

## Goal dello Sprint 0

- individuare la struttura principale e le macro-entità del sistema e le loro interazioni.
- definire un piano di lavoro.

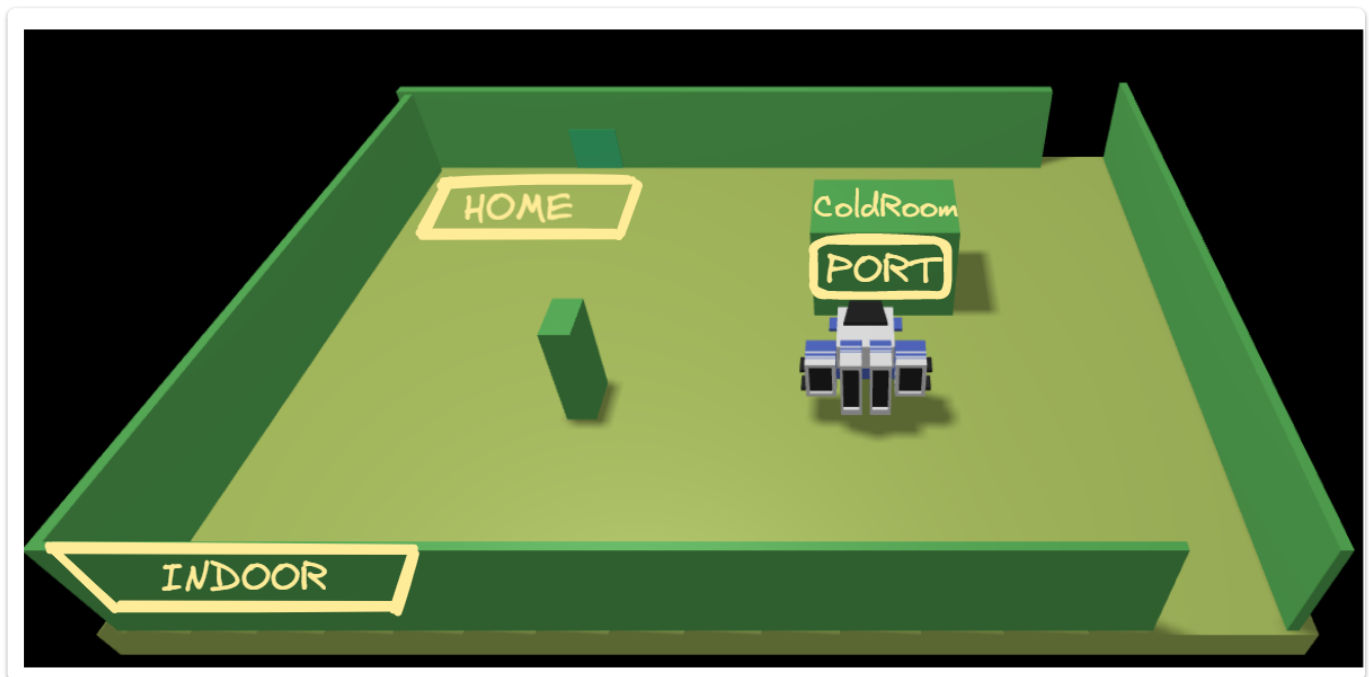
## Requisiti

A company intends to build a ColdStorageService, composed of a set of elements:

1. a service area (rectangular, flat) that includes:

- an INDOOR port, to enter food (fruits, vegetables, etc. )
- a ColdRoom container, devoted to store food, upto **MAXW** kg .

The ColdRoom is positioned within the service area, as shown in the following picture:



2. a DDR robot working as a transport trolley, that is initially situated in its HOME location. The transport trolley has the form of a square of side length **RD**.

The transport trolley is used to perform a deposit action that consists in the following phases:

1. pick up a food-load from a Fridge truck located on the INDOOR
2. go from the INDOOR to the PORT of the ColdRoom
3. deposit the food-load in the ColdRoom

3. a ServiceAccessGUI that allows an human being to see the current weight of the material stored in the ColdRoom and to send to the ColdStorageService a request to store new **FW** kg of food. If the request is accepted, the services return a ticket that expires after

a prefixed amount of time (**TICKETTIME** secs) and provides a field to enter the ticket number when a Fridge truck is at the INDOOR of the service.

