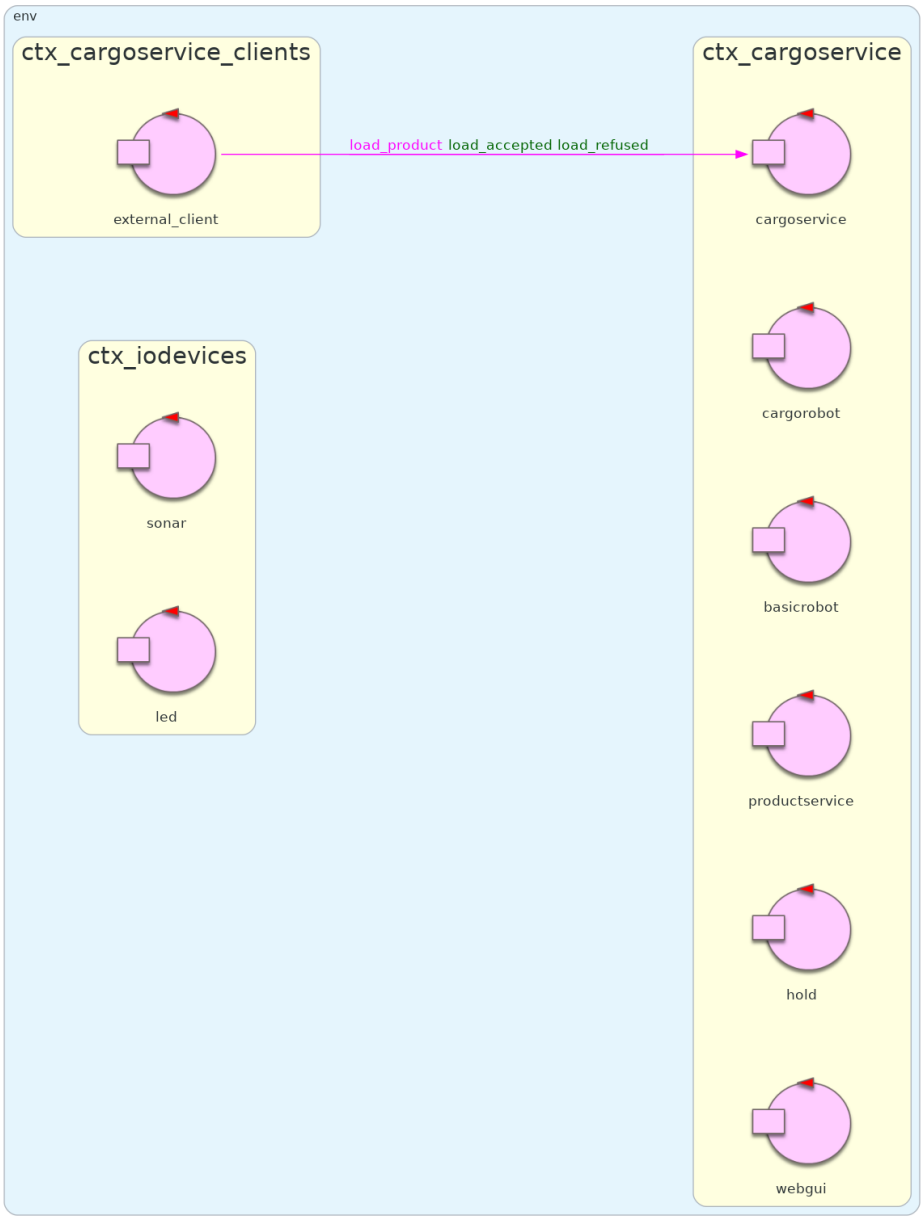
Sprint 1

Punto di partenza

L'analisi dei requisiti avvenuta nello Sprint 0 ha portato a definire una **prima architettura generale del sistema**.



arch0Arch

Obiettivi

L'obiettivo dello sprint 1 sarà affrontare il sottoinsieme dei requisiti relativi ai componenti *cargorservice* e *cargorobot*, effettuandone l'analisi del problema e la progettazione. Particolare importanza verrà data alle **interazioni** che questi componenti dovranno avere con il resto del sistema.

