# Vision:

Vores vision for projektet var at lave et system, som kunne hjælpe med håndteringen af varer på et centralt lager system(Central Storage System). Vi ville gøre det muligt for kunder, som i vores tilfælde er detail butikker, at bestille varer i vores system. Detail butikker bruger deres eget system til at bestiller varer, dette system vil så indeholde en tynd klient til vores system. En detail butik vil kunne sende en ordre til vores system, som vi vil kunne behandle. Når en ordne blev behandlet, vil de bestilte varer blive reserveret på lageret, og være klar til at plukket. 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, som ankommer på en europalle, på et samle bånd. Derefter får systemet besked på at sætte pallen 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 laves på en mobil enhed, som manageren havde på din gaffeltruck. Herfra ville 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.

# Design Mål:

Ud fra vores ikke-funktionelle krav har vi udarbejdet vores design mål.

## Performance:

Vi har ikke nogen direkte ikke-funktionelle krav, som er til knyttet performance, men det betyder ikke at performance ikke er vigtigt. Systemet skal selvfølgelig ikke have for lange ventetider for brugeren, men der er plads til mindre vente tid. System skal kunne håndtere flere detail butikker som sender en order på samme tid. Vi vil ikke arbejde direkte for at få bedre performance, men vi vil gennem hele opbygningen af system have performance til baghovedet.

## Pålidelighed

Modtagelsen af orderRequest har vi betragtet som vigtig, med hensyn til pålidelighed. Derfor bruger vi TCP protokollen som er en pålidelig protokol, da den har et kvitterings system. Dette vil give os tryghed i at en orderRequest vil blive modtaget, og endnu vigtigere, at brugere vil få et svar tilbage. Selve CSS og RCS skal være pålideligt, fejl kan selvfølgelig ikke fjernes helt. Men ved fejl skal systemet kunne starte op igen, uden databasen har fået indtastet forkerte oplysninger.

## Vedligeholdelse

Igen er vedligeholdelse ikke dikteret ud fra vores ikke-funktionelle krav, vi har dog valgt at have fokus på det alligevel. Systemet skal være til at videreudvikle på, og måde vi regner med at gøre det muligt, er at arbejde med tidligere brugte design arkitektur og mønstre. Ved at gøre dette, vil andre lettere får et indblik i vores system. Det vil dermed være lettere at udskifte dele af systemet, men nyere delsystemer.

## Strategier

????

## Deployment

Har jeg misforstået noget?

Første forsøg:

Systemets deployering kan ske i flere faser, men en komplet deployering vil være kompleks. Detail store system vil klart være den letteste, da det kun kræver en opdatering af et måske allerede eksisterende detail store system. Udviklere til dette Detail store system skal kun skal kun kende formatet af en orderRequest vores system skal modtage. Dette sværeste vil være deployeringen af robot system, da det kræve et specielt bygget system, og tager tid at konstruere. Selve CSS systemet vil kunne deployeres let, og kan konfigureres sideløbende med robot system.

Andet forsøg:

System er delt op i flere systemer og locationer. Hovedsysteret er CSS, som varetager systemet væsentligste funktioner. Såsom database og lager håndtering. RCS er robotten som henter og lagre varer, den får alt dens information om hvor varer skal placeres og hentes fra CSS. DSS systemet kan tilgå få resurser i CSS, men virker som sit eget system.

Tredje forsøg:

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. Denne klientet 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.

Robottens opgave er at sætte varer på lager, og hente dem igen, samt at kunne scanne varer. Den har ikke selv adgang til nogle information, så den vil blive kontrolleret af serveren. Serveren med tilhørende database står for alt håndtering at informationer, det er denne som modtager et orderRequest fra DSS, og det der denne som giver RCS besked på at hente en varer. Udover dette vil den også holde styr på hvor alle varer står på lageret. Serveren vil kunne indgå i en domain struktur, da det vil kunne lette opgaven med adgangskontrol

## Persistens

En vigtigt i vores ikke-funktionelle krav er database håndteringen. Data i databasen skal beholdes længe og skal stemme overens med hvad der virkeligt er på lager. Det er vigtigt ved nedbrud, at der ikke bliver skrevet forkert information til databasen. Vi har valgt at benytte MySQL som database, dette valgt blev truffet fordi alle havde erfaring med denne database. Vi har valgt ikke at gå ind i backup, da SQL backup kan fås fra et tredjeparts program.

## Adgangskontrol

Det skal ikke muligt for uvedkommende at benytte vores system, derfor skal kommunikationen mellem detail store system på den måde fjerne uvedkommendes mulighed for dette. Om der bliver implemeret en form for kryptering eller lignende er ikke besked på dette tidspunkt. Sikkerheden ved Admin og Manager klienter vil højst sandsynligt blive håndteret at det styresystem som CSS kommer til at ligge på.