4. a ServiceStatusGUI that allows a Service-manager (an human being) to supervises the **state** of the **service**.

## Alarm requirements

The system includes a Sonar and a Led connected to a RaspberryPi.

The Sonar is used as an 'alarm device': when it measures a distance less that a prefixed value **DLIMIT**, the transport trolley must be stopped; it will be resumed when Sonar detects again a distance higher than **DLIMIT**.

The Led is used as a *warning devices*, according to the following scheme:

- the Led is **off** when the transport trolley is at HOME
- the Led **blinks** while the transport trolley is moving
- the Led is **on** when transport trolley is stopped

## Service users story

The story of the ColdStorageService can be summarized as follows:

1. A Fridge truck driver uses the *ServiceAccessGUI* to send a request to store its load of **FW** kg. If the request is accepted, the driver drives its truck to the INDOOR of the service, before the ticket expiration time **TICKETTIME**.
2. When the truck is at the INDOOR of the service, the driver uses the *ServiceAccessGUI* to enter the ticket number and waits until the message **charge taken** (sent by the ColdStorageService) appears on the *ServiceAccessGUI*. At this point, the truck should leave the INDOOR.
3. When the service accepts a ticket, the transport trolley reaches the INDOOR, picks up the food, sends the **charge taken** message and then goes to the ColdRoom to store the food.
4. When the deposit action is terminated, the transport trolley accepts another ticket (if any) or returns to HOME.
5. While the transport trolley is moving, the Alarm requirements should be satisfied. However, the transport trolley should not be stopped if some prefixed amount of time (**MINT** msec) is not passed from the previous stop.
6. A *Service-manager* might use the ServiceStatusGUI to see:
  - the **current state** of the transport trolley and its **position** in the room;
  - the **current weight** of the material stored in the ColdRoom;
  - the **number of store-requests rejected** since the start of the service.

## Analisi del TF23

Nelle discussioni con il committente, sono emerse alcune problematiche:

- Il problema del load-time lungo.
- Il problema del driver distratto (non coerente, rispetto alle due fasi: scarico preceduto da prenotazione).
- Il problema del driver malevolo.
- Il problema di garantire che una risposta venga sempre inviata sempre solo a chi ha fatto la richiesta, anche quando la richiesta è inviata da un 'alieno' come una pagine HTML

## Il problema del load-time lungo

Il problema del load-time lungo è stato affrontato da Arnaudo/Munari con l'idea di inviare due messaggi di 'risposta' (una per dire al driver che il ticket inviato è valido e una per inviare `chargeTaken`). A questo fine hanno fatto uso diretto della connessione TCP stabilita da una versione prototipale dell'`accessGui` fatta come GUI JAVA.

Per consentire questa possibilità anche a livello di modellazione qak, in *ActorBasicFsm* è stato introdotto il metodo `storeCurrentRequest()` che permette di ricordare la richiesta corrente (cancellata da una `replyTo`). Questo però è un trucco/meccanismo che potrebbe risultare pericoloso.

Meglio affrontare il problema dal punto di vista logico, impostando una interazione a DUE-FASI tra driver e service (compito che può svolgere la *serviceAccessGui*).

- FASE1: il driver invia il ticket e attenda una risposta (immediata) come ad esempio `ticketaccepted/ticketrejected`
- FASE2: il driver invia la richiesta `loaddone` e attenda la risposta ( `chargeTaken` o fallimento per cause legate al servizio)