I requisiti affrontati nello sprint 1 saranno i seguenti:

- implementare un sistema che è in grado di accettare/rifiutare le richieste di carico
- implementare un sistema in grado di effettuare un intervento di carico nella sua interezza. Questo significa in ordine:
    - andare all'*IO-port*
    - aspettare il *container* se non è già presente
    - caricare il *container* una volta arrivato
    - spostare il *cargorobot* nella *laydown-position* corretta
    - scaricare il *container* nello slot
    - ritornare alla *home*
- implementare un sistema in grado di interrompere ogni attività in caso di sonar malfunzionante, e in grado di riprendere le attività interrotte una volta risolto il guasto

Va notato che i requisiti affronati in questo sprint presupporrebbere già l'implementazione di altri componenti del sistema come *hold* e *sonar*. L'implementazione di questi componenti verrà però affrontata solamente negli sprint successivi. Per questo motivo nello sprint 1 verranno utilizzati dei **componenti mock** che simuleranno il comportamento dei componenti mancanti in **maniera, però, semplicistica**. Ad esempio, *hold* non causerà mai il rifiuto di una richiesta di carico in quanto non terrà traccia di alcuno stato del deposito.

Analisi del problema | cargoservice

Come detto nello sprint0, l'attore *cargoservice* è il componente principale del sistema. Il suo compito è quello di fare da **orchestratore**; in altre parole, deve coordinare le operazioni degli altri componenti del sistema col fine di eseguire le operazioni specificate dai requisiti nel giusto ordine.

La tipica sequenza di attività di *cargoservice* è la seguente:

1. *cargoservice* **riceve una richiesta di carico** da parte di un cliente.
    - la richiesta di carico contiene il PID del prodotto da caricare
2. dopo aver ricevuto la richiesta di carico, *cargoservice* fa una richiesta a *productservice* per **recuperare il peso del prodotto da caricare** associato al PID ricevuto dal cliente.
3. *cargoservice* riceva la risposta alla sua query da *productservice*. Quest'ultima può contenere:
    - un errore in caso il PID inviato dal cliente non sia registrato nel DB. **In questo caso *cargoservice* può rispondere al cliente con un opportuno messaggio di errore**
    - il peso del prodotto in ogni altro caso
4. dopo aver recuperato il peso del prodotto da caricare, *cargoservice* può passare a verificare se lo **stato del *deposito* permette di soddisfare la richiesta**. Si è definito nello sprint 0 che il mantenimento dello stato del *deposito* è responsabilità del componente *hold*; di conseguenza,

cargoservice invierà a quest'ultimo un messaggio contenente il peso del prodotto da caricare. Si possono verificare tre casi:

   - richiesta non soddisfacibile in quanto si eccederebbe il peso *MaxLoad* del deposito. *Hold* risponde con un opportuno messaggio di errore
   - richiesta non soddisfacibile in quanto manca uno *slot* libero in cui posizionare il *container*. *Hold* risponde con un opportuno messaggio di errore
   - richiesta soddisfacibile. *Hold* risponde con un messaggio contenente il nome dello slot prenotato dalla richiesta corrente

5. se la richiesta viene accettata, *cargoservice* può semplicemente richiedere a *cargorobot* di gestire il container, delegando a lui tutta la logica di attesa, trasporto e deposito del *container* con una operazione del tipo ***handle_load_operation(slot)***.

6. *cargoservice* attende il completamento dell'intervento di carico da parte di *cargorobot*. Nel frattempo, eventuali altre richieste di carico vengono accodate.

