på PFW er queryTable(). Den bruges til at intialisere systemet og hente Resultset ud fra databasen for hver persistens klasse. Det er en ’dirty’ metode forstået på den måde at den ikke udtrykker ånden i et persistens framework. Der er intet lazy initialization over den. ’En milliard records’ problematikken anvendt på PFW fortæller om en System start der nærmest låser databasen indtil alle records er læst ud. Dette er bestemt en ting der skal arbejdes videre på i senere iterationer.

**public** ResultSet queryTable(String entity){

FEntityMapper map = **null**;

**for**(FEntityMapper m:entities){**if**(m.getEntity().equals(entity)){map=m;}}

**if**(!(map==**null**)){

String s = "SELECT \* FROM "+map.getSchema()+"."+map.getTableName()+";";

**try** {**return** dbstat.executeQuery(s);}

**catch** (SQLException e) {e.printStackTrace();}

}

**return** **null**;

}

**Kode 3.9.4.8:** queryTable metode

PFW er et godt eksempel på genanvendelig kode! Det kan i princippet anvendes på nye systemer uden at skulle ændres. Det er blot en første begyndelse på og indblik i de muligheder der er for at lave genanvendeligt design der er let at udvide på og har været meget mere spændende end hvis vi blot havde anvendt Hibernate eller andre persistens frameworks.

## 3.10.1 Forbindelse mellem DSS og CSS

Som beskrevet tidligere skal der laves en forbindelse mellem klient og server, for dette kan lade sig gøre skal der bruges et port nummer, vi har valgt til at være 55555. Nummeret i sig selv er ikke vigtigt, det er bare vigtigt ikke at bruge et nummer, som andre kendte processer bruger. Serveren laver så en ny serverSocket med den valgte port, og venter derefter på en klient som tilsluttet. Når en klient opretter forbindelse sker det blev hjælp af klassen ClientHandler. For at flere klienter kan tilslutte på samme tid, bliver denne klasse kørt i den tråd for sig selv. Hver af disse tråd med ClientHandler’e har en scanner tildelt, og da vi kun forventer at modtage en type beskeder, så scanner scanneren beskeden og sender den videre til EFacade.

Klienten starter med at anmode om en forbindelse til serveren, ved hjælp af port nummeret. Når forbindelsen er oprettet kan klienten så sende en besked ved hjælp af en PrintWriter. Klienten er kun lavet som en test klasse, så vi kunne teste server klassen.

#### Behandling af besked

Efter en besked er sendt videre til EFacade, er den dens opgave at lave en ordre ud fra den. Dette sker ved at beskeden bliver splittet med parameteren ”;”. Når beskeden er splittet op er de nødvendige parameter, til at konstruere en ordre med, til rådighed. Efter ordren er skabt, iterere metoden hen ovre dens itemtypes og det givne antal Items af hver ItemType bliver reserveret og knyttet til ordren.

//Creates a Order from at requestOrder

**public** **boolean** createOrder(String packetInfo) {

String[] tempPacketInfo;

tempPacketInfo = packetInfo.split(";");

**if** (tempPacketInfo[0].equals("01")) {

EOrder tempOrder = **new** EOrder(tempPacketInfo[1], tempPacketInfo[3]);

**for** (**int** k = 4; k < tempPacketInfo.length; k = k + 2) {

**int** currentNumberOfItems = Integer.*parseInt*(tempPacketInfo[k + 1]);

**for** (**int** j = 0; j < currentNumberOfItems; j++) {

tempOrder.add(stock.remove(tempPacketInfo[k]));

}

}

orders.add(tempOrder);

**return** **true**;

} **else** {

**return** **false**;

}

}

**Kode 3.9.1.1**: CreateOrder metode

# 3.10 Test

Test fortages under hele implementeringen og efter endt implementering. Der er forskellige testformer og i dette afsnit vil de forskellige former blive beskrevet generelt og i forhold til implementeringen af CSS.

## 3.10.1 Testtyper

* Unitest (modultest)
* Integrationstest
* Strukturel test
* Evaluering af designmål
* Ydeevnetest
* Funktionel test
* Accepttest

#### Unitest

Unittest eller modultest bruges undervejs i implementeringen af de enkelte dele af systemet. Det vil sige at man tester helt ned på metoder i klasser. Når man tester klasser og metoder så laves der testcases der beskriver input og ventet output for metoden. Man deler testværdier ind i grupper, der har samme adfærd og tester derefter kun på en eller få værdier fra hver gruppe for ikke at lave for mange ens test. Det vigtigt at udvælge testdata, så man får testet grænseværdier for metoden.

