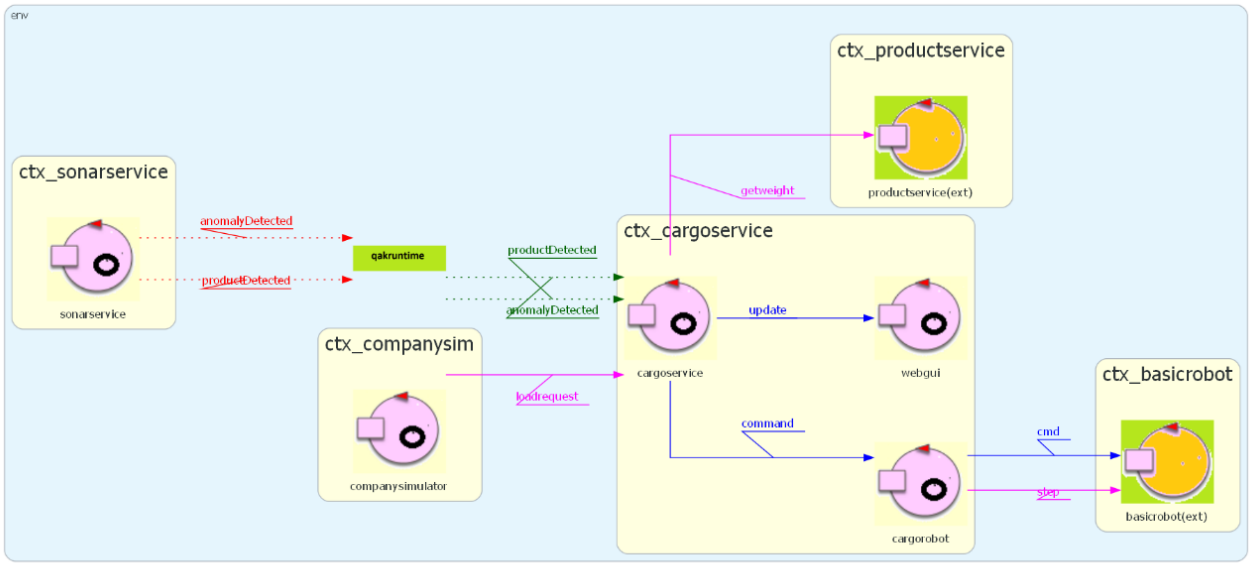
Sprint 1:

Punto di partenza: attraverso l'analisi dei requisiti del committente nello Sprint0, il team è riuscito a definire una prima architettura generale del problema.



cargoservice_overviewArch

Inoltre, si vuole fornire per comprensione un vocabolario riassuntivo di termini definiti nello sprint0.

Vocabolario:

Termine	Significato attribuito
Container	Contenitore in cui viene inserito il prodotto
loadrequest /richiesta di carico	Richiesta mandata dalla compagnia, specificando un PID
prodotto	Nel sistema è l'oggetto trasportato dal robot, la cui presenza può innescare diversi eventi
prodotto registrato	Prodotto conosciuto da ProductService a cui è associato un PID e un peso(Weight)
Microservizio	Componente software progettato per svolgere una specifica funzione del sistema. Ogni microservizio comunica con gli altri tramite messaggi, rendendo il sistema flessibile e scalabile.
GUI (Graphical User Interface) /WebGUI	Interfaccia grafica utente accessibile via web, che consente di visualizzare in tempo reale lo stato della stiva e interagire in modo intuitivo con il sistema.
Bounded Context	Il "bounded context" (**contesto limitato**) è un concetto fondamentale nel Domain-Driven Design (DDD) e si riferisce a un ambito applicativo ben definito e autonomo all'interno del quale vengono definite entità, regole e logiche di business in modo univoco e chiaro.

	All'interno di un bounded context, il significato di ogni entità o concetto è inequivocabile e specifico per quel contesto, evitando ambiguità e conflitti con altri contesti.
IOPort	Punto fisico (porta) attraverso il quale i contenitori dei prodotti entrano o escono dalla nave. È il punto in cui il sonar rileva la presenza di un prodotto.
Sonar	Sensore a ultrasuoni che misura la distanza tra sé e un oggetto. Nel nostro sistema serve per rilevare se un contenitore è presente all'IOPort.
DDRobot	è un robot che utilizza due motori indipendenti per muovere le ruote o i cingoli. È il supporto fisico che viene comandato da cargorobot.
PID (Product Identifier)	Numero intero univoco assegnato a ciascun prodotto registrato, usato per tracciarne l'identità all'interno del sistema.
Slot	Spazio fisico nella stiva della nave dove può essere posizionato un contenitore. Esistono 4 slot disponibili; uno è sempre occupato (slot5).
Cargorobot	Robot mobile autonomo (a guida differenziale) incaricato di trasportare i contenitori dall'IOPort fino allo slot assegnato e poi tornare alla posizione HOME.
Stiva	Area rettangolare della nave in cui i contenitori vengono caricati. Contiene gli slot e l'IOPort.
ProductService	Microservizio che gestisce la registrazione dei prodotti. Verifica i dati e assegna un PID univoco
CargoService	Microservizio che riceve richieste di carico, controlla i vincoli, assegna gli slot e coordina il caricamento tramite cargorobot.
SonarService	Microservizio che rileva la presenza di un contenitore all'IOPort tramite i dati forniti dal sonar.
DFREE	Distanza soglia usata dal sonar: se la distanza misurata è maggiore di DFREE per 3 secondi, si ipotizza un malfunzionamento del sensore.
MaxLoad	Peso massimo complessivo che la nave può sopportare. Il sistema rifiuta richieste che farebbero superare questo limite.
Worker	Persona che colloca fisicamente i contenitori sull'IOPort dopo che sono stati registrati.
Sistema logico di riferimento	Rappresentazione concettuale dell'intero sistema, con attori, componenti e interazioni, usata come base per l'architettura e la progettazione tecnica.

