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 ordre, der er oprettet i systemet. Arbejdsprocesser i forbindelse med at sætte nye vare 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

Denne 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.

Indholdsfortegnelse

[Del I Prolog 2](#_Toc343504621)

[1.1 Synopsis 3](#_Toc343504622)

[1.2 Forord 3](#_Toc343504623)

[1.3 Læsevejledning 7](#_Toc343504624)

[1.4 Indledning 8](#_Toc343504625)

[1.4.1 Beskrivelse 8](#_Toc343504626)

[1.4.2 Formål 8](#_Toc343504627)

[1.4.3 Mål 8](#_Toc343504628)

[1.4.4 Inddragelse af fagområder 9](#_Toc343504629)

[Del II Procesmodel 10](#_Toc343504630)

[2.1 Projektværktøjer 11](#_Toc343504631)

[2.1.1 Versionsstyring 11](#_Toc343504632)

[2.1.2 Cacoo 11](#_Toc343504633)

[2.2 Projektstyring 11](#_Toc343504634)

[2.2.1 Unified Process 11](#_Toc343504635)

[2.2.2 SCRUM 12](#_Toc343504636)

[2.3 Den overordnede projektplan 12](#_Toc343504637)

[2.4 Det faktiske forløb i projektet 13](#_Toc343504638)

[Del III Resultater 14](#_Toc343504639)

[3.1 Vision 15](#_Toc343504640)

[3.2 Krav til systemet 16](#_Toc343504641)

[3.3 Aktørliste 16](#_Toc343504642)

[3.4 Use Cases 17](#_Toc343504643)

[3.4.1 Prioritering af Use cases 18](#_Toc343504644)

[3.4.2 High-level Use Cases 18](#_Toc343504645)

[3.4.3 Expanded Use Cases 19](#_Toc343504646)

[3.5 Klassediagram 22](#_Toc343504647)

[3.6 Systemsekvensdiagrammer og Systemoperationskontrakter 22](#_Toc343504648)

[3.8 Softwarearkitektur 25](#_Toc343504649)

[3.8.1 Designmål 25](#_Toc343504650)

[3.8.2 Arkitektur - PCMEF+ framework 26](#_Toc343504651)

[3.8.3 Anvendte designmønstre 29](#_Toc343504652)

[3.9 Detaljeret Design 34](#_Toc343504653)

[3.9.1 Design sekvensdiagrammer 34](#_Toc343504654)

[3.9.2 Persistens framework 37](#_Toc343504655)

[3.9.3 Singleton pattern 38](#_Toc343504656)

[3.9.4 Database design 38](#_Toc343504657)

[3.10 Implementering 39](#_Toc343504658)

[3.10.1 Pakker og klasser i Java 39](#_Toc343504659)

[3.10.2 Singleton 39](#_Toc343504660)

[3.10.3 Observer 39](#_Toc343504661)

[3.10.1 Forbindelse mellem DSS og CSS 40](#_Toc343504662)

[3.11 Test 40](#_Toc343504663)

[3.11.1 Testtyper 41](#_Toc343504664)

[3.11.2 JUnit 41](#_Toc343504665)

[3.12 Datakommunikation 41](#_Toc343504666)

[3.13 Styring regulering og overvågning 42](#_Toc343504667)

[3.13.1 PLC’en 43](#_Toc343504668)

[3.13.2 Protokol 43](#_Toc343504669)

[3.13.3 Implementering 43](#_Toc343504670)

[3.13.4 Problem med kommunikation med PLC 44](#_Toc343504671)

[3.14 Deployering 45](#_Toc343504672)

[Del IV Epilog 46](#_Toc343504673)

[4.1 Konklusion 47](#_Toc343504674)

[4.2 Perspektivering 47](#_Toc343504675)

[Bilag 48](#_Toc343504676)

[Appendix A 49](#_Toc343504677)

[1. Projektgrundlag 49](#_Toc343504678)

[1.1 Interessenter i projektet 49](#_Toc343504679)

[Projektvejleder 49](#_Toc343504680)

[1.2 Projektgruppen 49](#_Toc343504681)

[1.3 Beskrivelse 49](#_Toc343504682)

[1.4 Formål 49](#_Toc343504683)

[1.5 Mål 50](#_Toc343504684)

[1.5.1 Produktmål 50](#_Toc343504685)

[1.5.2 Procesmål 50](#_Toc343504686)

[1.6 Inddragelse af fagområder 50](#_Toc343504687)