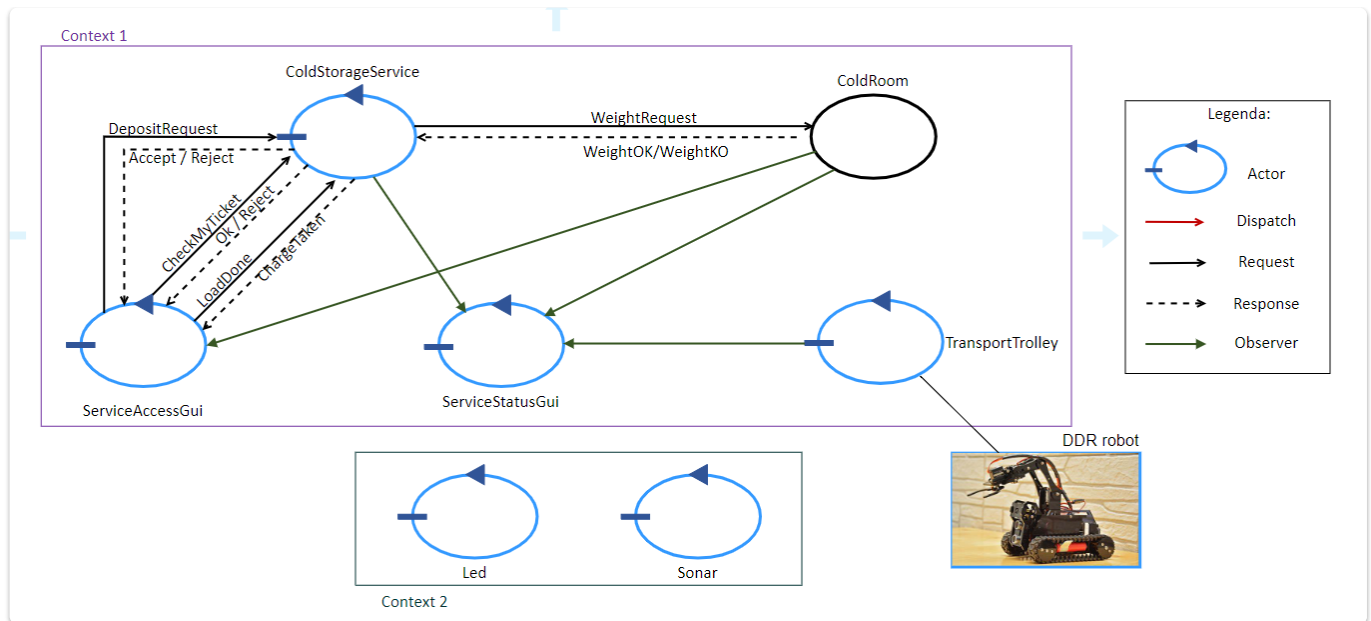
## Il problema del driver distratto

Questo problema ha indotto il committente ad affermare che:

quando un agente esterno (driver) invia il ticket per indurre il servizio a scaricare il truck, si SUPPONE GARANTITO che il carico del truck sia UGUALE al carico indicato nella prenotazione.

Ciò in quanto non vi sono sensori (balance , etc) che possano fornire il valore del carico effettivo sul Truck.

## Analisi preliminare dei requisiti



## Service Area

```
val ServiceArea = object {
  int LatoLungo
  int LatoCorto
}
```

## HOME

```
val Home = object {
  int x = 0
  int y = 0
}
```

## INDOOR port

```
val IndoorPort = object {
  int x = 0
  int y = MAX_Y
}
```

## Porta della ColdRoom

Lato sud del ColdRoom Container. Transport Trolley potrà interagire con ColdRoom attraverso questa.

## ColdRoom Container

Contenitore in posizione fissa in Service Area, il cui punto di accesso è la [Porta della ColdRoom](#), in grado di ricevere e contenere cibo da un lato specifico. Ha una capienza pari a MAXW kg.

```
var ColdRoom = object {  
    int MAXW  
    int CurrentWeight = 0  
    int x  
    int y  
}
```

## DDR robot

*Differential Drive Robot*, vedi [robot](#).