7. terminato l'intervento di carico, *cargorobot* sblocca *cargoservice* rispondendo alla sua precedente richiesta (evento di sincronizzazione). Da questo punto in poi *cargoservice* torna a poter servire le richieste di carico.

Considerazioni

Le attività che cargoservice deve effettuare non pongono particolari problemi da analizzare, si tratta solo di effettuare una serie di richieste. Tuttavia, è stata presa una decisione: quella di **rendere il *cargorobot* "intelligente"**.

Si sarebbe potuto rendere il *cargorobot* un mero esecutore di comandi, aggiungendo a *cargoservice* la responsabilità di dettare la sua posizione e che cosa deve fare in ogni momento. Si è preferito, invece, rendere il *cargorobot* più intelligente e indipendente per tre motivi principali:

- l'analisi del dominio effettuata nello sprint 0 ha delineato il *cargorobot* come un componente con delle mosse più sofisticate
- *cargoservice* giova di un *cargorobot* con delle mosse più sofisticate in quanto queste producono un abstraction gap minore
- principio di singola responsabilità: *cargoservice* si occupa di fare solo da orchestrare mentre *cargorobot* si occupa di effettuare le azioni del *DDR* descritto nei requisiti

Nuovi Messaggi

Nello Sprint 0 si erano definiti i messaggi con cui interagire con: *cargoservice*, *productservice* e *basicrobot*. L'analisi della sequenza di attività del *cargoservice* suggerisce i seguenti nuovi messaggi.

Messaggi per l'interazione con hold

```
Request reserve_slot         : reserve_slot(WEIGHT)
Reply   reserve_slot_success : reserve_slot_success(SLOT) for reserve_slot
Reply   reserve_slot_fail    : reserve_slot_fail(CAUSA) for reserve_slot
```

Messaggi per l'interazione con cargorobot

```
Request handle_load_operation   : handle_load_operation(SLOT)
Reply   load_operation_complete : load_operation_complete(OK) for
handle_load_operation
```

Modello cargoservice

L'analisi della sequenza di attività suggerisce anche gli stati dell'attore QAK con cui modellare cargoservice

```
QActor cargoservice context ctx_cargoservice {
    [#
        var Cur_prod_PID = -1
        var Cur_prod_weight = -1
    #]

    State s0 initial{
        println("$name | STARTS") color blue
    }
    Goto wait_request


    /* inizio ciclo di servizio */

    State wait_request{
        println("$name | waiting for request") color blue
    }
    Transition t0
       whenRequest load_product -> get_prod_weight


    /* Tento di recuperare il peso del prodotto richiesto */

    State get_prod_weight {
        onMsg( load_product : load_product(PID) ) {
            [# Cur_prod_PID = payloadArg(0).toInt() #]
            println("$name | checking with productservice
                     for the weight of PID: $Cur_prod_PID") color blue

            request productservice -m getProduct : product($Cur_prod_PID)
        }
    }
    Transition t0
       whenReply getProductAnswer -> check_prod_answer


    State check_prod_answer {
        onMsg( getProductAnswer : product( JSonString ) ) {
            [#
                val jsonStr = payloadArg(0)
```

```
            Cur_prod_weight = Product.getJsonInt(jsonStr, "weight")
        #]

        println("$name | the weight of PID: $Cur_prod_PID") color blue
    }
}
Goto reserve_slot if [# Cur_prod_weight > 0 #] else get_weight_fail


State get_weight_fail {
    [#
        val Causa = "Non è stato possibile recuperare il peso di
                     PID: $Cur_prod_PID in quanto non registrato
                     dentro a productservice."
    #]
    println("$name | $Causa") color red

    replyTo load_product with load_refused : load_refused($Causa)
}
Goto wait_request


/* Tento di prenotare uno slot */

State reserve_slot {
    println("$name | checking with hold if a reservation with
             (PID: $Cur_prod_PID, KG: $Cur_prod_weight)
             is possible") color blue

    request hold -m reserve_slot : reserve_slot($Cur_prod_weight)
}
Transition t0
   whenReply reserve_slot_success -> load_cargo
   whenReply reserve_slot_fail    -> reserve_slot_fail


State reserve_slot_fail {
    onMsg( reserve_slot_fail : reserve_slot_fail(CAUSA) ) {
        [#
            val Causa = payloadArg(0)
            val CausaMsg = "impossibile prenotare uno slot per
                            (PID: $Cur_prod_PID, KG: $Cur_prod_weight).
                            \n\tCausa: $Causa"
        #]
        println("$name | $CausaMsg") color red

        replyTo load_product with load_refused : load_refuse($CausaMsg)
    }
}
Goto wait_request


/* Richiesta soddisfacibile */
```

```
    State load_cargo {
        onMsg( reserve_slot_success : reserve_slot_success(SLOT) ) {
            [#
                val Reserved_slot = payloadArg(0)
            #]
            // rispondo al cliente
            println("$name | (PID: $Cur_prod_PID, KG: $Cur_prod_weight)
                     is going to be placed in slot: $Reserved_slot")
                     color green
            replyTo load_product with load_accepted :
                                      load_accepted($Reserved_slot)
            // attivo il cargorobot
            println("$name | waiting for load completion") color blue
            request cargorobot -m handle_load_operation :
                                  handle_load_operation($Reserved_slot)
        }
    }
    Transition t0
       whenReply load_operation_complete -> load_request_done


    State load_request_done {
        println("$name | product (PID: $Cur_prod_PID, KG: $Cur_prod_weight)
                  successfully loaded!") color green
    }
    Goto wait_request
}
```