[1.7 Samarbejdet 50](#_Toc343504688)

[1.8 Dokumenter 50](#_Toc343504689)

[1.9 Ressourcer 51](#_Toc343504690)

[1.10 Projektstyring 51](#_Toc343504691)

[1.11 Versionsstyring 51](#_Toc343504692)

[1.12 Den overordnede projektplan 51](#_Toc343504693)

[1.13 Plan for inceptionsfasen 51](#_Toc343504694)

[1.14 Use-case Diagram 53](#_Toc343504695)

[Appendix B 54](#_Toc343504696)

[Appendix C 54](#_Toc343504697)

# 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 Figur, Tabel eller kode og et tilhørende specifikt nummer.

#### 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. |
| **StockEmployee** | Ham der er ansat på *Stock* og har til opgave at ekspederer ordre. |
|  |  |

**Tabel 1.3.1**: Begrebstabel

#### CD-indhold

Den medfølgende cd indeholder følgende:

* Rapport.pdf
* …

#### Afprøvning af program

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

Bla bla (beskrivelse af hvordan programmet afprøves)

# 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 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.

## 1.4.2 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 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.

# Del II Procesmodel

# 2.1 Projektværktøjer

Rapporten skrives i Word og lægges på CD’en i pdf-format, mens diagrammer laves i Cacoo. Alle dokumenter gemmes og versionsstyres i GitHub.

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.

Java kildekoden versionsstyres gennem GitHub.

## 2.1.1 Versionsstyring

I opstartsfasen brugte vi dropbox til udveksling af dokumenter, mens der til versionsstyring er brugt GitHub. GitHub er et frit distribueret versionsstyringssystem.

## 2.1.2 Cacoo

Cacoo er et online værktøj der bruges til at tegne og modellere figurer og diagrammer i. Projektgruppen har valgt at bruge dette værktøj til udarbejdelse af UP artefakter, da vi allerede havde gode erfaringer med værktøjet.

# 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 elabaration.
* Elaboration – Denne fase kan gentages, og vil normalt blive det. I første elabaration vil man arbejde det de højst prioriterede use cases. Udfra 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 fungere 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 resultatets baseret værktøj, hvor at 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 opnået 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 forgår 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 symbolisere 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 og manglen af en kunde. Hvis vores 4 mands gruppe skulle afgive en mand til at spille projekt owner samt kunde, 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 som positionerne, men den ide droppede vi hurtigt igen. Det ville blive for kompliceret, og sænke projektforløbet mere end det gavnede. Den del af Scrum som vi valgte at benyttet 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 arbejder, blev antallet af møde 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 for 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 ordne 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 hel ordre, hvorefter alle varer på denne ordre vil blive leveret til at 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 et administrator, som vil kunne oprette nye varer og se hvilke aktive ordre system 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

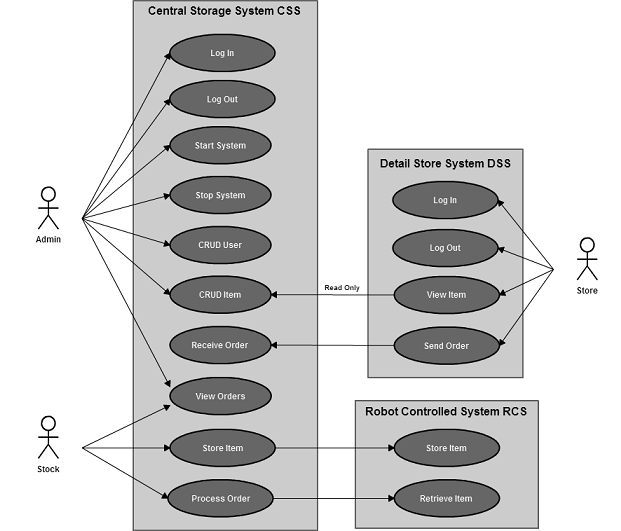
*Tabel 3.3.1* indeholder de aktører, der findes på CSS. En aktør kan enten være en person eller et andet system.

|  |  |  |
| --- | --- | --- |
| **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) |
| **Retrieve item** | 010, 011, 012 | M (P10) |

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

## 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. (1. iteration er de centrale use cases og de resterende lægges som billag)

|  |  |
| --- | --- |
| 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. |

|  |  |
| --- | --- |
| 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. |

|  |  |
| --- | --- |
| 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. |

|  |  |
| --- | --- |
| Retrieve Item | |
| Use case: | Retrieve Item |
| Actors: | CSS (Central Storage System) |
| Purpose: | Retrieve a specific Item from a specific position. |
| Description: | The RCS is given an Item no. and with it a specific position; the item is then retrieved and delivered to the user. Additionally the system updates the information in the database. |