## Transport trolley

Transport trolley è un DDR robot capace di spostarsi all'interno di [Service Area](#). I comandi che è in grado di compiere sono descritti nell'apposita [documentazione](#).

```
int RD          #lunghezza del lato del quadrato
```

## Food-load

Carico (in kg) che il robot preleverà da Indoor e depositerà in ColdRoom Container.

```
int FoodLoad
```

## TicketTime

```
long TicketTime  #tempo espresso in secondi
```

## Ticket

```
int TicketNumber
```

## ServiceAccesGUI

GUI che permette ai driver di:

- visualizzare la quantità di cibo (in peso) contenuta all'interno di ColdRoom.
- richiedere la generazione di un Ticket da presentare in un secondo momento.
- presentare il Ticket assegnatogli in precedenza nel momento in cui il driver arriva in INDOOR port.
- inviare la richiesta "LoadDone" quando il driver è pronto a scaricare.

```
QActor serviceaccessgui context ctxcoldstoragearea {
    [#
        var Ticket = " "
        var Ticketok = false
        var PESO = 0
    #]

    State s0 initial {
        printCurrentMessage
        println("SAG - in attesa") color yellow
    } Goto work

    State work {
        //random tra 10 e 20
        [# PESO = Math.floor(Math.random() *(20 - 10 + 1) + 10).toInt()
        #]
        println("SAG - chiedo $PESO") color yellow
        request coldstorageservice -m depositRequest :
depositRequest($PESO)

        } Transition t0 whenReply accept -> gotoindoor
                                whenReply reject -> tryagainlater

    State tryagainlater{
        println("SAG - rifiutato") color yellow
    }Transition wait whenTime 5000 -> work

    State gotoindoor{
        onMsg( accept : accept(TICKET)){
            [#      Ticket = payloadArg(0)
            #]
            println("SAG - accettato, Ticket: $Ticket") color yellow
        }
    }Transition t2 whenTime 3000 -> giveticket
}
```

```

    State giveticket{
        println("SAG - consegna il biglietto") color yellow

        request coldstorageservice -m checkmyticket :
checkmyticket($Ticket)
    }Transition tc whenReply ticketchecked -> checkresponse

    State checkresponse {
        onMsg (ticketchecked : ticketchecked(BOOL)){
            [# Ticketok = payloadArg(0).toBoolean()
                # ]

        }
        println("SAG - biglietto accettato? : $Ticketok") color yellow
    } Goto work if [# !Ticketok #] else unloading

    State unloading{
        println("SAG - scarico") color yellow
    }Transition t4 whenTime 3000 -> loaddone

    State loaddone {
        request coldstorageservice -m loaddone : loaddone($PESO)
    } Transition t6 whenReply chargetaken -> work
}

```

## ColdStorageService

ColdStorageService si occupa di gestire le richieste di scarico merce, questo comprende:

- ricevere le richieste di permesso di scarico.
- generare Ticket assegnati al singolo driver che ne ha fatto richiesta.
- ricevere e verificare i Ticket nel momento in cui il driver arriva in INDOOR.

```

QActor coldstorageservice context ctxcoldstoragearea {

    [#

        var TICKETTIME = 10000;

        var Token = "_"
        var InitialToken = "T"
    ]
}

```

```

        var Ticket = ""
        var Sequenza = 0
    #]

    State s0 initial{
        println("coldstorageservice - tickettime: $TICKETTIME") color blue
        printCurrentMessage
    } Goto work

    State work {
    }Transition t0  whenRequest depositRequest -> checkforweight
                                whenRequest checkmyticket -> checktheticket
                                whenRequest loaddone -> loadchargetaken

    State checkforweight {
        onMsg(depositRequest : depositRequest(PESO)){
            [# var Peso = payloadArg(0).toInt() #]
            println("coldstorageservice - richiedo $Peso") color blue
            request coldroom -m weightrequest : weightrequest($Peso)
        }
    }Transition t1 whenReply weightKO -> reject
                                whenReply weightOK -> returnticket

    State reject {
        println("coldstorageservice - non c'è comunque posto, vai a casa")
color blue
        replyTo depositRequest with reject : reject( reject )
    } Goto work

    State returnticket {

        [# Ticket = "T".plus(Token)
            var Now = java.util.Date().getTime()/1000

            Ticket = Ticket.plus( Now ).plus(Token).plus( Sequenza)
            Sequenza++
        #]
        println("coldstorageservice - accettato") color blue
        replyTo depositRequest with accept : accept( $Ticket )
    } Goto work