#### Integrations- og struktureret test

Her testes moduler og komponenter sammen for at sikre at grænseflader mellem de enkelte dele fungerer som det er tilsigtet. Unittest test cases kan bruges og der kan tilføjes nye for test af grænsefladerne.

#### Ydeevne og Funktionel test

Tester hele systemet i sammenhæng og bestemmer om kravene er opfyldt. Ydeevne test skal opfylde de ikke-funktionelle krav i forhold til fx Stres-, volume- og kvalitetstest. Funktionel test er test af de funktionelle krav. Testcases skrives ud fra Use Cases og krav til systemet.

#### Accepttest

Test af det samlede system og bestemmer om alle krav er opfyldt og systemet er klar til brugeren.

## 3.10.2 JUnit

Til automatisering af test kan bruges JUnit, som er et open source framework, der bruges til at teste Java kode med. Når test-metoder/klasser er skrevet køres alle testen ved en kommando og det er kun i de tilfælde, hvor testen fejler at man får besked. Det betyder at man ikke skal kigge outputdata igennem manuelt for at søge efter fejl i metoder.

## 3.10.3 Test af CSS

MANGLER

## 3.10.4 Evaluering af designmål

#### Driftssikkerhed

**Systemet skal være robust så det ikke går ned ved f.eks. ved fejlagtigt brugerinput**

**Systemet skal være pålideligt. Overensstemmelse mellem specificeret og observeret adfærd**

**Systemet skal være tilgængeligt. Den del af tiden hvor systemet ikke kan bruges til udførelse af normale opgaver skal være minimal**

#### Anvendelighed

**Systemet skal være brugervenligt og intuitivt**

Vi udførte denne test ved hjælp af Andreas Rune Fugl, som er Ph.d. studerende ved instituttet. Vi satte testen op som et rollespil, hvor Andreas blev instrueret hvilken rolle han havde i det centrale lager. Derefter fik han lov til at prøve programmet, som han så kom med feedback på.

Han mente selve programmeret var overskueligt og let at finde rundt i, i både admin delen og manager delen. Han havde dog flere forespørgsler, som han mente ville gøre admin delen mere brugbar. Disse forespørgsler var blandt andet: en informativ startside, en log og ændring af navnet på en menu.

På baggrund af denne test, vil vi mene at brugervenlighed af programmet er acceptabelt, men der er stadig plads til forbedring.

**Systemet skal støtte brugeren i dennes arbejde**

Det er vigtigt at systemet hjælper brugeren i de daglige arbejdsgange. Det vil sige det skal understøtte arbejdsgange. Vi har ikke en rigtig bruger at evaluere dette designmål med.

# 3.11 Datakommunikation

Til forbindelsen mellem DSS og CSS har vi valgt at bruge Transmission Control Protocol(TCP), som er en pålidelig protokol. Vi mente i lang tid at forbindelsen skulle have været lavet med User Datagram Protocol(UDP) som er en upålidelig protokol. Forskellen mellem en pålidelig og upålidelig protokol, ligger i servicen Error Control som TCP har, og UDP ikke har. Error Control vil sige at modtageren kvitterer for hver pakke processen modtager og kontrollerer for at en pakke til bliver modtaget flere gange.

Grunden til at vi mente UDP kunne bruges til vores program, er at protokollen normalt bliver brugt til at sende små beskeder, og det var sådan vi mente kommunikation mellem DSS og CSS skulle være. Så en orderRequest pakke bare ville være en specielt formateret besked, som CSS kunne konstruere en ordre ud fra. Det ville også kunne virke hvis en orderRequest pakke blev mistet, men det ville give et problem hvis en kvittering forsvandt. Så ville ordren blive lavet, men kunden vil aldrig få en kvittering. Når kunden så gensender en orderRequest pakke, vil der blive lavet en ordre for meget.

Når en TCP klient forbinder til TCP server, bliver der oprettet et virtuelt miljø hvor de kommunikere. TCP benytter en service som bliver kaldt stream delivery til at sende en strøm af data, igennem det virtuelle miljø. Derfor skal TCP serveren lyttet efter klienter, som den derefter skal lave en forbindelse med. Når denne forbindelse skal laves er det nødvendigt er bruge et port nummer.