## 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 1. iteration vist. De resterende *Expanded Use Cases* er vedlagt som bilag.

|  |  |  |  |
| --- | --- | --- | --- |
| 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**) | | |

|  |  |  |  |
| --- | --- | --- | --- |
| View Orders | | | |
| Use case: | | **View Orders** | |
| Actors: | | Admin & Stock | |
| 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 | | | |
|  |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **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. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 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. | | |

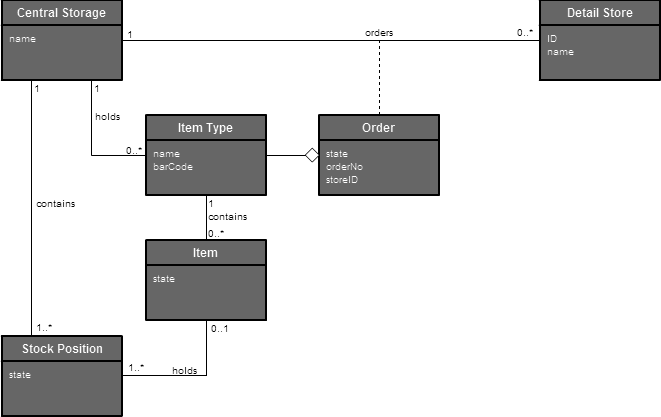
# 3.5 Klassediagram

*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.

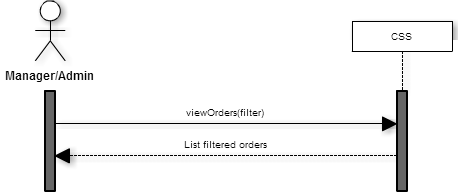


**Figur 3.5.1**: Klassediagram

# 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.

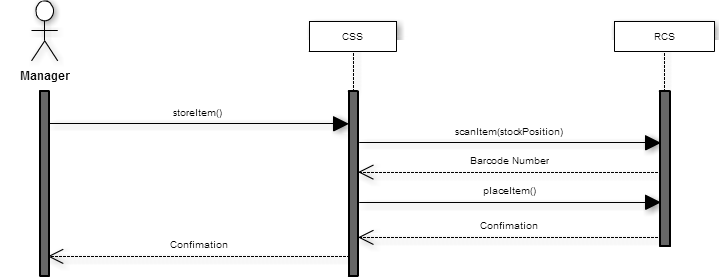
Manager eller Administrator vælger at se hvilke ordre der findes i systemet med en given status (filter) (åben eller lukket)



**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. |

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’. |

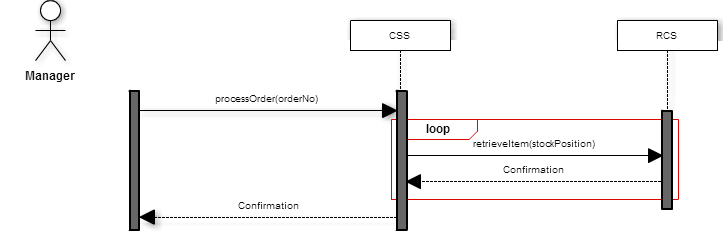
CSS modtager en bestilling fra DSS og CSS kvitterer for bestillingen.



**Figur 3.6.3:** 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. |

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 3.6.4:** 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’. |

# 3.8 Softwarearkitektur

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

## 3.8.1 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.

## 3.8.2 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:

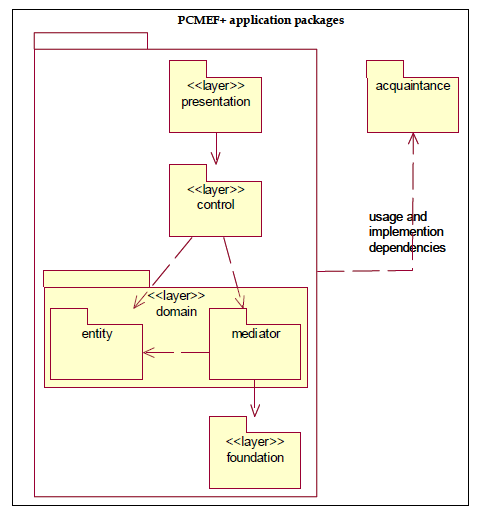
* Presentation Layer: presentation (GUI og interface til System aktører)
* Control layer: control (styring af domain og interface til presentation)
* Domain layer: Entity ( den del af systemet hvor instanser af businesmodel klasser opbevares og interagerer) samt Mediator (formidler kontakt mellem Entity 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 3.8.2.1**: Diagram over PCMEF+