# UDP eller TCP opbygning:

Jeg ser opbygningen af vores orderRequest som en lang String. Som indeholder:

Detail store nummer – ”0123”

Timestamp for ordre – ”05-12-2012 21:39:11”

dato for forventet levering – ”06-12-2012”

Antal itemtypes – ”0123”

Itemtype1 nummer – ”0123”

Itemtype1 antal

Itemtype2 nummer

Itemtype2 antal

Itemtype…. nummer

Itemtype…. Antal

Eksempel: 0123;05-12-2012 21:39:11;06-12-2012;0002;0001;0001;0002;0001

Butik 123 sender en orderRequest den 05-12-2012 klokken 21:39:11 på 2 forskellige itemtypes. Deres ID er 0001 og 0002. butikken vil have en af hver og forventer levering den 06-15-2012

# 1. Elaberation

…..

## 2.3 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 XXXX.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | | | |
|  |  | | |

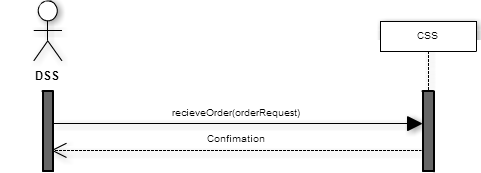
|  |  |  |  |
| --- | --- | --- | --- |
| **Store Item** | | | |
| **Use case:** | | **Store Item** | |
| **Actors:** | | Manager, RCS | |
| **Purpose:** | | Store a specific Item on its associated stock position. | |
| **Overview:** | | The item is scanned and its stock position is retrieved from CSS 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. | | |

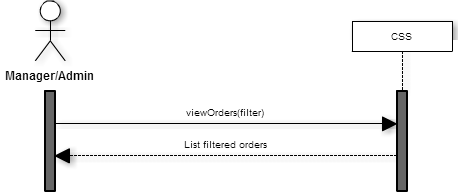
Retrieve Item blev ned prioteret i fordel af de 4 som er beskrevet. Grunden til dette er, at det er en use case som kører på RCS. Og ikke på CSS som de andre 4 gør.

## 2.4 System-sekvens Diagrammer

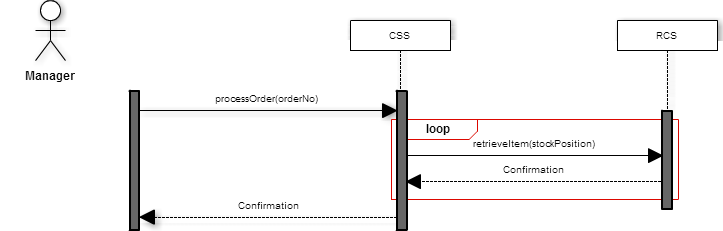
System sekvensdiagrammerne viser aktørens interaktion med systemet. Allerede her kan vi begynde at se, at de SCRUM point vi havde givet hver use case måske ikke ville være korrekte. Som tidligere beskrevet havde vi givet Receive Order 30 point. Denne beslutning blev taget fordi vi ikke havde et overblik, over hvad det krævede at modtage en besked fra et andet system.



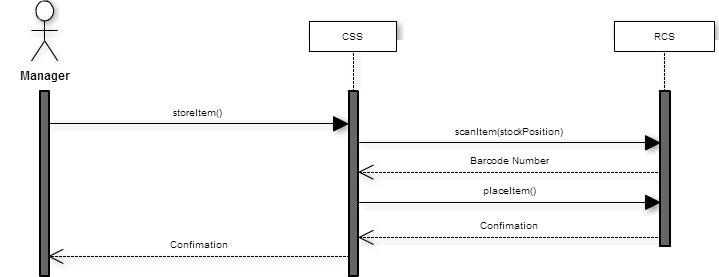
View Order har stadig 8 point, den fremgår stadig som en simpel use case.



Retrieve Item er blevet opdøbt til process order, så den nu behandler en del order i stedet for kun et item. På baggrund at disse diagrammer virker use case ikke til at være mere kompliceret ind først antaget. Udover at der skal loopes over flere items.