Efter det blev besluttet at bruge TCP i stedet for UDP, blev formatet for besked lavet. Den består af følgende felter:

* Besked type – Dette felt indeholder en String med en værdi. Hvis værdien er 01 er der tale om en orderRequest. Dette felt er lavet i tilfælde at systemet skulle kunne modtage flere forskellige typer af beskeder.
* Butik navn – Dette felt indeholder en String med navnet på kundens butik.
* Krypteringsfelt – Dette er valgfrit, det er tænkt som en mulighed for senere at implementere et krypterings system i CSS. I øjeblikket er det ikke implementeret, så default værdien er 0000.
* Leveringsdato – Dette felt indeholder en String på en dato i formatet år-måned-dag(yyyy-mm-dd), som fortæller hvornår en butik vil have varer leveret.
* Itemtype barcode – Dette felt indeholder en String med barcoden for et item
* Itemtype antal – Dette felt indeholder en string med antallet af den tidligere nævnte itemtype.

Disse 5 felter skal altid være til stede i en besked til vores system. Beskeden kan også indeholde flere itemtyper, hvis man vil bestille flere typer items. Hvis dette er tilfældes skal der tilføjes 2 felter mere, første med itemtype barcode og næste med itemtype antal. På denne måde kan man tilføje så mange itemtypes man ønsker. Hver felt skal være separeret af et semikolon

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Besked type | Butik navn | Krypteringsfelt | Itemtype barcode | Itemtype antal | (valgfrit)  Itemtype barcode | (valgfrit)  Itemtype antal |

Et eksempel på en besked er:

01;IKEA Odense;0000;2012-12-12;123456789999;2;223456789999;3;323456789999;1

Her bestiller IKEA Odense 2 varer med barcoden 123456789999, 3 vare med barcoden 223456789999 og lidt sidst 1 vare med barcoden 323456789999 til levering 12/12-2012.

# 3.12 Styring regulering og overvågning

Til simulering af robotten (RCS) på lageret har vi valgt en PLC. Der kommunikeres med PLC’en over Serielport via et RS232-kabel. Da der ikke længere er indgang for RS232-stik i computere er en USB-til-RS232-konverter anvendt. Fra CSS sendes tre beskeder til RCS nemlig *scanItem*, *storeItem* og *retreiveItem*. Forbindelsen mellem *CSS* og *RCS* skabes via en *RCS* interface *IERCS* i *Entity* pakken i *Domain*-laget.

## 3.12.1 PLC’en

PLC’en skal simulere robotsystemet og i forbindelse med scanItem sættes en vare på transportbåndet og *Manageren* vælger, gennem *CSS,* scanItem og kommandoen sendes til PLC’en, hvorefter transportbåndet kører varen hen forbi stregkode-skanneren, hvor den skannes. RCS sender stregkoden sendes tilbage til *CSS*.

Item

Stregkode - Skanner

**Figur 3.12.1.1**: PLC – transportbånd med skanner

## 3.12.2 Protokol

Der defineres en protokol for kommunikation mellem *CSS* og *RCS*. Protokollen er simpel og indeholder kun information om metode-kald og parametre. Data der sendes fra *CSS* til *RCS* er af formatet ”metode:parameter/13” fx ”storeItem:stockPosition/13”. Retur sendes ”metode:returværdi/13” fx ”scanItem:barcode/13”. Data sendt mellem *CSS* og *RCS* fortolkes på hver side.

## 3.12.3 Implementering på CSS siden

I *Kode 3.13.2* ses hvordan forbindelsen via serielporten åbnes og sættes op. Det er vigtigt at opsætningen er den samme for *RCS* og *CSS*. Under setSerialPortParams følgende parametre for kommunikationen:

* Baudraten til 19200 bits pr. sekund
* SerialPort.DATABITS\_8 fortæller hvor mange data-bits der sendes pr. pakke.
* SerialPort.STOPBITS\_1 fortæller hvornår en pakke er slut.
* SerialPort.PARITY\_NONE Parity er en bit der kan sættes for at hjælpe med til at finde fejl, men NONE betyder at den ikke er sat.

CommPortIdentifier portIdentifier = CommPortIdentifier.*getPortIdentifier*(portName);

**if** ( portIdentifier.isCurrentlyOwned() )

{

System.*out*.println("Error: Port is currently in use");

}

**else**

{

CommPort port = portIdentifier.open("CSS",2000);

**if** ( commPort **instanceof** SerialPort )

{

SerialPort serialPort = (SerialPort) port;

serialPort.setSerialPortParams(19200,SerialPort.*DATABITS\_8*,SerialPort.*STOPBITS\_1*,SerialPort.*PARITY\_NONE*);

**Kode 3.12.3.1:** Opsætning og åbning af Serielport

Når Serielporten er åben og sat op kan der oprettes to streams en inputStream og en outputStream til kommunikation mellem RCS og CSS.

