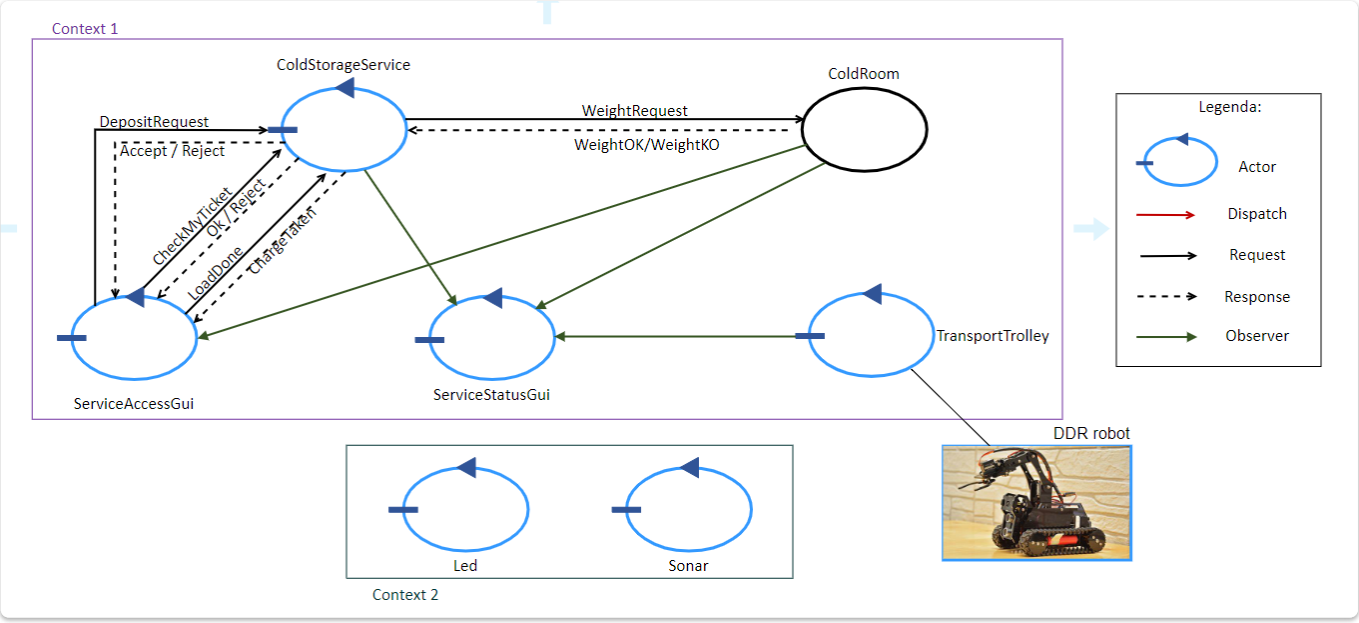
- [ ] Alcuni link fanno riferimento a doc solo sul mio pc, da cambiare...

Prodotto dello Sprint 0

È stata individuata un'architettura logica iniziale che definisca le macro-entità del sistema e le loro interazioni, link al modello precedente.