Analisi del problema | cargorobot

Come detto nello sprint 0, e consolidato durante l'analisi di *cargoservice*, l'attore *cargorobot* è il componente responsabile delle attività del *DDR* all'interno del *deposito*. *cargorobot* implementa le sue azioni logiche comunicando con il *basicrobot*, che a sua volta comunica con l'ambiente virtuale *WEnv*. Il tutto per effettuare gli interventi di carico richiesti da *cargoservice*.

Con l'analisi dei requisiti e l'analisi di *cargoservice* si è già delineata la sequenza di attività del *cargorobot*:

1. *cargorobot* riceve da *cargoservice* una richiesta di gestione di un container contenente il nome dello *slot* riservato.

2. *cargorobot* si dirige verso la pickup-position e aspetta che arrivi il container se non è già presente.

3. Una volta che il container è arrivato all'*IO-port*, *cargorobot*:

    - recupera il container
    - si posizione nella corretta *laydown-position*
    - deposita il container nello *slot* prenotato

4. terminato l'intervento di carico, *cargorobot* può ritornare alla *home* rispondendo a *cargoservice* con un messaggio di successo.

Considerazioni

- Una volta ricevuta la risposta di successo, ***cargoservice* torna a essere recettivo** a richieste di carico da parte dei clienti e a poter servire quelle che si sono nel frattempo accodate.
- Nulla vieta che *cargorobot* possa incominciare a effettuare immediatamente altri interventi di carico **prima di essere ritornato alla home**. Anzi, sarebbe una soluzione più efficiente.
- Il requisito numero 5 specifica che **in un qualsiasi momento *cargorobot* può essere interrotto** fino a quando il sonar non smette di essere difettoso.
- Dai requisiti forniti, **non c'è nessun motivo per cui un intervento di carico dovrebbe fallire**. Di conseguenza, l'unica risposta contemplata da *cargoservice*, in seguito ad una richiesta verso *cargorobot*, è una risposta di successo.

Problematiche

L'analisi fatta fino ad ora evidenzia una serie di problematiche.

Come fa *cargorobot* a conoscere le coordinate a cui si deve posizionare, e l'orientamento che deve avere, dato solo il nome dello *slot* riservato nella richiesta di intervento di carico?

cargorobot dovrà mantenere nel suo stato una **mappa** che associa: nomi degli slot, con le coordinate delle laydown-position e l'orientamento che deve avere una volta raggiunte quest'ultime.