InputStream in = serialPort.getInputStream();

OutputStream out =serialPort.getOutputStream();

**Kode 3.12.3.2:** Opsætning og åbning af Serielport

For at kunne sende beskeder, vente og lytte på feedback fra PLC’en implementeres in- og outputstreams i tråde. Så længe der er noget der skal sendes kører tråden.

message = message + "/ ";

**byte**[] test = message.getBytes();

test[test.length-1] = 13;

**this**.out.write(test);

**Kode 3.12.3.3:** Sender besked til PLC

PLC’en ved at den er færdig med at modtage data når den læser en ”/13” og det er derfor vigtigt at hver besked fra CSS til PLC’en ender med ”/13”.

## 3.12.4 Implementering på RCS siden

#### Ladder

#### Struktureret text

**RCSLoop - Structured Text Language Page 1**

RCS:RCSControl:RCSMainControl 18-12-2012 00:15:39

Structured Text C:\RSLogix 5000\Projects\CSS\RCS\_Robot\_Control\_System.ACD

RCSLoop RSLogix 5000

if RCSH then // Kontroller message

Scan.LEN := 10; // scanItem: + chr(13)

Scan.data[0] :=32+83;

Scan.data[1] :=32+67;

Scan.data[2] :=32+65;

Scan.data[3] :=32+78;

Scan.data[4] := 73;

Scan.data[5] :=32+84;

Scan.data[6] :=32+69;

Scan.data[7] :=32+77;

Scan.data[8] :=58;

Scan.data[9] :=13;

bc.LEN := 13; // ?123456789999 +chr(13)

for j:=1 to 8 do bc.data[j] := 49+j; end\_for;

bc.data[9] :=57;

bc.data[10] :=57;

bc.data[11] :=57;

bc.data[12] :=13;

// Test for 'scanItem:' message

if MsgBuffer.LEN = 10 then

scanbool := 1;

for j:=0 to 9 do

if not (MsgBuffer.data[j] = Scan.data[j]) then

scanbool := 0;

end\_if;

end\_for;

if scanbool then

// 'Find' en barcode

barcodes := T3.ACC mod 3;

case barcodes of

0 : bc.data[0] :=49; // barcode1 = 12345678999

1 : bc.data[0] :=50; // barcode2 = 22345678999

2 : bc.data[0] :=51; // barcode3 = 32345678999

end\_case;

Scan.Len := Scan.LEN-1;

CONCAT(Scan,bc,OutBuffer);

AWT232\_Control.LEN := OutBuffer.LEN;

end\_if;

end\_if;

RCSM :=1;

RCSH :=0;

end\_if;

123456789

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

56

## 3.12.5 Beskrivelse af kommunikationen

I dette afsnit beskrives kommunikationen mellem CSS og RCS. Efter der er skabt forbindelse mellem *CSS* og *RCS* sker følgende hændelser i forbindelse med fx *scanItem*:

1. CSS sender beskeden ”scanItem:/13” til RCS.
2. RCS skanner item’et
3. RCS sammen sætter en message i formatet ”scanItem:barcode/13”
4. RCS sender message til CSS.

## 3.12.4 Problem med kommunikation med PLC

Vi skabte forbindelse og udførte kommunikation mellem CSS og PLC’en, men desværre var det ikke muligt at få den skanner, der sidder på PLC’en til at virke. PLC’en har kun en Seriel Kommunikationsport og denne bruges til kommunikation mellem PLC og CSS.

Stub

# 3.13 Deployering

CSS består af 2 klienter, en robot og en server med tilhørende GUI. De 2 klienter er DSS klienten og Manager klienten. DSS klienten er en tynd klient som ligger i et andet system, som en detail store benytte. Disse klienter skal kun have en mulighed for at se CSS’s varer, og at kunne sende en orderRequest. Vi forventer denne klient vil blive kørt på en normal computer. Manager klienten er en tykkere klient, som vi helst ville se kører på en håndholdt tablet. Denne klient skal gøre det muligt for manageren at se aktive ordre som skal håndteres, samt at sætte varer på lager.



**Figur 3.13.1.1**: PLC – transportbånd med skanner

# Del IV: Epilog

# 4.1 Konklusion

Igennem projekt er der blevet udviklet på et multiklient software system til håndtering af varer og ordre på et central lager.

Ud af de 5 produkt mål vi lavede i vores projektgrundlag er 3 til dels blevet opfyldt, mens de 2 sidste ikke er nået et stadie vi er tilfredse med. Vi har fået lavet et program til at håndtere varer, dog mangler der stadig meget funktionalitet, men det var forventet. Vores system er lavet som en klient/server og der anvendes netværksprotokoller til at gøre muligt.

Med hensyn til vores systems anvendelse af automation i form af et PLC system, så har vi arbejdet imod at skabe en forbindelse til en PLC, men der blev aldrig konstrueret noget reelt brugbart for PLC systemet at gøre.