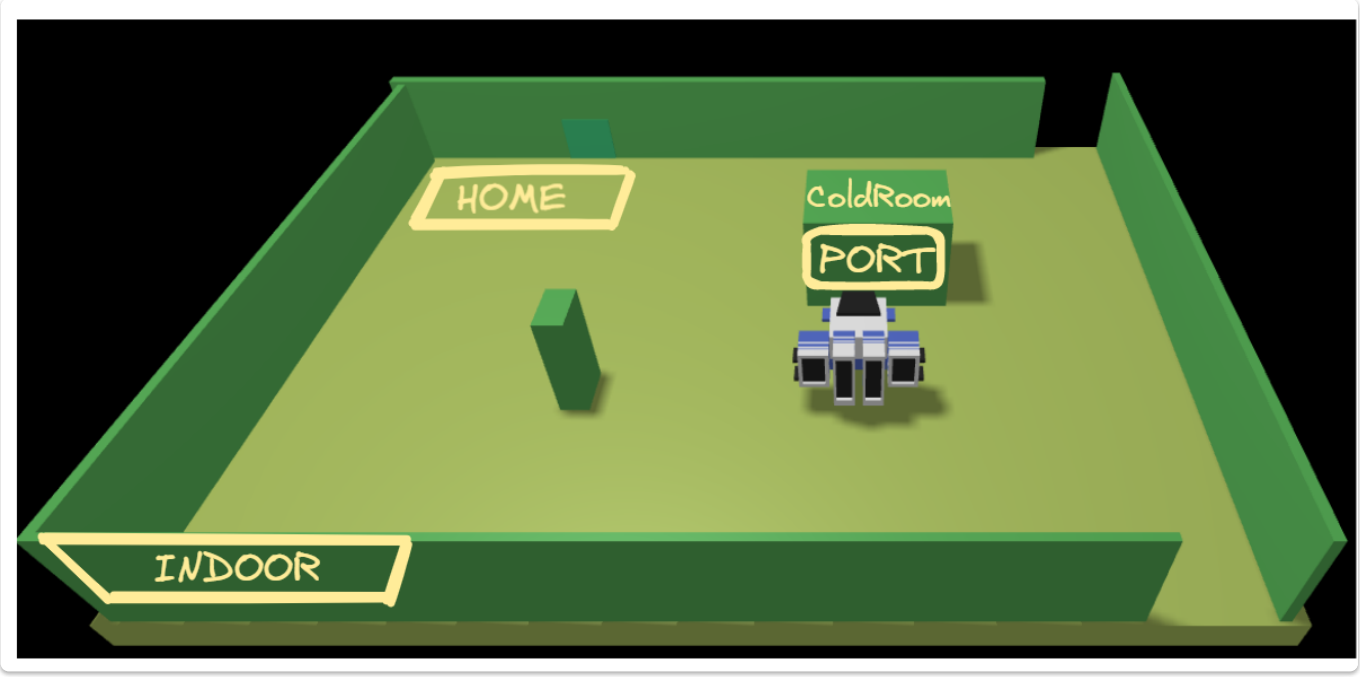


Goal Sprint 1

1. Transport Trolley + ColdStorageService

> **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.

Requisiti relativi allo sprint corrente



Requisiti

Analisi dei Requisiti

analisi requisiti sprint 0

> **domanda** >
>
> Possiamo limitarci a mettere il riferimento allo sprint 0 o dobbiamo riportare tutto?