Come fa *cargorobot* a interrompere/riprendere le sue attività?**

Siccome il guasto del sonar può avvenire in qualsiasi momento, *cargorobot* dovrà essere **sempre recettivo all'evento di guasto** per potersi interrompere tempestivamente. Questa significa che *cargorobot* dovrà necessariamente avere uno **stato persistente di attesa**, in cui attende che il *sonar* riprenda a funzionare, ed uno **stato di ripristino**, in cui riprende cio che stava facendo prima.

Come fa *cargorobot* a bloccare il *basicrobot* in movimento?**

Un guasto del sonar può avvenire dopo che *cargorobot* ha comandato *basicrobot* di spostarsi, ma prima che *basicrobot* abbia terminato lo spostamento. In queste situazioni, *cargorobot* può entrare nello stato di attesa citato prima, ma *basicrobot* (che è un attore dotato di un proprio flusso di controllo) continuerebbe ad eseguire il piano di spostamento.

Fortunatamente, il committente ha previsto un evento di nome: ***"Event alarm : alarm(X)"*** nell'interfaccia del *basicrobot* con cui è possibile interrompere l'esecuzione del piano.

Per bloccare il *basicrobot* è quindi sufficiente emettere l'evento alarm nello stato persistente di attesa di *cargorobot*.

Come fa *cargorobot* a ricordarsi dove doveva andare una volta interrotto durante uno spostamento?**

cargorobot dovrà mantenere nel suo stato due informazioni aggiuntive:

- un flag che indica se si stava muovendo
- la sua destinazione corrente

In questa maniera, se interrotto durante uno spostamento, nello stato di ripristino sarà possibile inviare a *basicrobot* una nuova richiesta di spostamento verso la destionazione.

Se *cargorobot* viene interrotto mentre non si sta spostando, nello stato di ripristino non sarò necessario effettuare questa richiesta.

Come fa *cargorobot* a sapere se il *container* è già presente all'IO-port o meno, prima di arrivarci?

Similmente ai guasti del *sonar*, **un *container* può arrivare all'IO-port in qualsiasi momento** e per questo motivo *cargorobot* dovrà essere sempre recettivo agli eventi del *sonar* che avvisano della presenza/assenza di un *container*.

A questo punto, sarà sufficiente che *cargorobot* mantenga nel suo stato un **flag booleano** che salva l'informazione riguardante la presenza/assenza di un container. *cargorobot* aggiornerà questo stato con delle **routine di gestione** che si attivano in corrispondenza degli eventi del sonar.

Nuovi messaggi

L'analisi fatta fino ad ora porta a definire i seguenti nuovi messaggi.

Eventi del sonar

```
Event container_arrived : container_arrived(X)
Event container_absent  : container_absent(X)
Event interrompi_tutto  : interrompi_tutto(X)  "evento che avvisa di un
guasto del sonar"
Event riprendi_tutto    : riprendi_tutto(X)    "evento che avvisa del
ripristino del sonar"
```

Questi messaggi sono stati modellati come eventi in quanto è presumibile che avranno altri componenti oltre a *cargorobot* come destinatari negli sprint successivi

- *web-gui* <- container_arrived, container_absent
- *led* <- interrompi_tutto, riprendi_tutto