Til sidst havde vi som punkt at skrive en god rapport, men

# 4.2 Perspektivering

MANGLER

Ved projektets afslutning havde vores system langt mindre funktionalitet, end vi havde forventet i opstartsfasen. Men de funktioner som er tilgængelige er essensen af hele systemet, fra bruger input til robot output. Vi havde i inceptions fasen forventninger om at nå 2 elabarations faser, hvor vi ville have alle vores Must use cases i første og derefter vores Should use cases i anden. Dette viste sig at være et urealistisk mål, da de essentielle var en større mundfuld end forventet. Vores estimat for disse use cases ødelagde dermed meget af vores planlægning og til sidst følte vi os tvunget til at aflyse 2. elabration.

# 4.3 Refleksioner

## 4.3.1 Projektetablering

MANGLER

## 4.3.2 Planlægning

MANGLER

Vores planlægning gik fra at være meget lidt detaljeret til at være ikke eksisterende. Under etableringen lavede vi en overordnet tidsplan, hvor vi beskrev hvad vi skulle lave i de kommende uger. Denne plan tog ikke højde for tidsforbruget i de forskellige uger, så vi mente vi kunne indhente tabt tid i uger med mere gruppe arbejde. Resultatet blev at tidsplanen skred, til at du afleveringens datoen var et fast standpunkt. ??

## 4.3.3 Gennemførsel

MANGLER

## 4.3.4 Afrapportering

MANGLER

# 4.4 Litteraturliste

(BRU) Bernd Bruegge & Allen H. Dutoit: *Object-Oriented Software Engineering. Using UML, Patterns and Java*. 3rd edition. Pearson 2010.

(LAR) Craig Larman: *Applying UML and Patterns. An introduction to Object-oriented Analysis and Design and the Unified Process*. 3PrdP edition. (Chapter 38).

(MAC) Leszek Maciaszek, Bruc Lee Liong: *Practical Software Engineering: A Case-Study Approach*. Addison-Wesley, 2004 ISBN: 0321204654. Format: Paper Bound w/CD-ROM. (Chapter 9).

(REN) Klaus Renzel & Wolfgang Keller**:** *Client/Server Architectures for Business Information Systems. A Pattern Language*. <http://www.sdm.de/g/arcus>)

(SOM) Sommerville: Software Engineering, chapter 18 and 19

(CHR1) Henrik Bærbak Christensen: Configuration management

(CHR2) Henrik Bærbak Christensen: Frameworks

(CHR3) Henrik Bærbak Christensen: Test

Seriel og ASCII manipulation : Communicate with an ASCII Device

# Bilag

# Appendix A - Projektgrundlag

# 1. Projektgrundlag

## 1.1 Interessenter i projektet

* *Projektgruppen*: Der skal bedømmes ud fra projektforløbet og resultat af dette.
* *Steffen Peter Skov*: Der er gruppens vejleder og lige ledes skal vurdere rapporten.
* *Censor*: Der skal vurdere rapporten og systemet.

# Projektvejleder

Steffen Peter Skov har været projektvejleder på projektet og vi har løbende haft møder med ham efter behov. Steffen har givet konstruktiv kritik.

## 1.2 Projektgruppen

Projektgruppen består af fire personer: Anders Kold, Henning Fich, Nico Rasmussen

og Kristina Hussak. Alle i projektgruppen vil deltage aktivt i projektarbejdet.

## 1.3 Beskrivelse

Central Stock System (CSS).

Et nyt centrallager skal implementere et system til styring af varer: Bestilling, Modtagelse, lager og udlevering samt grundlæggende bogholderi.

Bestilling af vare til lageret foretages hos eksterne leverandører og registreres, som ”bestilt” i systemet.

En vares rejse gennem centrallageret forløber som følger: Ved ankomst til varemodtagelsen (EUR paller) scannes varen ind med stregkode eller QR kode og køres i dybdereol. Et robotstyret lager sørger automatisk for at varen opbevares forsvarligt til den skal udleveres. Varer der skal kasseres skal scannes ud af systemet, som kasserede.

Når en butik ønsker at bestille vare fra lagret, sender de en bestilling til systemet.

Vare bestilt i systemet bliver hentet på lager og gjort klar til pakning. Ved udlevering bliver varen udskrevet fra lageret, og Systemet sørger for bogføring.

Systemet skal være indlysende at bruge for personalet(brugervenligt) og stabilt, det må bl.a. ikke gå ned på brugerfejl. Der skal laves backup en gang pr. døgn. Systemet skal være så uafhængig af platform som muligt.