#### PCMEF+ i CSS

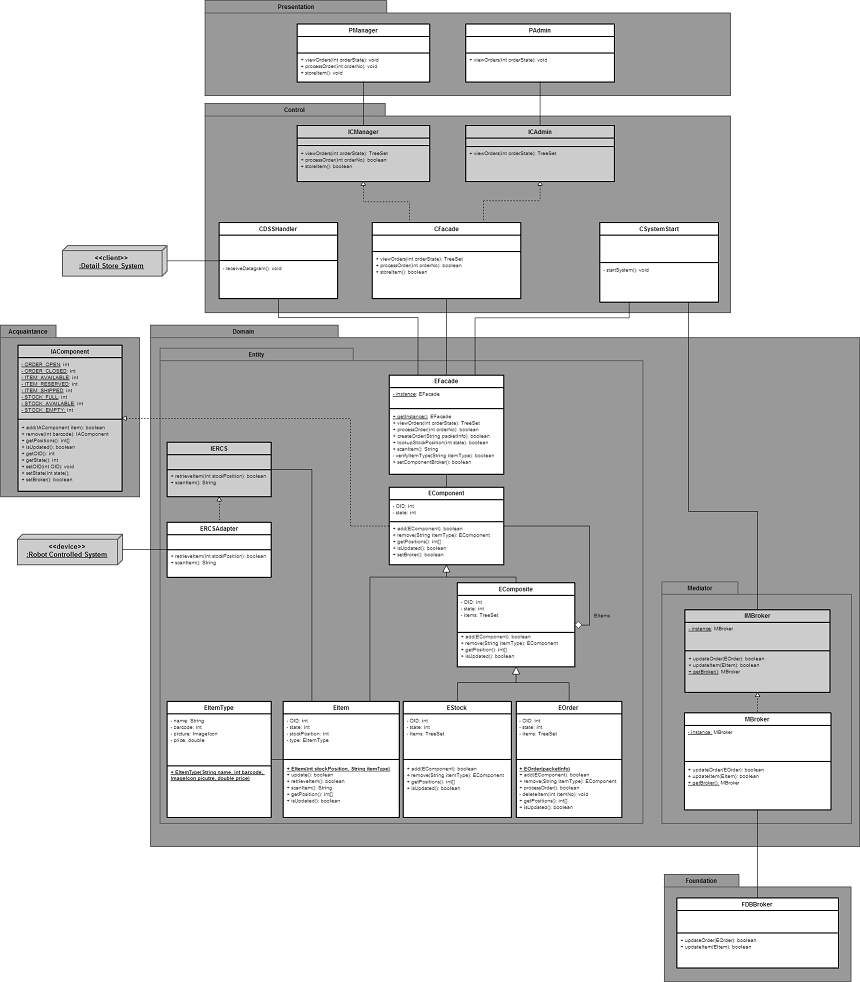
Selve CSS med de 4 Use Cases der blev udvalgt til 1. Iteration er ikke så omfattende et system at det berettiger til at bruge PCMEF+ men det er jo heller ikke pointen. Pointen er at modellere CSS på dette framework som om det var et stor og kompliceret system hvor hver enkelt del bliver udviklet og vedligeholdt 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 3.8.2.2**: CSS klassediagram

## 3.8.3 Anvendte designmønstre

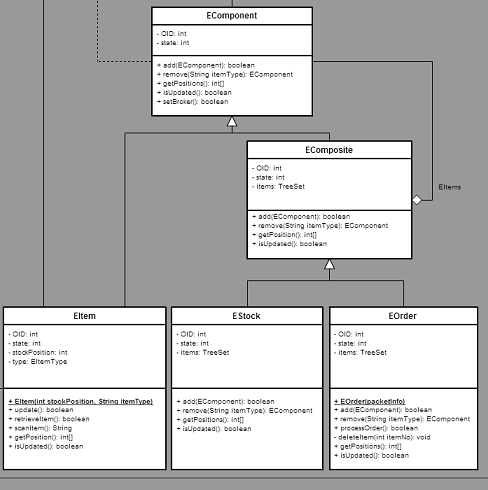
En væsentlig del af projektet er forståelse og anvendelse af designmønstre. Designmønstre er skabeloner til løsning af programopgaver. Igen for at sikre genanvendelig software. I det følgende afsnit beskrives de designmønstre der er anvendt i CSS.