Modello cargorobot

L'analisi fatta fino ha portato al seguente modello.

```
QActor cargorobot context ctx_cargoservice{
    [#
        // stato
        val Step_len = 330

        val positions = hashMapOf(
            "home"      to arrayOf(0, 0),
            "io_port"   to arrayOf(0, 4),
            "slot1"     to arrayOf(1, 1),
            "slot2"     to arrayOf(1, 3),
```

```
            "slot3"     to arrayOf(4, 1),
            "slot4"     to arrayOf(4, 3)
        )

        val directions = hashMapOf(
            "home"      to "down",
            "io_port"   to "down",
            "slot1"     to "right",
            "slot2"     to "right",
            "slot3"     to "left",
            "slot4"     to "left"
        )

        lateinit var Destination : String
        lateinit var Reserved_slot : String

        var moving  = false

        var container_present = false
    #]

    State s0 initial {
        println("$name | STARTS") color magenta

        println("$name | $MyName engaging ... ") color yellow
        request basicrobot -m engage : engage($MyName, $Step_len)
    }
    Transition t0
        whenReply engagedone    -> wait_request
        whenReply engagerefused -> end


    /* inizio ciclo di servizio */

    State wait_request{
        println("$name | waiting for request") color magenta
        [# moving = false #]
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenRequest handle_load_operation    -> go_to_io_port


    /* vado a prendere il container */

    State go_to_io_port {
        // aggiorno il mio stato recuperando le coordinato dello slot
prenotato
        onMsg( handle_load_operation : handle_load_operation(SLOT) ) {
            [#
                Reserved_slot = payloadArg(0)
```

```
                // il doppio !! serve a dire al compilatore Kotlin di stare
tranquillo
                // e di recuperare il valore dalla mappa anche senza fare
dei null-check
                val coords = positions[Reserved_slot]!!
                val X = coords[0]
                val Y = coords[1]
            #]
            // DEBUG:
            println("$name | cargorobot reserved_slot is $Reserved_slot =
($X, $Y)") color magenta
        }

        // vado verso la io-port
        println("$name | going to io-port") color magenta
        [#
            // aggiorno la mia destinazione per ricordarmi dove devo andare
in caso di interruzioni
            Destination = "io_port"

            val coords = positions[Destination]!!
            val X = coords[0]
            val Y = coords[1]
        #]

        request basicrobot -m  moverobot : moverobot($X, $Y)

        [# moving = true #]
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenReply moverobotdone              -> arrived_at_io_port


    State arrived_at_io_port {
        println("$name | arrived at io-port") color magenta
        [#
            moving = false

            val Direction = directions[Destination]!!
        #]
        forward basicrobot -m setdirection : dir($Direction)

        // se il container c'è gia, mi mando un autodispatch cosi da non
dover aspettare
        if [# container_present #] {
            [# container_present = false #]
            autodispatch continue : continue(si)
        }
    }
    Transition t0
        whenInterruptEvent interrompi_tutto -> wait_resume_msg
```

```
        whenMsg continue                    -> pick_up_container
        whenEvent container_arrived         -> pick_up_container


    State pick_up_container {
        println("$name | picking up container") color magenta
        [# moving = false #] // non cambia rispetto allo stato precedente
ma meglio essere espliciti

        delay 3000 // tempo arbitrario per caricare il container sul
cargorobot

        // durante il carico del container potrebbe essere arrivato una
interruzione,
        // mi mando un messaggio per ricordarmi che posso procedere
        autodispatch continue : continue(si)
    }
    Transition t0
        whenInterruptEvent interrompi_tutto -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenMsg continue                    -> go_to_reserved_slot


    /* vado a depositare il container */

    State go_to_reserved_slot {
        [#
            // aggiorno la mia destinazione per ricordarmi dove devo andare
in caso di interruzioni
            Destination = Reserved_slot

            val coords = positions[Destination]!!
            val X = coords[0]
            val Y = coords[1]
        #]

        println("$name | going to my reserved slot: $Reserved_slot = ($X,
$Y)") color magenta

        request basicrobot -m  moverobot : moverobot($X, $Y)
        [# moving = true #]
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenReply moverobotdone              -> arrived_at_reserved_slot


    State arrived_at_reserved_slot {
        println("$name | arrived at reserved slot") color magenta
        [#
            moving = false
```

```