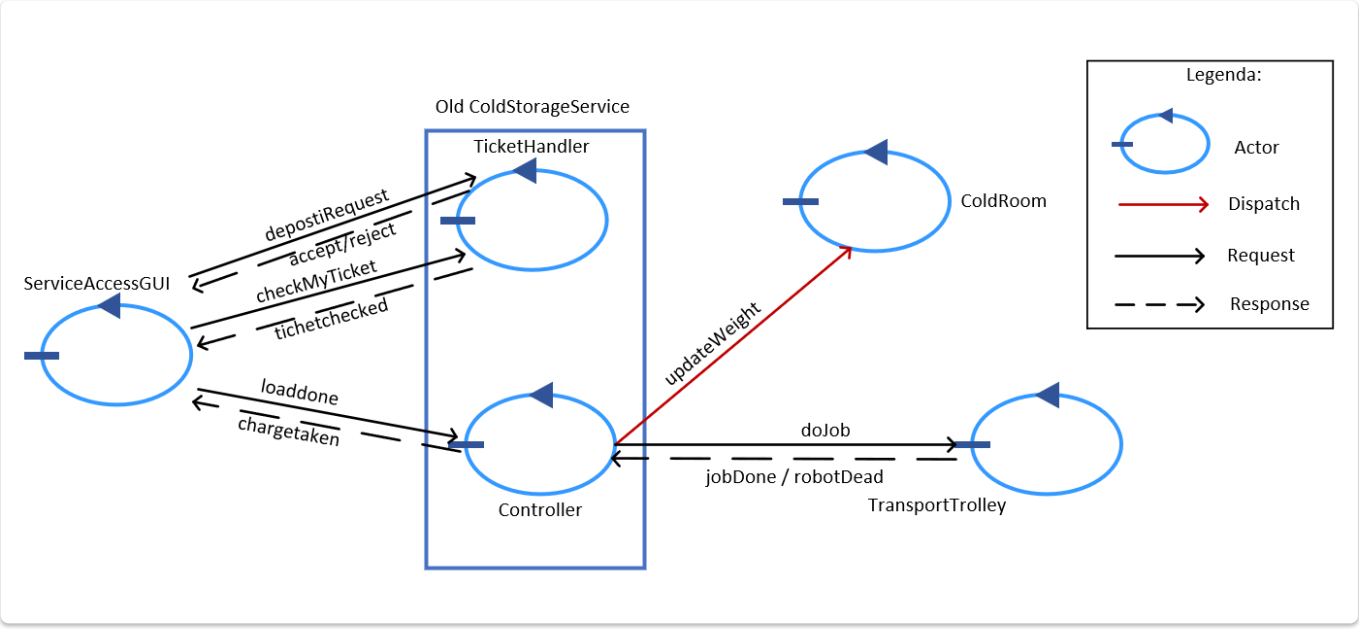
Analisi del Problema

Responsabilità di ColdStorageService

ColdStorageService è un componente caratterizzato da troppe responsabilità, abbiamo quindi deciso di sostituirlo con 2 attori:
- Controller: si occupa di gestire il robot ed aggiornare il peso di ColdRoom.
- TicketHandler: si occupa di gestire il ciclo di vita dei Ticket.

Nello sprint corrente ci occuperemo solo del Controller. La logica di gestione dei ticket è rimandata allo sprint successivo (Sprint 1.1 - V3)

Cerchiamo quindi di realizzare la seguente Architettura logica:



Segnale per Transport Trolley

Introduciamo un nuovo segnale "doJob" di tipo Req/Res inviato dal controller.

```
Request doJob : doJob(KG)
Reply jobdone : jobdone(NO_PARAM)
Reply robotDead : robotDead(NO_PARAM)
```

> **motivazioni** >
>
> Definiamo il seganle come un req/res poichè vogliamo sapere se il servizio richiesto è andato a buon fine oppure se il DDR robot ha avuto problematiche che lo hanno interrotto prima di proseguire con una seconda doJob.
>
> Limitiamo il controller ad un semplice comando di doJob, non è compito suo sapere quali operazioni deve compiere il robot per portare a termine il lavoro, è compito del robot stesso.

ATTENZIONE: la risposta deve essere inviata appena il carico è rilasciato nella ColdRoom e non quando il robot torna alla home per requisiti.

Aggiornamento peso ColdRoom

Se il servizio è andato a buon fine e viene restituita una "jobdone" allora il Controller aggiorna il peso della ColdRoom tramite Dispatch.