#### 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 3.8.3.1**: 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 3.8.3.1**: 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.

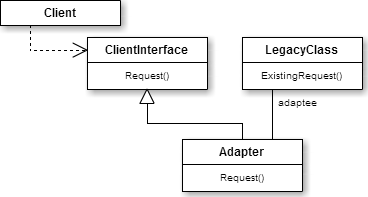


**Figur 3.8.3.2**: Adapter

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.

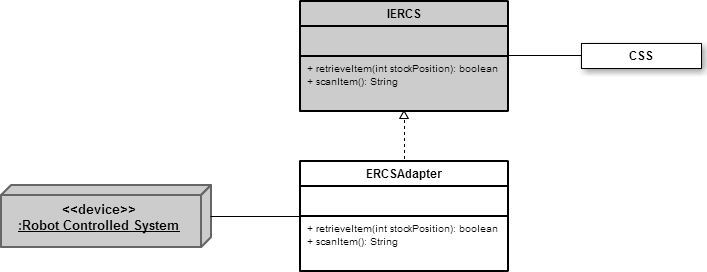
3. metode: Adapter: Ved at bruge en adapter kan vi skabe en meget direkte forbindelse mellem RCS og CSS, i denne pakke, som indeholder item klasse, som skal arbejde tæt sammen med RCS.



**Figur 3.8.3.3**: 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.

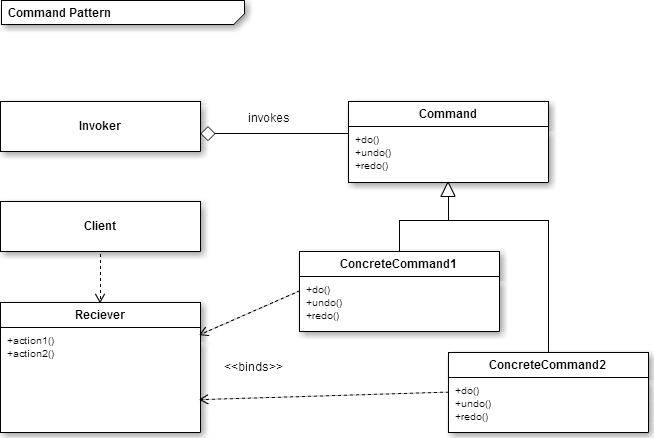


**Figur 3.8.3.4**: Adapter i CSS

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.



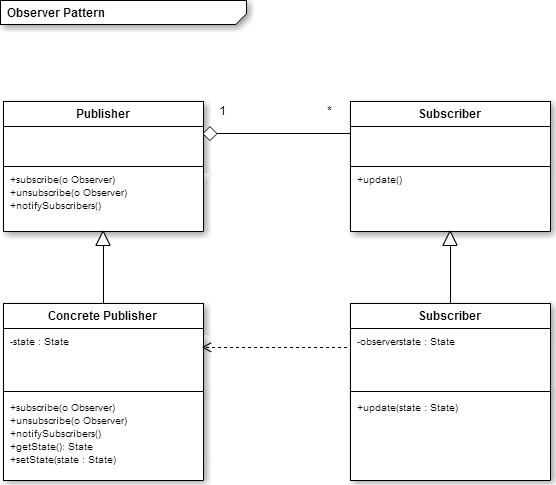
**Figur 3.8.3.5:** Command Pattern

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 3.8.3.6:** 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 funtionalitet 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.