```



```

State checktheticket {
    onMsg(checkmyticket : checkmyticket(TICKET)){
        [#      var Ticket = payloadArg(0)
            var Ticketvalid = false;

            var StartTime = Ticket.split(Token, ignoreCase=true,
limit=0).get(1).toInt()
            var Now = java.util.Date().getTime()/1000
            if( Now < StartTime + TICKETTIME){
                Ticketvalid = true
            }

            #]
            println("coldstorageservice - biglietto valido?
$Ticketvalid") color blue
            replyTo checkmyticket with ticketchecked :
ticketchecked($Ticketvalid)
        }
    } Goto work

State loadchargetaken {
    onMsg(loaddone : loaddone(PESO) ){
        [# var Peso = payloadArg(0).toInt()#]
        println("coldstorageservice - chargetaken peso dichiarato: $Peso")
color blue
    }
    replyTo loaddone with chargetaken : chargetaken( NO_PARAM )

}Goto work

}

```

## ServiceStatusGUI

Componente che permette al Service-manager (persona fisica) di supervisionare lo [Stato del servizio](#)

## Stato del Servizio

Lo stato del servizio comprende:

- Lo stato e la posizione del TransportTrolley.
- Lo stato della ColdRoom (peso corrente su totale).
- Il numero di richieste negate dall'inizio del servizio.

## Segnali

```
Request depositRequest : depositRequest(PESO)
```

```
Reply accept : accept(TICKET)
```

```
Reply reject : reject(NO_PARAM)
```

```
Request weightrequest : weightrequest(PESO)
```

```
Reply weightOK : weightOK( NO_PARAM )
```

```
Reply weightKO : weightKO( NO_PARAM )
```

```
Request checkmyticket : checkmyticket(TICKET)
```


```
Reply ticketchecked : ticketchecked(BOOL)
```

```
Request loaddone : loaddone(PESO)
```

```
Reply chargetaken : chargetaken(NO_PARAM)
```

```
Dispatch startToDoThings : startToDoThings( NO_PARAM )
```

Name	Sender	Receiver	Type	Motivazioni
DepositRequest	ServiceAccesGUI	ColdStorageService	Req/Resp	Deve attendere la risposta: Accepted/Rejected
LoadDone	ServiceAccesGUI	ColdStorageService	Req/Resp	Deve attendere la risposta: ChargeTaken
WeightRequest	ColdStorageService	ColdRoom	Req/Resp	Risposta necessaria per proseguire: WeightOK/KO
CheckMyTicket	ServiceAccesGUI	ColdStorageService	Req/Resp	Deve attendere la risposta: Ok/Rejected

 [Tipi di segnali >](#)

in generale le ragioni per i vari tipi di messaggio sono:

- req/resp se ho bisogno di ricevere una risposta
- dispatch se è un messaggio per un componente specifico che conosco e non mi interessa la risposta
- event se è per uno o più componenti che non conosco direttamente (io emetto e chi è interessato riceve)

## Contesti

```
Context ctxcoldstoragearea ip [host="localhost" port=8040]
Context ctxLedSonar ip [host="127.0.0.1" port=8088]
```

## Keypoints

### 1) Aggiornamento di ServiceStatusGUI

SSG dovrà presentare i dati aggiornati del sistema ad ogni istante, dovrà quindi comportarsi come un **Observer**, sfruttiamo la tecnologia degli [StreamQActor](#).