attore	Entità che svolge un ruolo attivo nel sistema, eseguendo azioni e comunicando con gli altri attori attraverso messaggi
Linguaggio QAK	Linguaggio modellistico usato per descrivere e simulare il comportamento dei componenti del sistema come “attori”
POJO	Plain Old Java Object : un oggetto di una classe in java

GOAL dello Sprint 1:

1. Enunciazione esplicita dei requisiti del cargoservice
2. Analisi dei requisiti enunciati
3. Definizione dell’architettura logica con modello eseguibile in qak
4. Progetto e realizzazione

1. Enunciazione esplicita dei requisiti:

Per automatizzare le operazioni di carico nella stiva di una nave, il sistema software si articola in microservizi autonomi ma cooperanti, tra cui, cargoservice.

Esso rappresenta il nucleo centrale del processo:

- Riceve le richieste di carico
- Valuta le richieste di carico (le accetta o rifiuta)
- Assegna gli slot ai prodotti
- Riceve le informazioni dal sonar
- Coordina i movimenti del DDrobot (sottosistema cargorobot)

Il team di sviluppo si propone, in questa fase, di analizzare in modo tecnico i requisiti del microservizio da sviluppare cargoservice.

2.Analisi dei requisiti enunciati:

Esplicitiamo le funzionalità, comportamenti e vincoli che il sistema deve necessariamente e logicamente rispettare.

RF1. Gestione richiesta di carico: il sistema deve gestire correttamente le richieste di carico ricevute. Queste richieste arrivano con il PID del prodotto da caricare.

RF1.1 Controllo esistenza prodotto / Get Weight:

cargoservice richiede al product-service informazioni riguardo il prodotto (PID), ovvero il peso.

Se ciò non va a buon fine, si deve rifiutare la richiesta di carico.

RF1.2 Validazione della richiesta:

Una volta ottenuto il peso del prodotto PID, cargoservice fa un ulteriore controllo: rifiuta la richiesta se il peso del prodotto supera il MaxLoad o se la stiva è già piena (non ci sono slot disponibili).

In caso contrario, assegna uno slot al prodotto.

RF1.3. Attesa rilevamento contenitore: una volta validata la richiesta, il cargoservice deve attendere che gli arrivi un segnale (da parte del sonarservice) che indica la presenza di un contenitore all'IOPort.

RF1.4. Trasporto contenitore:

una volta rilevato il contenitore all'IOPort, il cargoservice manda un comando al cargorobot affinché coordini il DDRobot perché recuperi il contenitore e lo porti fino allo slot assegnato.

RF2. Aggiornamenti stiva: cargoservice manda aggiornamenti periodici alla webgui per mostrare lo stato attuale della stiva.

RF3. Gestione anomalia sonar: cargoservice deve gestire e segnalare i malfunzionamenti del sonar fermando il DDrobot momentaneamente.

Riprende le attività una volta ripristinata la condizione corretta.

Requisiti non funzionali (vincoli su prestazioni, sicurezza, usabilità):

- **AFFIDABILITÀ**:
  - Il sistema deve essere in grado di rilevare e gestire anomalie hardware (es. malfunzionamento del sonar), garantendo una risposta controllata senza crash e blocchi irreversibili
- **REATTIVITÀ / TEMPISTICHE:**
  - **Il sistema deve reagire entro un tempo T che sia 'ragionevole':** viene considerata ragionevole un tempo di circa tre secondi.
- **DISPONIBILITÀ:**
  - Il servizio deve essere sempre attivo tranne durante l'elaborazione di una richiesta o un'anomalia. Deve riprendere autonomamente appena possibile.
- **USABILITÀ:**
  - La GUI deve essere intuitiva e aggiornata dinamicamente.
- **SCALABILITÀ TECNICA:**

  Architettura a microservizi indipendenti, scalabili e distribuibili su più nodi computazionali.
- **MODULARITÀ / MANUTENIBILITÀ:**

  Ogni componente deve essere testabile e aggiornabile separatamente.