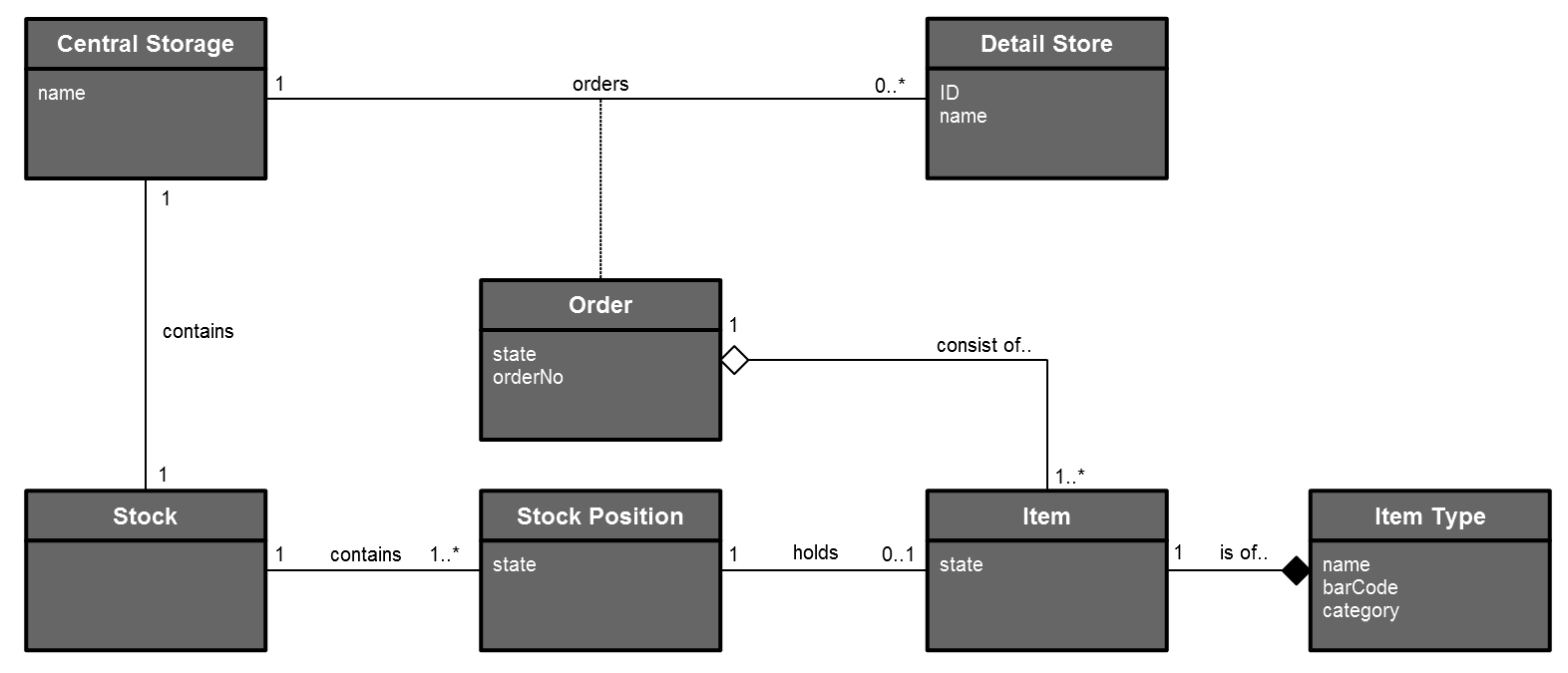


Store item er blevet en lille smule mere kompliceret end Process order, men den virker stadig relativt let.

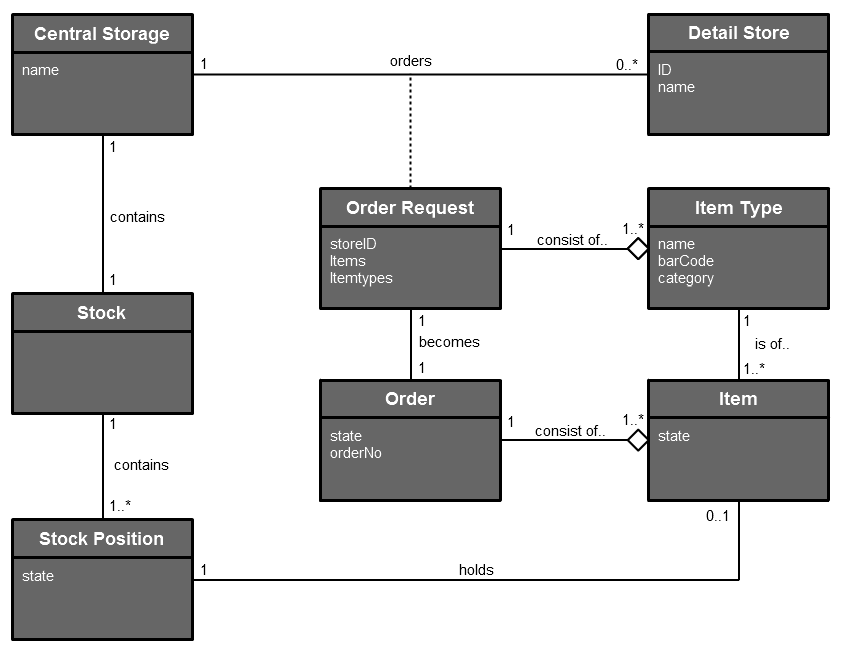


## Domain Model:

Ud fra de ting vi havde snakket om da vi lavede sekvens diagrammerne, har vi alle en forståelse for hvilke koncepter vores domain model skulle indeholde. Dette er den første:



Som hurtigt blev skrevet om til denne:



Grundet ikke dette var at en ordre ikke indeholdte et konkret item, men en item type. En Detail Store sender en orderRequest med en liste af item types. Når systemet så omskriver denne orderRequest til en konkret order, bliver konkrete item tilknyttet denne order.

## System Operationer kontrakter

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

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

|  |  |
| --- | --- |
| 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’. |

|  |  |
| --- | --- |
| 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’. |

# Interactions Diagrammer

Vi valgte at skrive alle vores Interactions diagrammer som Design sekvens diagrammer, fordi vi mener at det er lettere at læse.