## 1.4 Formål

At fremstille et System der kan varetage den grundlæggende funktionalitet for drift af et centrallager. Systemet skal medvirke til nedbringelse af driftsomkostninger gennem automatisering af arbejdsprocesser.

## 1.5 Mål

### 1.5.1 Produktmål

Målet for produktet er:

1. at få udviklet et system, der kan håndtere og administrere varer på et lager.
2. at få udviklet et distribueret system (f.eks. Client/Server)
3. at få udviklet et system der indbefatter automation (PLC)
4. at anvende netværksprotokoller i kommunikationen mellem de enkelte arbejdsstationer i systemet.
5. at få det beskrevet i en god rapport.

### 1.5.2 Procesmål

Ved projektets afslutning skal det enkelte gruppemedlem alene og i samarbejde med andre:

1. være blevet bedre til at bruge de udviklingsværktøjer vi lærer om (UP, Scrum) til systemudvikling.
2. kunne opbygge og forstå en netværksbaseret softwareløsning.
3. have forståelse for softwarearkitekturen og have fokus på genbrugeligt design.
4. kunne anvende versionsstyringssystem til versionering af dokumenter og kildetekster.
5. have opnået større erfaring med MySQL, Java, netværksprotokoller og automation.
6. kunne lave en holdbar projektplan så milepæle og artefakter bliver klar til tiden.
7. have fået gode erfaringer med udvikling af et distribueret system.

## 1.6 Inddragelse af fagområder

* SRO bruges i forbindelse med lagerrobot og implementering i et distribueret system.
* SUD bruges i forbindelse med udvikling af systemets forskellige dele og fastlæggelse af systemets arkitektur og design.
* KOM bruges i forbindelse med socket programmering i det distribuerede system.

## 1.7 Samarbejdet

Projektgruppen skal arbejde som en fladt struktureret gruppe, hvor alle er aktive og tager hånd om projektet.

Der afholdes møde fast hver torsdag kl. 12.15 med mulighed for vejledning (i løbet af mødet). Der laves dagsorden og referat. Der er mødepligt. Yderligere møder i den kommende uge besluttes her. Arbejdsopgaver til næste møde uddelegeres.

Konflikter håndteres på en fornuftig måde, efterhånden som de opstår.

## 1.8 Dokumenter

Rapporten skrives i Word, mens diagrammer laves i Cacoo. Alle dokumenter gemmes i fælles Dropbox.

Der arbejdes i iterationer og der arbejdes på rapporten undervejs. De enkelte artefakter til rapporten skrives undervejs og lægges i særskilte dokumenter. Ved afslutning samles alle dokumenter til en samlet rapport.

Kildekode versionsstyres gennem GitHub som er et Open Source versionsstyringssystem.

## 1.9 Ressourcer

Automations delen er ikke endeligt fastlagt men vi forestiller os enten at bruge BradleyAllen Logix5000 eller Lego Mindstorm til at visualisere automation af et varelager. Der bruges alm. standard værktøjer til produktion af artefakter (Word til dokumenter, Cacoo til diagrammer, MySQL til Database osv.)

Til rådighed under projektet er vejleder Steffen Peter Skov samt vores undervisere indenfor de respektive områder – PLC, netværk og systemudvikling.

## 1.10 Projektstyring

Der anvendes UP til den overordnede systemudvikling (iterationer, artefakter). Scrum vil blive afprøvet som redskab under inceptionsfasen.

## 1.11 Versionsstyring

Til versionsstyring er GitHub blevet brugt. I opstartsfasen brugte vi dropbox til udveksling af dokumenter.