Vincoli fisici o tecnici (legati all'hardware):

- Il PID deve essere univoco
- Ci sono cinque slot, di cui il quinto sempre occupato.
- Gli slot una volta liberi sono riutilizzabili.
- I prodotti devono essere inseriti di dimensioni predefinite e registrate
- Il robot mobile è di tipo Differential Drive, ovvero è un robot che utilizza due motori indipendenti per muovere le ruote o i cingoli. È il supporto fisico che viene comandato da cargorobot.
- La stiva può contenere un massimo peso MaxLoad, quindi ogni richiesta di carico deve essere validata in base al peso totale attuale sommato a quello del nuovo carico.
- L'area della stiva è rettangolare, piatta e dotata di un solo punto di Input/Output (IOPort)

Analisi del problema:

Il gruppo di sviluppo propone la realizzazione di un sistema software distribuito, articolato su più nodi computazionali, con l'obiettivo di automatizzare le operazioni di carico dei contenitori nella stiva della nave.
Il sistema è composto da microservizi indipendenti, che devono cooperare e comunicare in modo coordinato, nonostante siano soggetti a eventi asincroni provenienti dal mondo fisico.

Per raggiungere questo obiettivo, il sistema deve affrontare e risolvere una serie di problematiche chiave, tra cui:

1. **La gestione sequenziale (in modo ordinato e non concorrente) delle richieste:**
   mentre il sistema sta controllando o eseguendo una richiesta di carico che ha accettato, non può considerarne altre, ovvero, non torna recettivo finché la richiesta non viene rifiutata o il prodotto non viene depositato nello slot assegnato.
2. **La sincronizzazione di eventi reali non deterministici:**
   - La richiesta di carico da parte della compagnia,
   - La presenza fisica del contenitore all'IOPort,
   - Il caricamento del contenitore nello slot da parte del robot.
   - La risposta da parte del productservice (microservizio fornito)

   il sistema dipende dall'avvenimento di eventi asincroni che quindi richiedono una gestione a stati del cargoservice, con sospensione temporanea dell'elaborazione finché una fase non è completata.
3. **Controllo del robot differenziale DDRobot:**
   cargoservice è responsabile dell'invio di comandi al sottoservizio cargorobot, che si occupa direttamente del controllo del DDRobot.
   il robot utilizzato è fornito esternamente, la sua interfaccia è nota ma non modificabile.

   Il sistema deve essere in grado di guidare il robot con comandi precisi per raggiungere partendo da un qualsiasi punto della stiva:

- Il posizionamento di default HOME
- La IOPort
- Lo slot assegnato

Il sistema deve quindi conoscere la mappa della stiva per poter pianificare i movimenti del robot.

4. **il monitoraggio dinamico dello stato della stiva**:
   il sistema deve mantenere lo stato coerente e aggiornato lo stato dei 4 slot utilizzabili, considerando:
   - assegnazione dinamica degli slot,
   - occupazione di uno slot dopo il caricamento del prodotto.

   La WebGUI deve ricevere questi aggiornamenti in tempo reale.

2. Definizione dell'architettura logica con modello eseguibile in qak:

Disclaimer:

I requisiti affrontati in questo sprint presuppongono l'implementazione di alcuni componenti, ovvero, il sonarservice e la webgui. La loro progettazione sarà affrontata in realtà nei successivi sprint, difatti essi verranno simulati per adesso attraverso degli elementi mock.

Cargoservice:

È il componente principale del sistema e rappresenta il nucleo del problema da affrontare in fase di progettazione.

Per mantenere un elevato livello di astrazione in cargoservice ed evitare l'introduzione di logiche troppo tecniche, si è scelto di suddividere le responsabilità tra questo modulo e un altro attore appartenente allo stesso contesto (cargorobot).

Cargoservice non comunicherà direttamente con il microservizio basicrobot fornito dal committente; tale interazione sarà invece gestita da cargorobot.

Sequenza di azioni di Cargoservice

1. Ricezione di una richiesta di carico da parte della compagnia, contenente il PID del prodotto da caricare.
2. Richiesta al product-service del peso del prodotto associato al PID.
3. **Una volta ricevuta la risposta dal product-service:**
   3.1. Se la risposta contiene il peso del prodotto (risposta positiva), si procede alla **validazione della richiesta**:

- La richiesta viene rifiutata se il peso supera il valore di **MaxLoad** o se la stiva risulta **piena** (assenza di slot disponibili).
- In caso contrario, viene **assegnato uno slot** al prodotto.
  3.2. Se la risposta è negativa (prodotto inesistente o errore), la richiesta di carico viene rifiutata.

4. Dopo l'assegnazione dello slot, cargoservice attende un **segnale di rilevamento** da parte del sonarservice, che confermi la presenza di un contenitore presso l'IOPort.
5. Una volta rilevato il contenitore, cargoservice invia un comando a cargorobot, il quale coordina il DDRobot per effettuare il **trasporto del contenitore** verso lo slot assegnato.
6. Cargoservice rimane in attesa del segnale di fine trasporto. Comunica quindi se il trasporto è andato a buon fine oppure no.

   Se si ha un esito negativo, la richiesta viene rifiutata.

   Altrimenti, si aggiornano i dati del peso della stiva e torna in attesa di una richiesta.

Attenzione: non sarà cargoservice a gestire le anomalie. Dato che molte delle anomalie comportano l'arresto del DDRobot, si è pensato di delegare questa responsabilità al cargorobot.

Il componente Cargorobot:

Come già anticipato, cargorobot è il componente responsabile della gestione delle attività del DDRobot all'interno della stiva.

Funzionamento generale

Cargorobot opera in risposta a un comando proveniente da cargoservice, che indica la necessità di spostare un contenitore presente all'IOPort verso uno slot specifico nella stiva.

