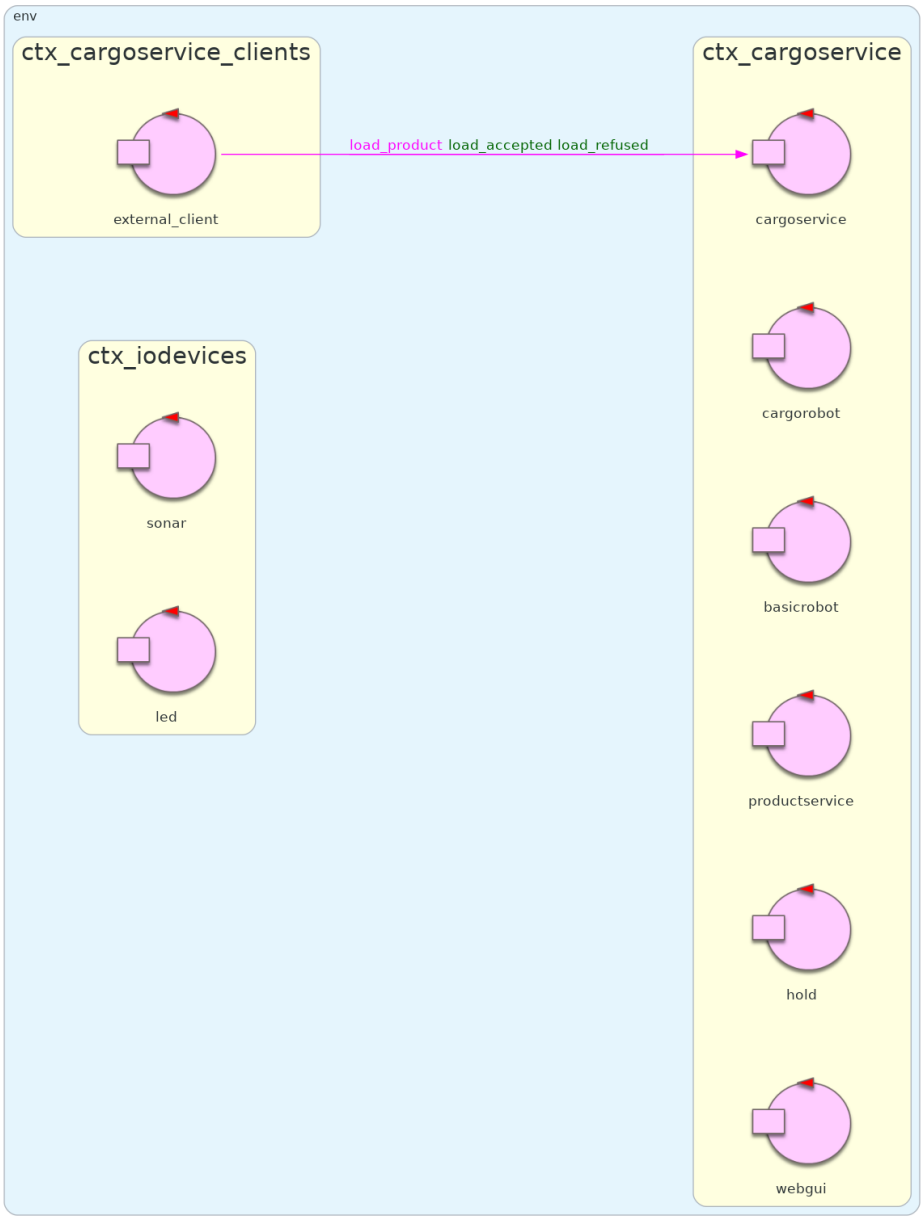
Sprint 1

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

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 cargorobot

// responsabilità e userstory di cargorobot

Durante l'analisi dei requisiti si è detto che il *cargorobot* è il componente responsabile del comando del DDR. Il *cargorobot* si interfaccia con il *basicrobot* (fornito dal committente) per muovere il DDR. Estendendo le funzionalità del *basicrobot* **colma l'abstraction gap** tra quest'ultimo e i requisiti.

L'analisi dei requisiti e l'analisi di *cargoservice* hanno già delineato in parte la sequenza di attività del *cargorobot*:

1. *cargorobot* riceve da *cargoservice* una richiesta di gestione di un container
2. *cargorobot* si dirige verso la pickup-position e aspetta che arrivi il container
    - **PUNTO APERTO: chi è che ascolta il sonar e aggiorna lo stato di cargorobot per confermare la presenza del container???**
3. *cargorobot* :
    - recupera il container
    - trasporta il container allo *slot* prenotato (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* del successo del suo intervento di carico

- **NB**: da questo momento in poi *cargoservice* torna ad essere recettivo a richieste di carico da parte dei clienti
- **NB2**: da questo momenti in poi *cargorobot* torna ad essere recettivo a richieste di carico da parte di *cargoservice*
- **NB3**: per adesso non c'è nessun motivo di fallimento per il cargorobot (timeout??) e quindi la risposta serve solo a sincronizzare *cargoservice* e *cargorobot*

NB: in un qualsiasi momento l'attività del cargorobot può essere interrotta. **PUNTO APERTO: come fa cargorobot a interrompersi e riprendere???**

(sostituire i punti aperti con domande e risposte) PUNTO APERTO: Come fa *cargorobot* a conoscere le posizioni notevoli in cui deve andare dato il nome di uno slot?

PUNTO APERTO: come fa *cargorobot* a sapere se il container è già presente all'IO-port o no?

- **PUNTO APERTO: il commmittente ci ha detto che possiamo fare come ci pare per quanto riguarda il momento in cui *cargoservice* può tornare a servire le richieste... la nostra scelta però deve essere opportunamente motivata. Cosa scegliamo???**

// codice qak di cargorobot

Definizione dei messaggi

Nuova architettura

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.

Piano di test

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 come ci si deve adattare all'interfaccia di productservice

parla di eventuali classi di supporto

ci sta anche lasciare inalterata della roba

(Opzionale) Osservabilità (Logging con prolog?)

// Deployment

Sintesi finale

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.