## 1.12 Den overordnede projektplan

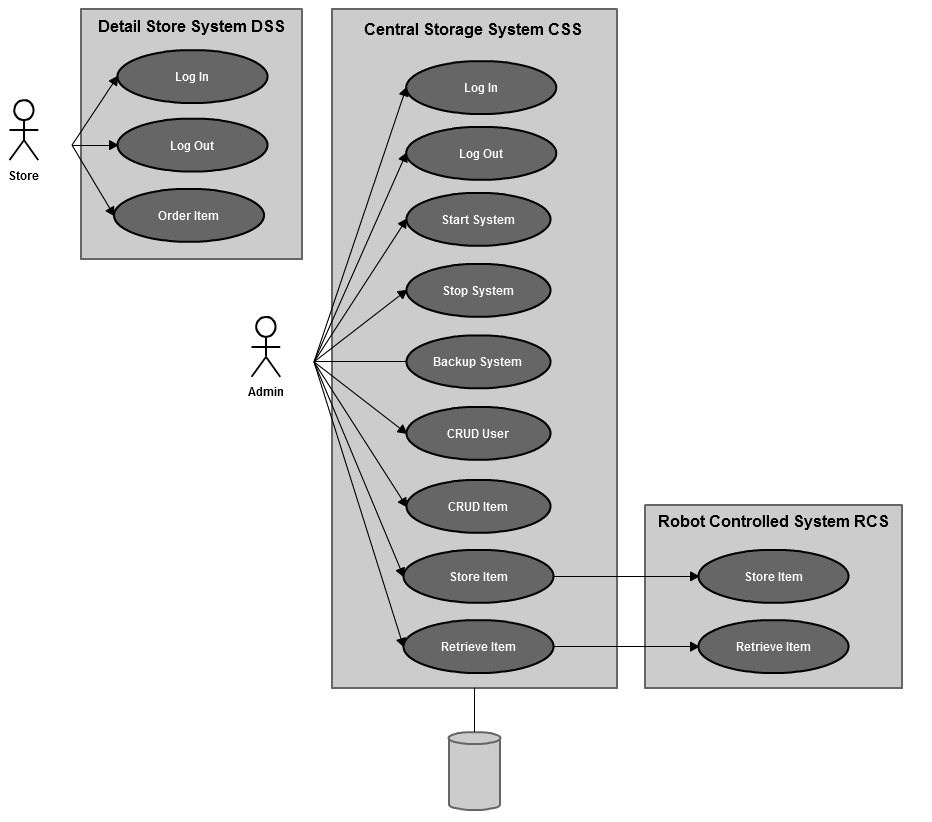
Den overordnede planlægning for projektets forløb fra start til slut.

|  |  |  |
| --- | --- | --- |
| **Uge** | **Emne** | **Beskrivelse** |
| **40** | Projektgrundlag |  |
| **41** | Inception – 1. Iteration |  |
| **42** | **EFTERÅRSFERIE** |  |
| **43** | Inception – 1. Iteration |  |
| **44** | Elaboration – 1. Iteration |  |
| **45** | Elaboration – 1. Iteration |  |
| **46** | Elaboration – 2. Iteration |  |
| **47** | Elaboration – 2. Iteration |  |
| **48** | Konstruktion |  |
| **49** | Konstruktion |  |
| **50** | Rapport |  |
|  |  |  |
| **AFLEVERING** | | |

## 1.13 Plan for inceptionsfasen

|  |  |  |
| --- | --- | --- |
| **Projektplan** | **Start Dato** | **Slut Dato** |
| **Projektgrundlag** |  |  |
| Projektintroduktion | **10/09** | **21/09** |
| Case-introduktion | **17/09** | **21/09** |
| Problemstilling | **17/09** | **21/09** |
| Problemformulering | **17/09** | **21/09** |
| Målsætning | **17/09** | **05/10** |
| * Produktmål | **17/09** | **05/10** |
| * Procesmål | **17/09** | **05/10** |
| Projektafgrænsning | **24/09** | **05/10** |
| **Projektværktøjer og -styring** |  |  |
| Rapportskrivning | **24/09** | **05/10** |
| Udviklingsmiljø | **24/09** | **05/10** |
| UML modellering | **24/09** | **05/10** |
| Database | **24/09** | **05/10** |
| Procesmodel | **24/09** | **05/10** |
| **Inception** |  |  |
| Produktvision | **17/09** | **05/10** |
| Funktionelle krav | **24/09** | **05/10** |
| Ikke-funktionelle krav | **24/09** | **05/10** |
| Aktørliste | **24/09** | **05/10** |
| Use-case diagram(brugsmønster) | **24/09** | **09/10** |
| Prioritering | **24/09** | **11/10** |
| Milepæle | **08/10** | **23/10** |
|  |  |  |
|  |  |  |

## 1.14 Use-case Diagram



# Appendix B - High-level Use Cases

|  |  |
| --- | --- |
| View Orders | |
| Use case: | View Orders |
| Actors: | Stock & Admin |
| Purpose: | Display a list of orders for the user to interact with. |
| Description: | The System constructs a list with all of the incoming orders and makes the list available on display for the user. |

|  |  |
| --- | --- |
| Store Item | |
| Use case: | Store Item |
| Actors: | Stock |
| Purpose: | Store a specific Item on its respective position. |
| Description: | The item is scanned and its position is retrieved from CSS and forwarded to the RCS (Robot Controlled System). The Item is then moved to its respective position and its information is stored in the database. |