Le azioni principali sono:

1. Ricezione del comando da parte di cargoservice con l'indicazione dello slot di destinazione.
2. Coordinamento con il basicrobot per il raggiungimento dell'IOPort.
3. Coordinamento con il basicrobot per il trasporto del contenitore allo slot indicato.
4. Coordinamento con il basicrobot per il ritorno alla posizione di default (HOME).

Nota: Cargorobot procede a ciascuna fase solo dopo aver ricevuto conferma dell'esecuzione corretta della precedente.
In caso di **fallimento**, la richiesta viene rifiutata (ad esempio per presenza di ostacoli).
In caso di **anomalie**, basicrobot viene messo in pausa finché non si ripristina una condizione operativa stabile.

Ruolo logico di Cargorobot

Si è deciso di non delegare a cargoservice la logica di navigazione nella stiva (es. individuazione slot, gestione percorso), per evitare un accoppiamento eccessivo tra astrazione e dettagli implementativi. Pertanto, cargorobot ha visibilità della **mappa della stiva** e dello **stato del robot**, così da poter pianificare i comandi verso basicrobot.

Movimenti disponibili del DDRobot

Le azioni fondamentali che basicrobot può compiere sono:

- **Avanzare** per una certa durata.
- **Arretrare** per una certa durata.
- **Ruotare a destra** di 90°.
- **Ruotare a sinistra** di 90°.
- **Eseguire uno "step"**, ovvero un avanzamento lungo la propria direzione pari alla lunghezza del robot.

MAPPA della stiva:

grazie allo step, si è fatta esplorare la stiva al robot in modo completo, ottenendo le sue dimensioni e il posizionamento degli elementi per il robot.

H	L	L	L	L	L	L
L	L	S1	S2	L	L	L
L	L	L	L	S5	L	L
L	L	S3	S4	L	L	L
L	L	L	L	L	L	L
P	O	O	O	O	O	O

Legenda:

- H: posizione iniziale (HOME)
- P: IOPort
- S1–S5: slot
- L: cella libera
- O: ostacolo

Resume di Cargorobot

Per consentire il ripristino in caso di interruzioni, cargorobot mantiene in memoria le **coordinate dello slot di destinazione**, inoltre, viene utilizzato il comando qak **ReturnFromInterrupt** per tornare allo stato precedente prima dell'interruzione.

Approfondimento del funzionamento del microservizio fornito basicrobot:

è un framework completo che gestisce movimento, pianificazione del percorso, ingaggio e percezione sonar, che nel cargoservice è esposto come attore Qak che interagisce con una serie di messaggi (request, reply, dispatch ed eventi) tramite protocollo MQTT.

Di seguito si riporta un elenco dei messaggi che basicrobot può ricevere e inviare.

Basicrobot può ricevere:

Tipo	Messaggio	Descrizione breve
request	engage (owner, stepTime)	Inizia l'ingaggio esclusivo per un controller, ovvero, gli si dà il permesso di controllare il robot
request	disengage(owner)	Rilascia l'ingaggio
request	step(duration)	Richiede un singolo passo
request	doplan(plan, stepTime)	Esegue un piano di movimento
request	moverobot(targetX, targetY)	Movimento diretto a coordinate assolute
request	getrobotstate()	Richiede stato attuale (posizione, direzione)
request	getenvmap(format)	Richiede la mappa ambientale
request	setrobotstate(x, y, dir)	Imposta manualmente lo stato (solo debug)
request	setdirection(dir)	Imposta direzione senza muoversi
dispatch	cmd(command)	Comando diretto sul robot (es: "w", "a", "h")

Basicrobot può inviare:

Tipo	Messaggio	Descrizione breve
reply	engagedone(owner)	Conferma avvenuto ingaggio
reply	engagerefused(owner)	Rifiuto dell'ingaggio (già occupato)
reply	stepdone(duration)	Conclusione positiva di un passo
reply	stepfailed(duration, cause)	Fallimento di un passo (es: ostacolo)
reply	doplandone(plan)	Piano eseguito correttamente
reply	doplanfailed(done, remaining)	Fallimento durante l'esecuzione del piano
reply	moverobotok(...)	Completamento corretto del moverobot
reply	moverobotfailed(done, remaining)	Fallimento nel moverobot
reply	robotstate(x, y, dir)	Stato del robot (risposta a getrobotstate)
reply	envmap(map)	Mappa ambientale (risposta a getenvmap)
event	sonar(distance)	Distanza rilevata dal sonar
event	alarm(reason)	Allarmi (es: disingaggio, errore, interruzione)

La gestione di anomalie hardware:

In seguito all'approfondimento riguardante il basicrobot, il team ha osservato che in realtà vi possono essere diversi tipi di anomalie che costringono il sistema a fermare il robot fino alla riparazione di esse:

dallo studio dei requisiti	il sonar all'IOPort può non rispondere correttamente o rilevare valori errati.
Da basicrobot	Ingaggio rifiutato
Da basicrobot	Fallimento del movimento del robot a un punto preciso

Da basicrobot	Allarme dovuto a disingaggio, errore o interruzione
Da basicrobot	Fallimento piano di movimento (doplanfailed)
Da basicrobot	Fallimento esecuzione step (stepfailed)

Nel caso dell'ingaggio rifiutato, è previsto che il sistema ritenti fino ad avere il permesso concesso.