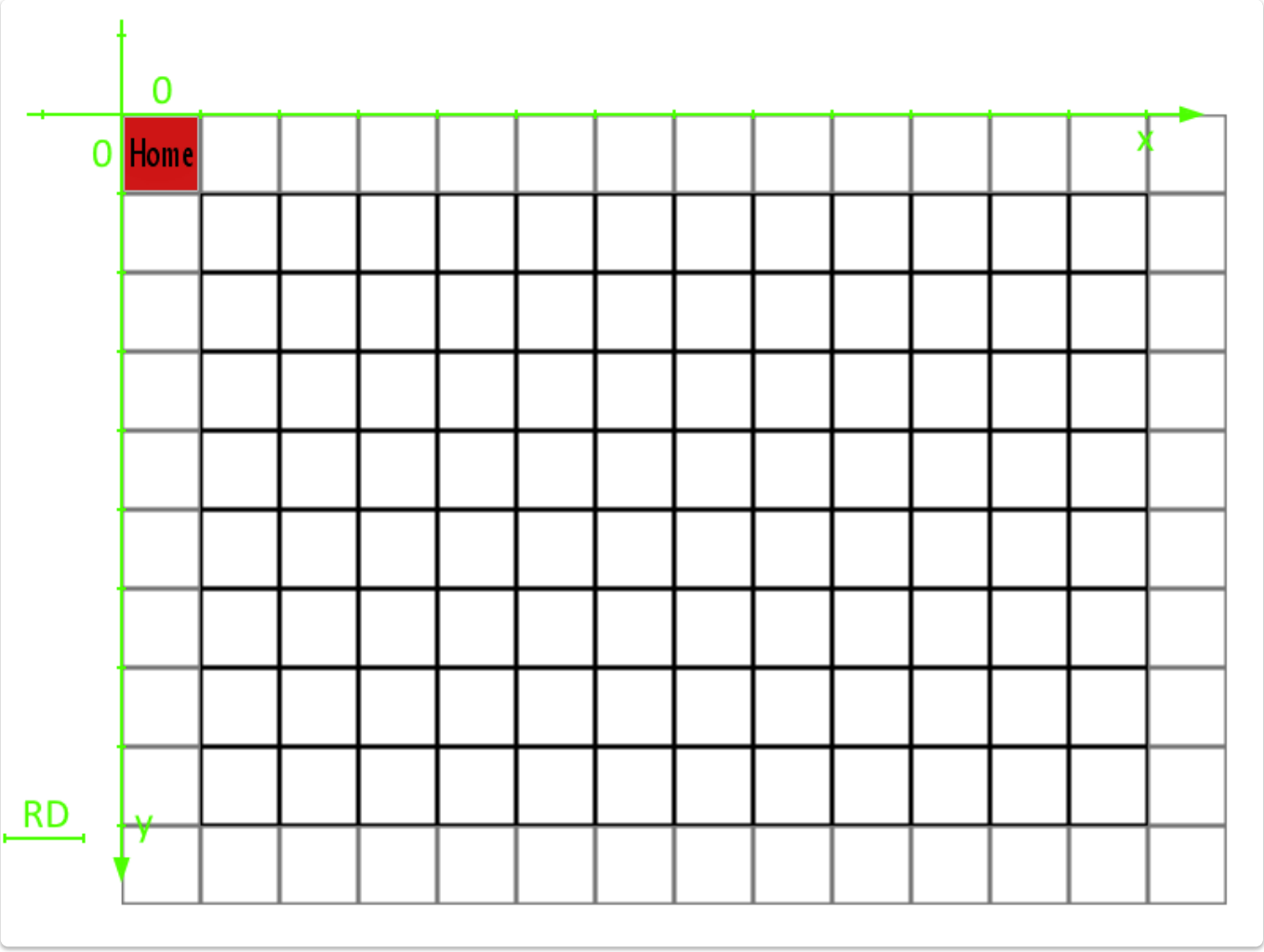
```
Dispatch updateWeight : updateWeight(PESO)
```

Da "doJob" a comandi per TransportTrolley

Dalla documentazione fornita è chiaro che non sia presente un comando che ci permetterebbe di limitarci ad un comando "doJob".
Se non fosse possibile implementarlo risulterebbe necessario aggiungere un componente intermedio che traduca in comandi comprensibili al TransportTrolley fornitoci.
Allo stesso modo è anche evidente la mancanza di un comando per caricare e scaricare i materiali trattati e quindi non risulta sufficiente.

Posizione nella Service Area

Per definire la posizione del TransportTrolley e permettere il movimento autonomo dividiamo la stanza in una griglia di quadrati di lato RD (lunghezza del DDR robot).
La Home corrisponderà all'origine (0, 0). Useremo coordinate crescenti verso il basso e verso destra.