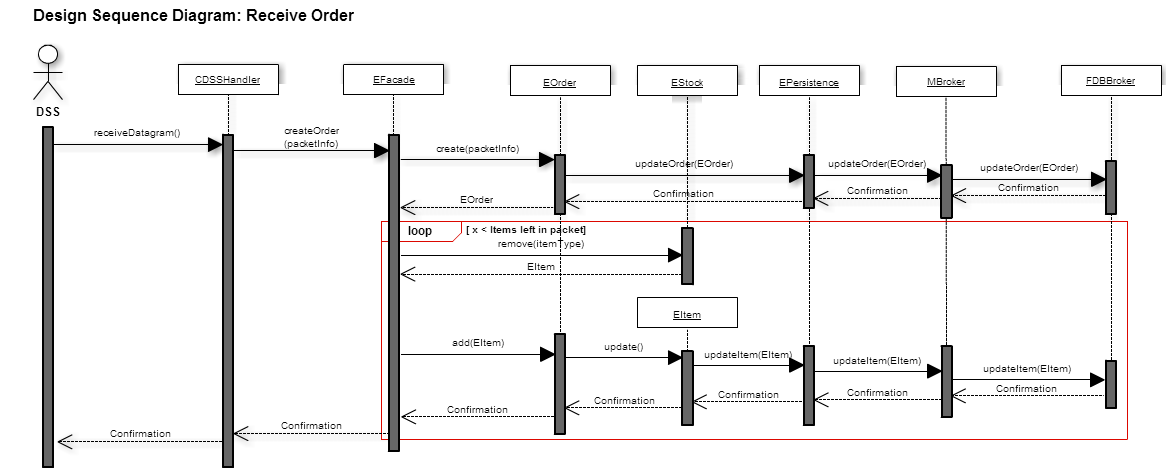
# 3.9 Detaljeret Design

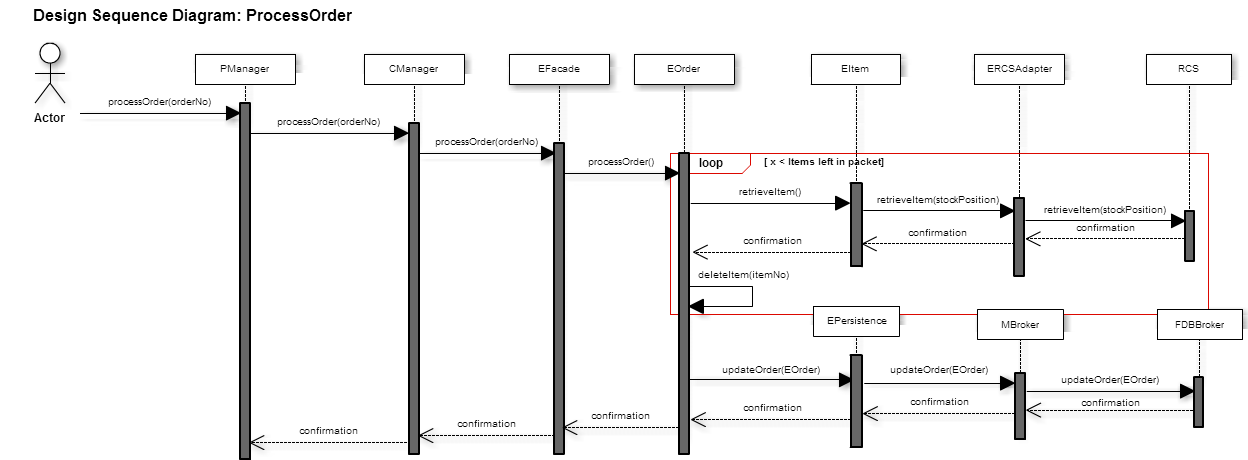
## 3.9.1 Design sekvensdiagrammer

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 *Process Order og Receive Order* Use casene og vedlægger de resterende som ***bilag i Appendix B.***.

Figur 3.9.1.1 Viser Sekvensdiagram for Receive Order og Figur 3.9.1.2 Viser Sekvensdiagram for Process Order.