È previsto il disingaggio solo in caso di spegnimento del sistema.

Per motivi implementativi, soprattutto per mantenere il livello di astrazione di cargorobot al di sopra rispetto a quella del basicrobot, si è preferito utilizzare 'moverobot'.

Per tutte le anomalie dovute ad allarmi del sonar, è previsto che il basicrobot si arresti fino alla risoluzione e riprenda il suo percorso.

Nel caso in cui sia proprio lo spostamento a fallire, il sistema considererà la richiesta insoddisfacibile e pertanto la rifiuterà.

3. Progettazione (link al folder del progetto):

Come accennato precedentemente in questo documento, sono stati creati dei componenti fittizi in grado di simulare il comportamento atteso di elementi che verranno sviluppati successivamente.

MOCK:

Per simulare il comportamento del sonarservice e della webgui sono stati aggiunti degli attori mock in qak.

sonar_mock: emette l'evento productDetected (trova il prodotto) o anomalyDetected(con il 25 % di probabilità)

```
QActor sonar_mock context ctx_sonarservice{
    [#import kotlin.random.Random #]
    State s0 initial{
        println("$name STARTING")
    }Goto waitForProduct

    State waitForProduct{
        [#  val randomValue = (0..100).random()     #]
        if[# randomValue/1 >= 75 #]{
            emit productDetected : productDetected(T)
        }
        else{
            emit anomalyDetected : anomalyDetected(T)
        }
    }Goto waitForProduct

}
```

Eventi emessi dal sonar

productDetected : productDetected(T)
anomalyDetected : anomalyDetected(T)
anomalyFixed : anomalyFixed(T)

webguimock: riceve il dispatch *update*

```
QActor webguimock context ctx_cargoservice{
    State s0 initial{
        println("$name STARTING")
    }Goto waitforupdate

    State waitforupdate{
        println("waiting for update of gui...")
    }Transition t0 whenMsg update -> updategui

    State updategui{
            println("updating webgui...")
    }Goto waitforupdate
}
```

Progettazione di cargoservice:

Sono stati dapprima definiti il contesto ctx_cargoservice e i messaggi scambiati da cargoservice con altri componenti esterni:

Da… a…	tipo	Messaggio
Companysimulator -> CargoService	Request	loadrequest : loadrequest(PID)
CargoService -> ProductService	Request	getweight: getweight(PID)
CargoService -> CargoService	Dispatch	accepted : accepted(PID, Weight, Slot)
CargoService -> CargoService	Dispatch	refused : refused(PID, Weight)
CargoService-> cargorobot	Dispatch	command : command(C)

Sono inoltre risultati necessari due POJO per la rappresentazione logica degli Slot in memoria.

POJO:

Per semplificare la progettazione del sistema e separare la logica applicativa dalla gestione dei dati, sono stati introdotti dei **POJO (Plain Old Java Object)** che modellano il comportamento degli **slot di carico** e della loro **gestione aggregata**.

In particolare:

- La classe Slot rappresenta un singolo scomparto in cui è possibile caricare un prodotto, con informazioni su **posizione**, **numero di spazi disponibili** e **stato di disponibilità**.
- La classe Slots funge da contenitore e gestore dell'intera collezione di slot, offrendo funzionalità per **recuperare uno slot**, **verificarne la disponibilità** e **registrare nuovi carichi**.

L'uso di questi POJO consente di astrarre e incapsulare la logica di gestione degli slot in componenti riutilizzabili e facilmente testabili, migliorando la chiarezza e la manutenibilità del codice.

POJO Slot:

Questa classe rappresenta una singola unità di slot (es. uno scomparto di un distributore automatico). Ogni slot ha un identificatore, una posizione, un numero di spazi disponibili e uno stato che indica se è disponibile o meno.

Campi principali:

- id: Identificatore univoco dello slot.
- numberOfSpaces: Numero di spazi disponibili all'interno dello slot.
- avaiable: Stato di disponibilità dello slot.
- positionx, positiony: Coordinate della posizione dello slot.

Logica:

- Lo slot è inizialmente disponibile, tranne se l'id è uguale a 5.
- Quando uno spazio viene occupato (occupySpace()), il numero di spazi disponibili si riduce. Se arriva a 0, lo slot diventa non disponibile.

Metodi utili:

- getId(), getX(), getY(): Accesso ai dati base.
- isAvaiable(): Verifica se lo slot è disponibile.
- getPosition(): Restituisce la posizione in formato stringa.
- toString(): Ritorna l'id come stringa.

POJO Slots:

Questa classe gestisce una collezione di oggetti Slot, inizializzati in base a un numero fornito. Funziona da contenitore e gestore logico della disponibilità degli slot.

Campi principali:

- slotList: Lista di oggetti Slot.

Logica:

- Gli slot vengono creati con coordinate predefinite in base all'id (da 1 a 5).
- Fornisce metodi per cercare slot per id, verificare disponibilità e registrare un prodotto in uno slot.

Metodi utili:

- getSlotById(int id): Restituisce lo slot con l'id specificato.
- getSlotPositionById(int id): Restituisce la posizione dello slot.
- getAvaialbeSlot(): Restituisce l'id del primo slot disponibile.
- registerProductInSlot(int id/Slot): Occupa uno spazio nello slot specificato.