# Appendix C - Expanded Use Cases

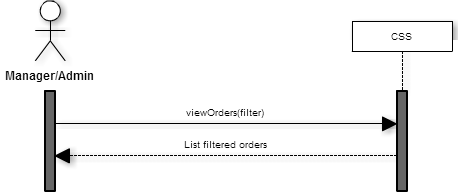
|  |  |  |  |
| --- | --- | --- | --- |
| **Store Item** | | | |
| **Use case:** | | **Store Item** | |
| **Actors:** | | Manager, RCS | |
| **Purpose:** | | Store a specific Item on its associated stock position. | |
| **Overview:** | | The item is scanned and its stock position is retrieved from CSS and forwarded to the RCS (Robot Controlled System). The Item is then moved to its associated stock position and its information is stored in the database. | |
| **Type:** | | Essential | |
| **Preconditions:** | | The item type must exist in the system | |
| **Postconditions:** | | The item is added to storage, and the System is updated. | |
| **Special Req.:** | | No special requirements needed. | |
|  | |  | |
| **Flow of Events** | | | |
| **Actor Action** | | | **System Response** |
| 1. The use case starts when Manager puts an item on a storage buffer, and clicks "Store Item" | | | 1. RCS: The conveyer belt transports the item to a scanner, and scans the item. the scanned info is send to CSS |
|  | | | 1. CSS: Looks up the item, using the scanned info, and sends the stock position back to the RCS |
|  | | | 1. RCS. Places the item on its position, and returns a confirmation to CSS. |
|  | | | 1. CSS: updates stock and returns a confirmation to the Manager. |
|  | | |  |
| **Alternative Flow of Events** | | | |
| **Line 2:** | Item type doesn’t exist in the system. Return Item to storage buffer, and notify the Manager. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| View Orders | | | |
| Use case: | | **View Orders** | |
| Actors: | | Admin & Manager | |
| Purpose: | | Display a list of orders for the user to interact with. | |
| Overview: | | The System constructs a list with all orders and makes the list available on display for the user. | |
| Type: | | Essential | |
| Preconditions: | | The user must be known to the System. | |
| Postconditions: | | A list of orders is made available on display for the user. | |
| Special Req.: | | No special requirements needed. | |
|  | |  | |
| Flow of Events | | | |
| Actor Action | | | **System Response** |
| 1. This use case starts when the user requests access to the list of orders. | | | 1. System produces a list of all orders requested and makes it available on display. |
|  | | |  |
| Alternative Flow of Events | | | |
|  |  | | |

# Appendix D - Systemsekvensdiagrammer og Systemoperationskontrakter

#### View orders

Manager eller Administrator vælger at se hvilke ordre der findes i systemet med en given status (filter) (åben eller lukket)

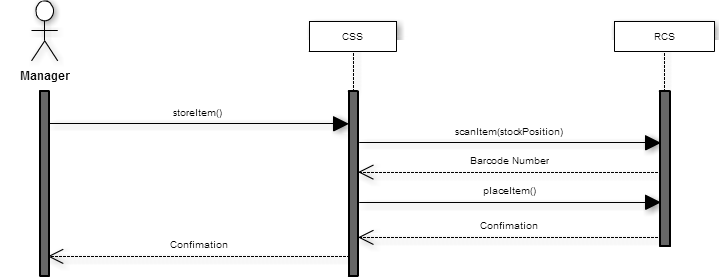


**Figur 3.6.1 :** View orders

|  |  |
| --- | --- |
| UC: View Orders | |
| Contract Name: | viewOrders(orderState) |
| Responsibilities: | Retrieve order information from database and create the instances. |
| Type: | System |
| Exceptions: | No exceptions for this contract. |
| Output: | A list of all orders and their items. |
| Preconditions: | No preconditions needed for this contract. |
| Postconditions: | Create an instance of an Order for every order with the chosen state. For each order create an instance of every Item within that order. |

#### Store Item

Manager modtager en ny vare og ønsker at sætte den på lager. Han vælger *Store Item* og CSS sender besked til RCS om at skanne og dernæst placere varen på lageret på en given position.



**Figur 3.6.2 :** Store item

|  |  |
| --- | --- |
| UC: Store Item | |
| Contract Name: | storeItem() |
| Responsibilities: | CSS requests RCS to scan and store an Item at a given position. |
| Type: | System |
| Exceptions: | ItemType doesn’t exist in the system (invalid barcode). Return Item to storage buffer, and notify the Manager. |
| Output: | A confirmation of either success or failure or returned. |
| Preconditions: | No preconditions needed for this contract. |
| Postconditions: | An instance of Item was created and given a free StockPosition and scanned Itemtype. State of the StockPosition is changed to ‘full’. |

# Appendix E - Design sekvensdiagrammer