### 2) Carico di lavoro di ColdStorageService

Tutto il lavoro del sistema al momento passa attraverso ColdStorageService, dalla gestione dei Ticket all'interazione con il TransportTrolley --> **Da valutare una divisione in più componenti**

### 3) ColdRoom, Attore o POJO?

Per requisiti il sistema deve essere distribuito, tutte le entità definite finora saranno quindi modellate come **Attori**, in particolare **ColdRoom** decidiamo di modellarla come attore e non come POJO per i seguenti motivi:

- Nonostante non sia nei requisiti è logico pensare che in futuro il sistema debba essere esteso con funzionalità per diminuire il peso contenuto in ColdRoom. Definire il componente come attore faciliterà questa aggiunta.
- Inoltre definire ColdRoom come attore esterno è in linea con il principio di singola responsabilità e alleggerisce il carico di lavoro di ColdStorageService.

```
QActor coldroom context ctxcoldstoragearea {
    //corrente: quanta roba c'è nella cold room
    //previsto: quanto deve ancora arrivare, ma per cui c'è un biglietto emesso
    [#
        var Peso = 0
        var MAXW = 50
    ]
```

```

#]

State s0 initial {
    printCurrentMessage
} Goto work

State work{

}Transition update whenRequest weightrequest -> checkweight

State checkweight {
    onMsg(weightrequest : weightrequest(PESO)){
        [# var PesoRichiesto = payloadArg(0).toInt()#]

        if [# Peso + PesoRichiesto <= MAXW #] {
            [# Peso += PesoRichiesto#]
            println("coldroom - accettato peso: $PesoRichiesto.
Peso totale in coldroom: $Peso") color green
            replyTo weightrequest with weightOK : weightOK(
NO_PARAM)
        } else {
            println("coldroom - rifiutato") color green
            replyTo weightrequest with weightKO : weightKO(
NO_PARAM )
        }
    }
} Goto work
}

```

## Posizione del robot?

Sarà necessario per il sistema riuscire ad identificare la posizione corrente del robot in ogni istante per pianificare il percorso da intraprendere.

Per risolvere il problema assoceremo alla **Service Area** un sistema di coordinate da definire in seguito.

## Discussioni col committente

*Richiesta al committente:*

*Dimensione della Service Area: 7m \* 5m.*

*Richiesta al committente:*

- *Capienza massima (MAXW) corrisponde a 50 kg*
- *La grandezza di ColdRoom Container è 1m \* 1m*
- *Posizione in Service Area come da figura iniziale*
- *Sarà possibile per il robot muoversi attorno alla ColdRoom*

*Richiesta al committente:*

- *dimensione del transport trolley corrisponde ad un quadrato di lunghezza  $RD = 1\text{ m}$ .*

## Divisione in Sprint

### 1. Transport Trolley + ColdStorageService [Sprint 1.0](#)

#### Descrizione >

Lo scopo del primo sprint è produrre una prima versione funzionante del core dell'applicazione. Questo comprende ColdStorageService con la logica di gestione dei Ticket e il TransportTrolley funzionante.

A questa parte deve essere affiancata una mock version della ServiceAccessGUI per la fase di testing.

### 2. Led e Sonar [Sprint 2](#)

#### Descrizione >

Nel secondo sprint verranno implementati il sistema di led e sonar con la logica ad essi associata.

### 3. ServiceStatusGui e grafiche migliorate [Sprint 3](#)

#### Descrizione >

Nel terzo sprint ci occuperemo della ServiceStatusGUI e delle interfacce grafiche finali.

## Divisione dei compiti

Ogni Sprint verrà affrontato insieme con divisione dei compiti specifica valutata di volta in volta.

## Piano di Lavoro

Sprint	GOAL	Tempo Stimato	Divisione del Lavoro	Note
SPRINT 1	Sviluppo del primo prototipo	2 man-days	3 persone	
	Sviluppo della ServiceAccessGUI	1 man-day	2 persone	
	Testing	2 man-hours	1 persona	
SPRINT 2	Sviluppo di Led e Sonar	4 man-hours	3 persone	
	Testing di Led e sonar	1 man-hour	1 persona	
	Implementazione con il resto del sistema	1 man-hour	1 persona	
	Testing completo	1 man-hour	1 persona	
SPRINT 3	Sviluppo della ServiceStatusGui	1 man-day	2 persone	
	Testing del sistema completo	2 man-hour	1 persona	
	Refactoring della user interface	1 man-hour	1 persona	
	Testing finale dell'intera applicazione	3 man-hour	3 persone	Il testing finale deve essere condiviso da tutti i membri del gruppo