Costanti aggiunte:

in cargoservice

MaxLoad	Peso massimo totale che può essere caricato nella stiva – **fissato a 1000**.
slotSpaces	Numero massimo di prodotti caricabili in ciascuno slot – **fissato a 5**.
slots	Oggetto di tipo Slots, rappresenta la collezione di slot disponibili nel sistema.

In cargorobot:

stepTime	Tempo di passo del robot, utilizzato nell'engage con basicrobot – es. **350ms**.
homeX, homeY	Coordinate della posizione di default "home" del robot.
ioX, ioY	Coordinate della porta di ingresso/uscita (IOPort) per il carico/scarico dei prodotti.

Variabili aggiunte:

in cargoservice

currentHoldWeight	Peso complessivo attualmente caricato nel sistema. Aumenta ad ogni carico completato.

currentSlot	ID dello slot attualmente selezionato per il carico in corso.
currentPID	ID del prodotto attualmente richiesto da caricare.
currentWeight	Peso del prodotto corrente da caricare, ricevuto dal productservice.

In cargorobot

destSlotX, destSlotY	Coordinate dello slot di destinazione in cui va caricato il prodotto.
currentDestX, currentDestY	Coordinate della prossima destinazione che il robot deve raggiungere (IOPort, slot o home).

Modelli eseguibili ottenuti:

Cargoservice:

```
QActor cargoservice context ctx_cargoservice{
  [#
    import main.java.Slots
    val MaxLoad = 1000
    val slotSpaces = 5
    val slots: Slots
    var currentHoldWeight = 0

    var currentSlot = -1
    var currentPID = -1
    var currentWeight = -1

  #]
  State s0 initial{
    println("$name STARTS")
    [#
      slots = new Slots(5,slotSpaces) //mockup
    #]
  }
  Goto waitrequest

  State waitrequest{
    println("WAITING FOR LOAD REQUEST...")
  }
  Transition t0 whenRequest loadrequest -> getweight

  State getweight{
    printCurrentMessage
    onMsg(loadrequest: loadrequest(PID)){
      [#
        var PID = payloadArg(0).toInt()

      #]
    }
    request productservice -m getweight : getweight($PID)
  }
  Transition t1
  whenReply returnweight -> validateRequest
  whenReply productnotexistent -> managerefusal
```

```
  State getweight{
    printCurrentMessage
    onMsg(loadrequest: loadrequest(PID)){
      [#
        var PID = payloadArg(0).toInt()

      #]
    }
    request productservice -m getweight : getweight($PID)
  }
  Transition t1
  whenReply returnweight -> validateRequest
  whenReply productnotexistent -> managerefusal


  State validateRequest{
    printCurrentMessage
    onMsg(returnweight : returnweight(PID, Weight)){
      [#
        val PID = payloadArg(0).toInt()
        val Weight = payloadArg(1).toInt()
        val Slot = slots.getAvaiableSlot()
        val canLoad = (currentHoldWeight + Weight) <= MaxLoad && Slot != -1
      #]
      if[#canLoad#]{
        [#
          currentPID = PID
          currentWeight = Weight
          currentSlot = Slot
        #]
        forward webguimock -m update : update("to load: $currentSlot")
        autodispatch accepted : accepted($PID, $Weight, $Slot)
      }
      else{
        autodispatch refused : refused($PID, $Weight)
      }

    }
  }
  Transition t2
  whenMsg accepted -> waitForProduct
  whenMsg refused -> managerefusal

  State managerefusal{
    println("Request refused. Back to wait.")
  }Goto waitrequest


  State waitForProduct{
    println("REQUEST ACCEPTED. Waiting for product on IOPort...")
  }
  Transition t3
  whenEvent productDetected -> serveloadrequest

  State serveloadrequest{
    println("Product detected. Moving robot...")
    [#
      val destination = slots.getSlotPositionById(currentSlot)
    #]
    forward cargorobot -m command : command("move to $destination")
  }Goto waitendofrequest

  State waitendofrequest{
    println("Waiting for robot to finish its task...")
    delay 300
  }
  Transition t4
  whenEvent finishedtransport -> lastoperations

  State lastoperations{
    printCurrentMessage
    onMsg(finishedtransport : finishedtransport(T)){
      [# val msg = payloadArg(0).toString()#]
      if[# msg=="failure" #]{
        println("There was a fatal error with the load. Load request rejected")

      }
      else{
        [#
          slots.registerProductInSlot(currentSlot)
            currentHoldWeight = currentHoldWeight + currentWeight
        #]
        println("product loaded successfully...")
        forward webguimock -m update : update("loaded to $currentSlot")
      }
      [#
        currentSlot = -1
        currentPID = -1
        currentWeight = -1
      #]
    }

  }Goto waitrequest


}
```