            val Direction = directions[Destination]!!
        #]
        forward basicrobot -m setdirection : dir($Direction)

        // scarico il container
        println("$name | laying down the container") color magenta
        delay 3000 // tempo arbitrario per scaricare il container dal
cargorobot

        // duranto lo scarico potrebbe essere arrivato una interruzione,
        // mi mando un messaggio per ricordarmi che posso procedere
        autodispatch continue : continue(si)
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenMsg continue                     -> back_to_home


    /* torno a casa */

    State back_to_home {
        // rispondo a cargoservice
        println("$name | load operation completed") color magenta
        replyTo handle_load_operation with load_operation_complete :
load_operation_complete(ok)

        // mi avvio verso casa
        println("$name | Back to home") color magenta
        [#
            // aggiorno la mia destinazione per ricordarmi dove devo andare
in caso di interruzioni
            Destination = "home"

            val coords = positions[Destination]!!
            val X = coords[0]
            val Y = coords[1]
        #]

        request basicrobot -m  moverobot : moverobot($X, $Y)
        [# moving = true #]
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenReply moverobotdone              -> at_home
        whenRequest handle_load_operation    -> stop_going_to_home // servo
subito eventuali richieste in coda


    State stop_going_to_home {
```

```
        println("$name | stop going to home and start serving new request
immediately") color magenta
        emit alarm : alarm(blocca) // blocco il basicrobot
        [# moving = false #]

        // aggiorno il mio slot prenotato (non viene fatto in
'go_to_io_port' dato che consumo
        // la richiesta 'handle_load_operation' prima di arrivarci, e
quindi il relativo blocco
        // onMsg{} di 'go_to_io_port' non viene eseguito)
        onMsg( handle_load_operation : handle_load_operation(SLOT) ) {
            [#
                Reserved_slot = payloadArg(0)

                // il doppio !! serve a dire al compilatore Kotlin di stare
tranquillo
                // e di recuperare il valore dalla mappa anche senza fare
dei null-check
                val coords = positions[Reserved_slot]!!
                val X = coords[0]
                val Y = coords[1]
            #]
            // DEBUG:
            println("$name | cargorobot reserved_slot is $Reserved_slot =
($X, $Y)") color magenta
        }

        // anche se estremamente improbabile, anche durante questo mini-
stato
        // potrebbe essere arrivata una interruzione.
        // Mi mando un messaggio per ricordarmi che posso procedere
        autodispatch continue : continue(si)
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenMsg continue                     -> go_to_io_port


    State at_home{
        println("$name | at home") color magenta
        forward basicrobot -m setdirection : dir(down)
        [#
            moving = false

            val Direction = directions[Destination]!!
        #]
        forward basicrobot -m setdirection : dir($Direction)
        // anche se estremamente improbabile, anche durante questo mini-
stato
        // potrebbe essere arrivata una interruzione.
        // Mi mando un messaggio per ricordarmi che posso procedere
        autodispatch continue : continue(si)
```

```
    }
    Transition t0
        whenInterruptEvent interrompi_tutto  -> wait_resume_msg
        whenInterruptEvent container_arrived -> container_arrived_handler
        whenInterruptEvent container_absent  -> container_absent_handler
        whenMsg continue                     -> wait_request


    /* gestisco le interruzioni */

    State wait_resume_msg {
        println("$name | sonar malfunzionante, mi fermo") color red

        emit alarm : alarm(blocca) // blocco il basicrobot
    }
    Transition t0
        whenEvent riprendi_tutto -> resume


    State resume {
        println("$name | riprendo") color green

        // se il basic robot si stava muovendo, gli dico di nuovo dove deve
andare
        // altrimenti rimane fermo dove è stato interrotto
        if [# moving #] {
            [#
                val coords = positions[Destination]!!
                val X = coords[0]
                val Y = coords[1]
            #]
            request basicrobot -m  moverobot : moverobot($X, $Y)
        }

        returnFromInterrupt
    }


    State container_arrived_handler {
        println("$name | container arrivato") color yellow

        [# container_present = true #]
        returnFromInterrupt
    }

    State container_absent_handler {
        println("$name | container assente") color yellow

        [# container_present = false #]
        returnFromInterrupt
    }


    /* esco */
```

```
    State end{
        println("$name | ENDS ") color red
        [# System.exit(0) #]
    }
}
```