**Figur 3.9.1.1** Receive Order. 

**Figur 3.9.1.2:** Process Order. 

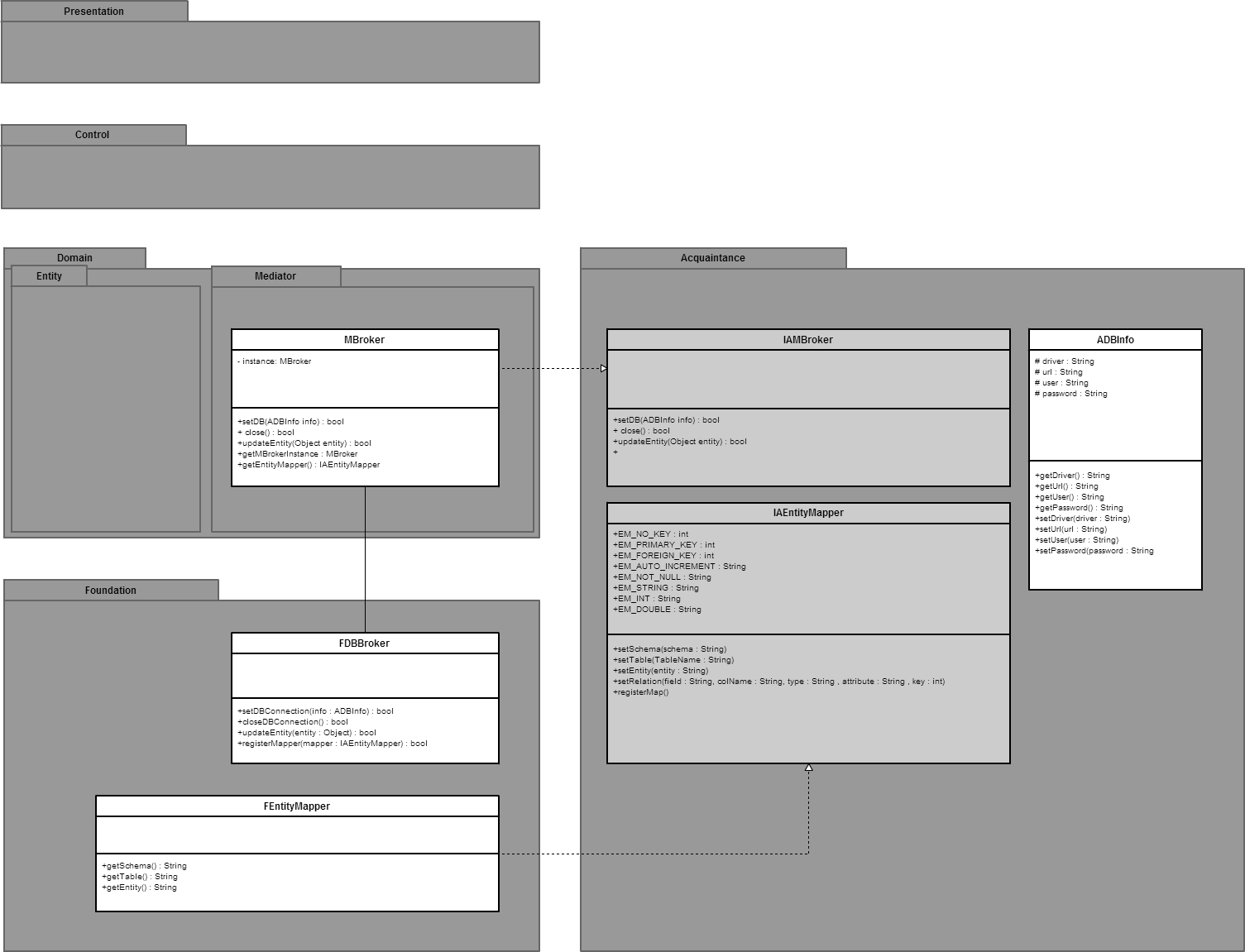
## 3.9.2 Persistens framework

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 3.9.2.1:** 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 opgave 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.

## 3.9.3 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.

*Tabel 3.9.3.1* vise Singleton klasser i CSS.

|  |  |
| --- | --- |
| **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 3.9.3.1**: Singleton klasser i CSS

## 3.9.4 Database design

FIGUR

# 3.10 Implementering

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

## 3.10.1 Pakker og klasser i Java

*Tabel 3.10.1.1* 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 3.10.1.1:** Pakker og java-klasser

## 3.10.2 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 3.10.2.1**: Singleton eksempel fra EFacade

## 3.10.3 Observer

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

PAdmin og PManager …

Figur der viser observers og subscribers i CSS.

## 3.10.1 Forbindelse mellem DSS og CSS

Som beskrevet tidligere skal der laves en forbindelse mellem klient og server, for dette kan lave sig gøre skal der bruges et port nummer, dette nummer har vi 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 skal er en string bliver splittet med parameteren ;, som de nødvendige parameter til at konstruere en ordre bliver brugt til dette.Efter ordren er skabt, interere metoden hen ovre den eller de itemtypes som beskeden indeholder. Det antal Items bliver så plukket fra stock, og sat på 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 3.10.1.1**: CreateOrder

# 3.11 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.

## 3.11.1 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 der 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 bestemme om kravene er opfyldt. Ydeevne test skal opfylde de ikke-funktionelle krav i forhold til fx Stres-, volume- og kvalitetstest. Funktions test de funktionelle krav. Testcases skrives ud fra Use Cases og krav til systemet.