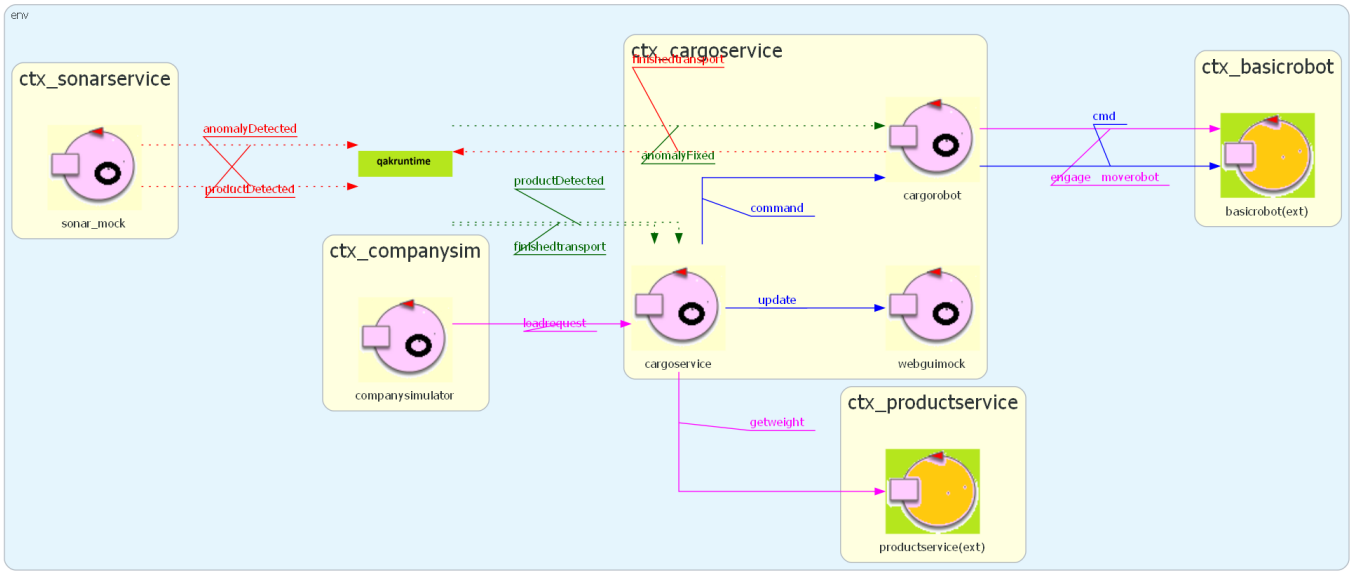
Cargorobot:

```
QActor cargorobot context ctx_cargoservice {
  [#
    val stepTime = 350
    val homeX = 0
    val homeY = 0
    val ioX = 5
    val ioY = 0

    var destSlotX = 0
    var destSlotY = 0
    var currentDestX = 0
    var currentDestY = 0


  #]

  State s0 initial {
    println("$name RUNNING")
  }Goto engageRobot

  State engageRobot {
    println("Engaging basic-robot...")
    request basicrobot -m engage : engage(name, stepTime)
  }
  Transition t0
  whenReply engagedone -> waitForCommand
  whenReply engagerefused -> retryEngage

  State retryEngage {
    println("Engage refused, retrying...")
    delay 500
  }Goto engageRobot

  State waitForCommand {
    println("Waiting for command")
  }
  Transition t1 whenMsg command -> prepareDelivery

  State prepareDelivery {
    printCurrentMessage
    onMsg(command : command(C)) {
      [#
        val coords = C.split(" ").last().replace("(", "").replace(")", "").split(",")
        destSlotX = coords(0).toInt()
        destSlotY = coords(1).toInt()

        phase = "pickup"

      #]
    }
  }Goto pickup

  State pickup{
    println("Going to IOPort to pick up product...")
    [#
          currentDestX = ioX
          currentDestY = ioY
        #]
        request basicrobot -m moverobot : moverobot(currentDestX, currentDestY)
  }
  Transition t2
  whenReply moverobotdone -> delivery
  whenReply moverobotfailed -> handleFailure
  whenInterruptEvent alarm -> handleAnomaly
  whenInterruptEvent anomalyDetected -> handleAnomaly

  State delivery{
    println("Delivering product to slot...")
          [#
            currentDestX = destSlotX
            currentDestY = destSlotY
          #]
          request basicrobot -m moverobot : moverobot(currentDestX, currentDestY)
  }
  Transition t3
  whenReply moverobotdone -> return
  whenReply moverobotfailed -> handleFailure
  whenInterruptEvent alarm -> handleAnomaly
  whenInterruptEvent anomalyDetected -> handleAnomaly
```

```
  State return{
    println("Returning to home position...")
          [#
            currentDestX = homeX
            currentDestY = homeY
          #]
          request basicrobot -m moverobot : moverobot(currentDestX, currentDestY)
  }
  Transition t4
  whenReply moverobotdone -> endOfTask
  whenReply moverobotfailed -> handleFailure
  whenInterruptEvent alarm -> handleAnomaly
  whenInterruptEvent anomalyDetected -> handleAnomaly

  State endOfTask{
    println("Transport complete. Emitting finishedtransport.")
        emit finishedtransport : finishedtransport(ok)
  }
  Transition t5
  whenReply moverobotfailed -> handleFailure
  whenInterruptEvent alarm -> handleAnomaly
  whenInterruptEvent anomalyDetected -> handleAnomaly


  State handleFailure {
    println("Robot move failed. Check system status.") color red
    emit finishedtransport : finishedtransport(failure)
  }Goto waitForCommand

  State handleAnomaly {
    println("Anomaly detected. Stopping robot...")
    forward basicrobot -m cmd : cmd(h)
  }Goto waitFix

  State waitFix {
    println("Waiting for anomaly to be fixed...")
    delay 200
  }
  Transition t5 whenEvent anomalyFixed  -> resuming

  State resuming{
    request basicrobot -m moverobot : moverobot(currentDestX, currentDestY)
    returnFromInterrupt
  }

}
```