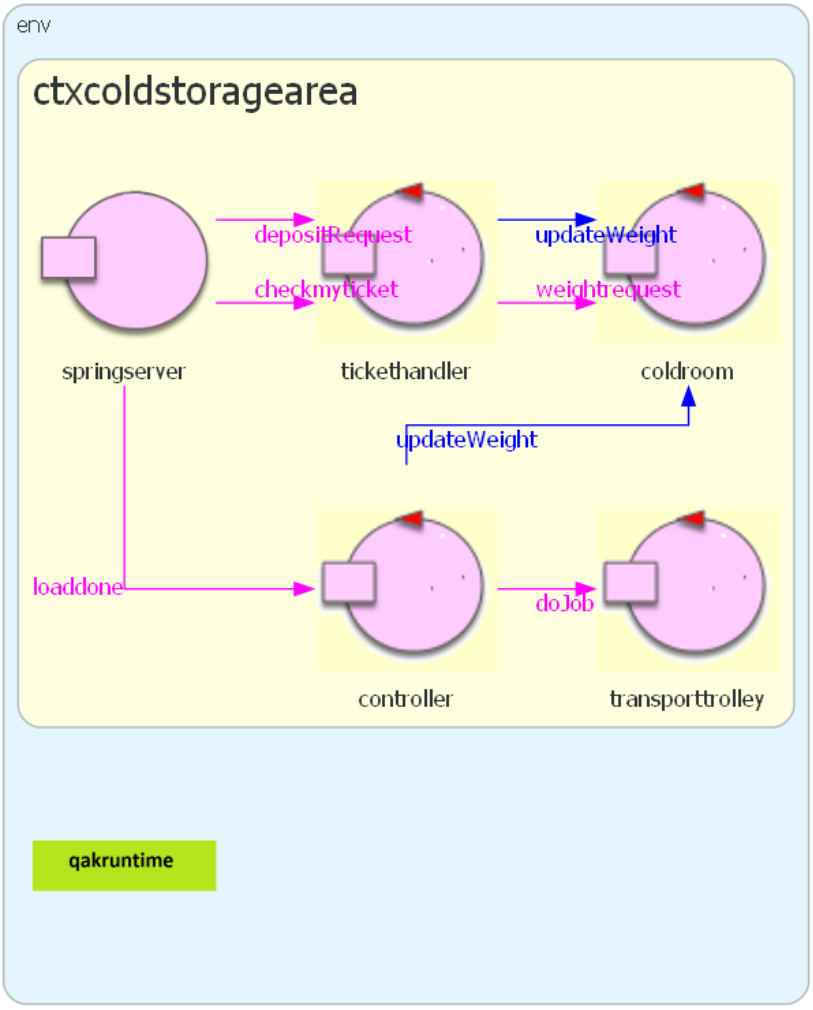
Date le dimensioni dell'area, Service Area sarà divisa in una griglia 4 x 6.
ColdRoom si troverà in posizione (5, 2).

Il TransportTrolley fornito possiede già il supporto a questo tipo di tecnologia. La mappatura della stanza deve essere fatta a priori e fornita tramite file all'avvio.

Peso massimo trasportabile

Dopo discussioni con il committente è stato decretato che il peso da scaricare non sarà mai maggiore del peso trasportabile del robot fisico.

Architettura logica dopo l'analisi del problema



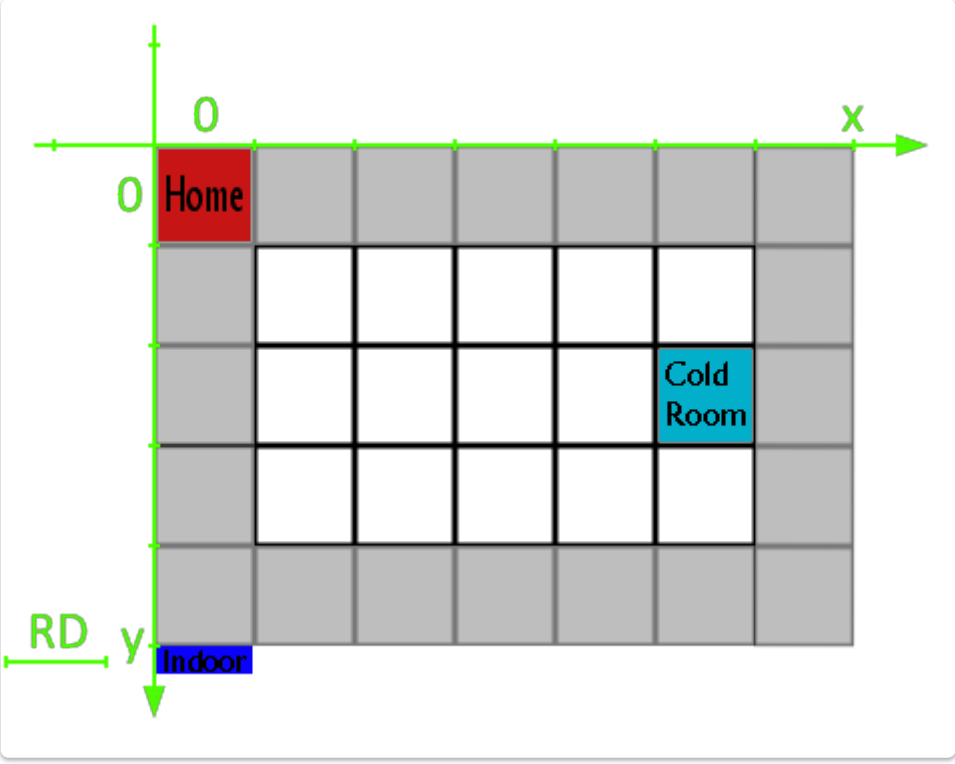
coldstorage2Arch

Progettazione

Sistema di coordinate

Sia RD l'unità di misura

```
Home = (0, 0)
Indoor = (0, 4)
ColdRoom = (5, 2)
ColdRoomPORT = (5, 3)      # posizione del robot per poter scaricare
Service Area = {
        height = 5              # asse x (0 -> 4)
        lenght = 7              # asse y (0 -> 6)
}
```