#### Accepttest

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

## 3.11.2 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.

# 3.12 Datakommunikation

Til forbindelsen mellem DSS og CSS har vi valgt at bruge Transmission Control Protocol(TCP), som er en pålidelig protokol. Vi mente er 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 order 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 laver 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 at tal, pakke er en orderRequest er tallet 01. 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 at senere implementerer 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.

# 3.13 Styring regulering og overvågning

Til simulering af robotten (RCS) på lageret har vi valgt en PLC. Der kommunikeres 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.

## 3.13.1 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 3.13.1.1**: PLC – transportbånd med skanner

## 3.13.2 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.

## 3.13.3 Implementering på CSS siden

I *Kode 3.13.2* ses hvordan forbindelsen via serielporten åbnes og sættes op. Det er vigtig 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 3.13.3.1:** 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 3.13.3.2:** 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.

**public** **static** **class** Writer **implements** Runnable{

OutputStream out;

**public** Writer ( OutputStream out ){

**this**.out = out;

}

**public** **void** run ()

{

**try**{

message = message + "/ ";

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

test[test.length-1] = 13;

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

}

**catch** ( IOException e )

{

e.printStackTrace();

}

}

}

**Kode 3.13.3.3:** OutputStream implementeret i en tråd.

## 3.13.4 Implementering på RCS siden

#### Ladder

#### Struktureret text

## Problem med kommunikation med PLC

Stub

Hjælp fra Carl

Erfaringer med at det ikke virker…

# 3.14 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.



# Del IV Epilog

# 4.1 Konklusion

# 4.2 Perspektivering

# Bilag

# Appendix A

# 1. Projektgrundlag

## 1.1 Interessenter i projektet

* *Projektgruppen*: Der skal bedømmes ud fra projektforløbet og resultat af dette.
* *Steffen Peter Skov*: Der er gruppen 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

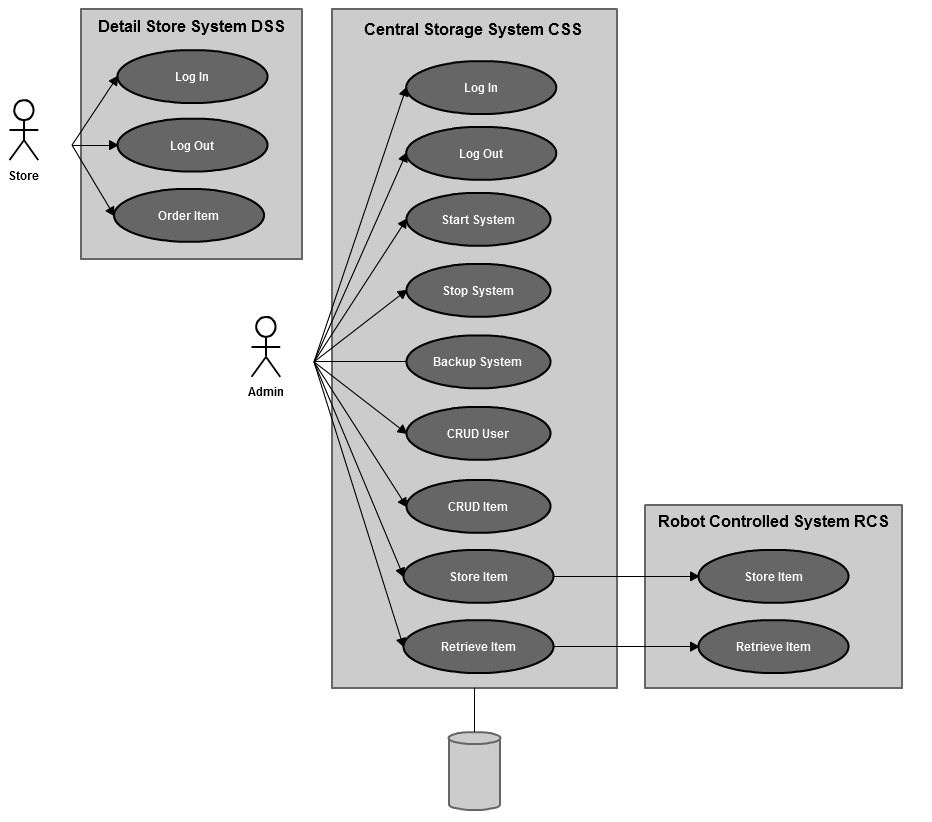
Den overordnede planlægning for projektets forløb fra start til slut.

|  |  |  |
| --- | --- | --- |
| **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

RESTERENDE DESIGN SEKVENSDIAGRAMMER

# Appendix C

Mødereferater