Schema dell'architettura al termine dello sprint 1:

Di seguito si riporta lo schema complessivo dell'architettura generato automaticamente da Qak. Esso rappresenta le interazioni tra i principali attori del sistema, in particolare cargoservice, cargorobot e il microservizio esterno basicrobot, oltre agli elementi simulati (sonar_mock, companysimulator, webguimock) e al productservice.
Le connessioni e i messaggi scambiati riflettono fedelmente le funzionalità descritte nella sezione precedente.



cargoservice_overviewArch

Si fornisce quindi un PIANO DI TESTING:

Si ritiene necessario controllare i flussi di esecuzione principali di cargoservice:

1) Caso in cui la richiesta di carico viene accettata:

```
@Test
public void testLoadRequestAccepted() throws Exception { //usiamo un pid che è presente nel DB
    String requestStr = CommUtils.buildRequest("tester","loadrequest", "loadrequest(10)", "cargoservice").toString();
    System.out.println("Richiesta: " + requestStr);
    String response = conn.request(requestStr);
    System.out.println("Risposta: " + response);
    assertTrue("TEST: richiesta accettata", response.contains("accepted"));
}
```

2) Caso in cui la richiesta viene rifiutata dal product service:

```
@Test
public void testLoadRequestRejectedByProductService() throws Exception {//usiamo un pid che non è presente nel DB
    // PID inesistente (es: 999)
    String requestStr = CommUtils.buildRequest("tester", "loadrequest", "loadrequest(-999)", "cargoservice").toString();
    System.out.println("Richiesta: " + requestStr);
    String response = conn.request(requestStr);
    System.out.println("Risposta: " + response);

    assertTrue("TEST: richiesta rifiutata per PID inesistente", response.contains("refused"));
}
```

3) Caso in cui la richiesta viene rifiutata per il peso troppo elevato:

```
@Test
public void testLoadRequestRejectedForWeight() throws Exception {
    // PID troppo pesante (es: 3 = 1100)
    String requestStr = CommUtils.buildRequest("tester", "loadrequest", "loadrequest(3)", "cargoservice").toString();
    System.out.println("Richiesta: " + requestStr);
    String response = conn.request(requestStr);
    System.out.println("Risposta: " + response);

    assertTrue("TEST: richiesta rifiutata per peso eccessivo",
        response.contains("refused"));
}
```

4) Caso in cui la richiesta venga rifiutata per mancanza di spazio negli slot:

```java
@Test
public void testLoadRequestRejectedForSlot() throws Exception {
    String requestStr = CommUtils.buildRequest(
        "tester",
        "loadrequest",
        "loadrequest(20)",      // PID 20 che dovrebbe far scattare il rifiuto per slot occupati
        "cargoservice"
    ).toString();

    System.out.println("Richiesta: " + requestStr);

    String response = conn.request(requestStr);

    System.out.println("Risposta: " + response);

    // Controlla che la risposta contenga 'refused' (come da tuo Oak)
    assertTrue("TEST: richiesta rifiutata per slot pieni",
        response.contains("refused"));
}
```

E di cargorobot:

1) Caso di una richiesta di consegna generica che vada a buon fine:

```java
@Test
public void testDeliveryCommand() throws Exception {
    String cmdMsg = CommUtils.buildDispatch("tester", "command", "command(0,0)", "cargorobot").toString();
    System.out.println("Invio comando: " + cmdMsg);
    conn.forward(cmdMsg);

    Thread.sleep(1000);

    assertTrue(true);
}
```

2) Caso di un'anomalia sonar:

```java
@Test
public void testAnomalyDetected() throws Exception {
    // Simulo evento anomalyDetected
    String anomalyMsg = CommUtils.buildEvent("sonar_mock", "anomalyDetected", "anomalyDetected(T)").toString();
    System.out.println("Invio evento anomalia: " + anomalyMsg);
    conn.forward(anomalyMsg);

    Thread.sleep(500);

    // Ora simulo evento anomalyFixed dopo ritardo
    String fixedMsg = CommUtils.buildEvent("sonar_mock", "anomalyFixed", "anomalyFixed(T)").toString();
    System.out.println("Invio evento anomalia fissata: " + fixedMsg);
    conn.forward(fixedMsg);

    Thread.sleep(500);

    assertTrue(true); // placeholder per verifiche
}
```

3) Caso di un fallimento di consegna:

```java
@Test
public void testMoveFailed() throws Exception {
    String failMsg = CommUtils.buildReply("basicrobot", "moverobotfailed", "moverobotfailed(T)", "cargorobot").toString();
    System.out.println("Invio moverobotfailed: " + failMsg);
    conn.forward(failMsg);

    Thread.sleep(500);

    assertTrue(true);
}
```