Piano di test

// richiestata rifiutata perchè prodotto non esiste

// test delle interruzioni tramite sonar_mock

Progettazione

Fase di progetto e realizzazione, che termina con il **deployment** del prodotto, unitamente a istruzioni (ad uso del committente) per la costruzione/esecuione del prdotto setsso.

parla di un file di config

parla di productservice e basicrobot usati come servizi esterni

parla del bug di cargorobot durante il tentativo di servire richieste immediatamente durante la fase di ritorno

parla di eventuali classi di supporto

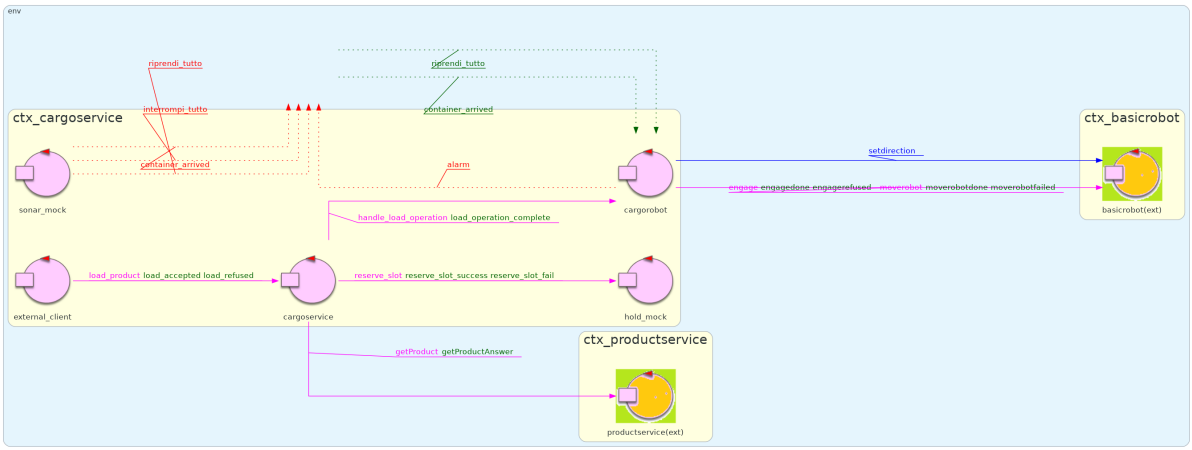
ci sta anche lasciare inalterata della roba

Deployment

Sintesi finale e nuova architettura

Ogni SPRINT dovrebbe terminare con una pagina di sintesi che riporta l'architettura finale corrente del sistema (con i link al modello e ai Test). Questa pagina sarà l'inizio del documento relativo allo SPRINT successivo.

Alla fine dello SPRINT, l'ARCHITETTURA INIZIALE DI RIFERIMENTO avrà subito una evoluzione che produce una nuova nuova ARCHITETTURA DI RIFERIMENTO, che sarà la base di partenza per lo sprint successivo.


env
ctx_cargoservice
ctx_basicrobot
ctx_productservice
riprendi_tutto
interrompi_tutto
riprendi_tutto
container_arrived
container_arrived
alarm
setdirection
engage engagedone engagerefused moverobot moverobotdone moverobotfailed
handle_load_operation load_operation_complete
load_product load_accepted load_refused
reserve_slot reserve_slot_success reserve_slot_fail
getProduct getProductAnswer
sonar_mock
cargorobot
basicrobot(ext)
external_client
cargoservice
hold_mock
productservice(ext)

arch1Arch