Definizione messaggi e contesti

```
System coldstorage

//------------------------------------------------------------------------

Request doJob : doJob(KG)
Reply jobdone : jobdone(NO_PARAM)
Reply robotDead : robotDead(NO_PARAM)

Dispatch updateWeight : updateWeight(PESO)

//------------------------------------------------------------------------

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

```
//-------------------------------------------------------------------
```

NOTA: in questo momento ColdRoom è definita nello stesso contesto di Controller, in futuro potrebbe non essere così (dipende dall'implementazione fisica della ColdRoom).

Controller

```
QActor controller context ctxcoldstoragearea {

	[# var KG = 0 #]

	State s0 initial { printCurrentMessage }
	Goto mockRequest

	# generiamo una richiesta casuale per il testing
	State mockRequest {
		[# KG = Math.random() #]
		request transporttrolley -m doJob : doJob($KG)
	} Transition endjob whenReply robotDead -> handlerobotdead
	                                           whenReply jobdone -> jobdone

	State jobdone{
		forward coldroom -m updateWeight : updateWeight($KG)
	} Transition repeat whenTime 15000 -> mockRequest

	State handlerobotdead{
		printCurrentMessage
	}
}
```

ColdRoom

```
QActor coldroom context ctxcoldstoragearea {
	[# var PesoEffettivo = 0 #]

	State s0 initial { printCurrentMessage }
	Transition update whenMsg updateWeight -> updateWeight

	State updateWeight {
		printCurrentMessage
		onMsg ( updateWeight : updateWeight(PESO) ) {
			[# PesoEffettivo += payloadArg(0).toInt() #]
		}
	} Transition update whenMsg updateWeight -> updateWeight
}
```

TransportTrolley

```
QActor transporttrolley context ctxcoldstoragearea {
	[# var Peso = 0 #]

	State s0 initial{
		forward robotpos -m setrobotstate : setpos(0,0,down)		//set
Home pos
	} Transition ready whenMsg robotready -> work

	State work{
		println("robot waiting") color green
	} Transition startworking whenRequest doJob -> startjob		//wait for
doJob

	State startjob{
		onMsg(doJob : doJob( KG )){
			[# Peso = payloadArg(0).toInt() #]
			println("peso ricevuto: $Peso") color green
		}
	} Goto movingtoarrival

	State movingtoarrival{
		request robotpos -m moverobot : moverobot(0,4)
//move to indoor
	} Transition gofetch whenReply moverobotdone -> movingtocoldroom

	State movingtocoldroom{
		request robotpos -m moverobot : moverobot(5,3)
//move to coldroom
	} Transition godrop whenReply moverobotdone -> waitforjob

//alla fine di waitforjob mandiamo la risposta "jobdone" e attendiamo per
verificare //che non ci siano altre richieste "doJob" da portare avanti prima di
tornare alla Home
	State waitforjob {
		replyTo doJob with jobdone : jobdone( 1 )
		println("transporttrolley ! aspetto") color green
	} Transition gofetchagain
			whenTime 3000 -> goinghome
```

```
//torna alla Home
                        whenRequest doJob -> startjob
//torna a scaricare

        State goinghome{
                request robotpos -m moverobot : moverobot(0,0)                //
Home pos
                forward robotpos -m setdirection : dir(down)
        }       Goto work
}
```

Deployment

1. Avviare il container itunibovirtualrobot23 su docker
   Viene lanciato l'ambiente virtuale con il robot all'indirizzo http://localhost:8090/
2. In intellij avviare il file MainCtxbasicrobot.kt del progetto BasicRobot
3. In intellij avviare il file MainCtxColdStorageArea.kt del progetto coldStorage