Sprint 1

Sommario

Obiettivi

Obiettivo dello sprint 1 è quello, partendo dall'architettura e dalle considerazioni fatte nello sprint 0, di analizzare, progettare e pianificare il deployment della parte del sistema designata al fine di ottenere un prototipo funzionante da poter mostrare al cliente. Nello specifico, sulla base delle priorità dei requisiti e dato il fatto che il componente che implementa il productservice è già fornito, lo sprint si concentrerà sullo sviluppo del software riguardante il cargorobot e il cargoservice.

Sottoinsieme di requisiti considerati

Come già enunciato nello sprint0 in questo sprint1 ci siamo concentrati nel sottoinsieme di requisiti relativi al core-business dell'applicazione. In particolare, facciamo riferimento a quelli ad alta priorità, necessari per il funzionamento del sistema.

Bisogna considerare che i requisiti trattati in questo sprint si basano sull'esistenza di altri componenti del sistema, come *il sonar*, che però non sono ancora stati sviluppati. La loro realizzazione è infatti pianificata per gli sprint successivi.
Per questa ragione, nello sprint 1 verranno impiegati dei componenti ***mock*** che replicheranno per i test il comportamento di quelli mancanti.

Di seguito vengono riportati i requisiti relativi a questo sprint in cui come già detto verranno sviluppate le funzionalità riguardanti il **productservice** (in realtà già fornito), il **cargoservice** e il **cargorobot.**

Requisiti productservice

- Un prodotto è caratterizzato da un PID, un nome e un peso
- Dovrà essere possibile registrare un prodotto specificando nome e peso
- A seguito della registrazione dovrà essere assegnato un PID a tale prodotto. Tale PID dovrà essere ritornato a chi ha chiesto la registrazione
- Il prodotto registrato dovrà essere memorizzato su un database

Requisiti cargoservice

- Deve poter accettare le richieste di carico di un container di prodotti
- Deve poter rifiutare la richiesta se tutti i 4 slot sono occupati oppure se con l'aggiunta del prodotto il peso totale della stiva superasse il parametro MaxLoad
- Deve essere accettata in tutti gli altri casi
- In caso di richiesta accettata:

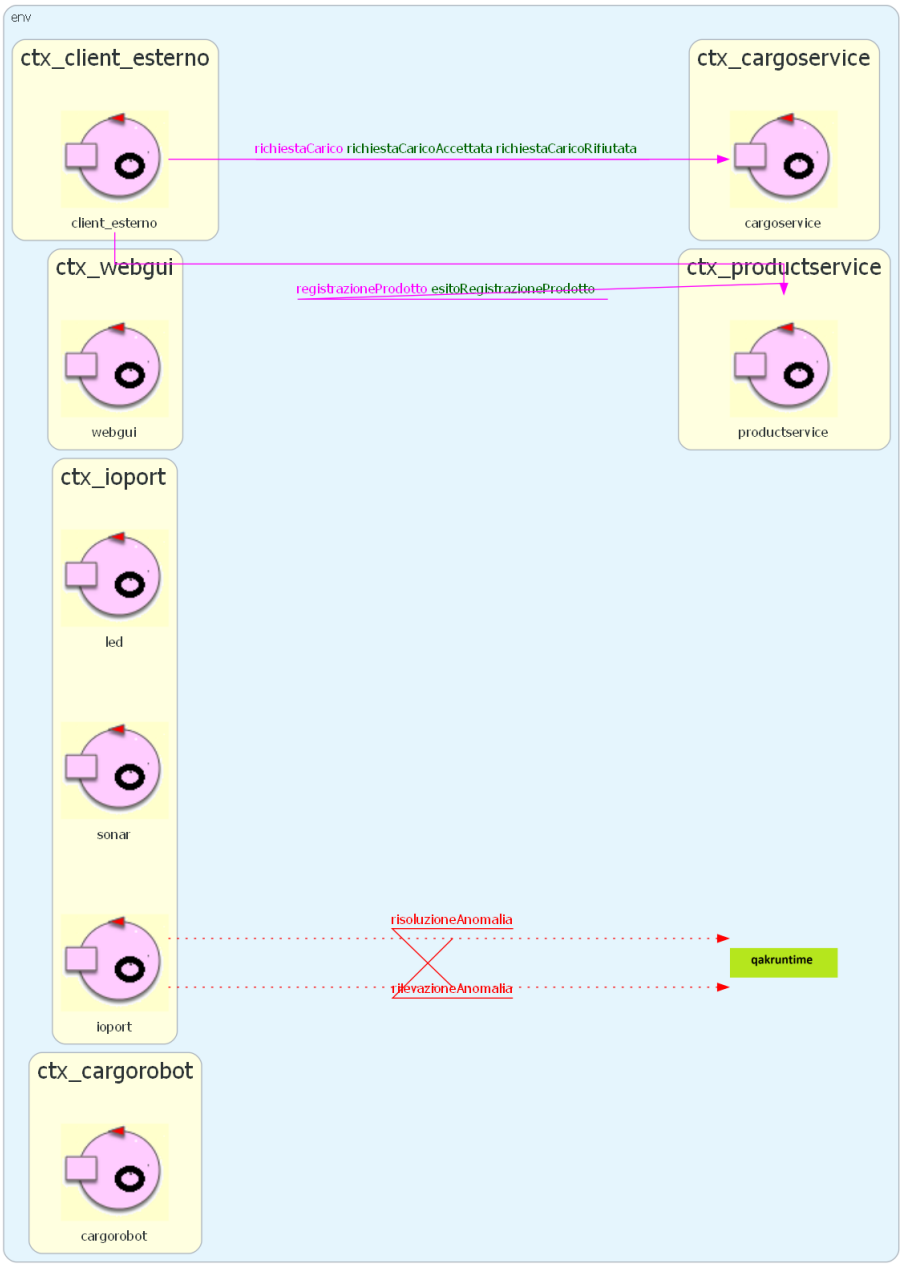
    - Deve associare il PID del prodotto a uno slot e ritornare il nome dello slot
    - Da questo momento non elabora altre richieste di carico
    - Attende la presenza del container all'IOPort
    - Si assicura che il container venga riposto nello slot corretto dal cargorobot
    - Una volta terminato il caricamento del container, il cargoservice può nuovamente elaborare le richieste di carico
- Deve poter informare e aggiornare una web-gui sullo stato della stiva
- Deve interrompere ogni attività in caso venga rilevata un'anomalia

Requisiti cargorobot

- Il cargorobot ha un duplice ruolo:
    - Come stabilito nello sprint 0, è responsabile delle attività del DDR all'interno del deposito.
    - *C*omunica con *basicrobot*, che a sua volta interagisce con l'ambiente virtuale *WEnv*, per eseguire gli interventi di carico richiesti da *cargoservice*.

- Sequenza di attività:
    - Riceve da *cargoservice* una richiesta di gestione di un container con indicazione dello slot riservato.
    - Si dirige verso la **IO-port.**
    - Recupera il container,
    - Si posiziona nello slot specificato.
    - Deposita il container nello slot.
    - Una volta terminato il deposito, ritornare alla **home** e può accettare nuove richieste di caricamento

Architettura dello sprint 0



cargorobotArch

Analisi del problema

È evidente che le due entità **productservice** e **cargoservice** già descritte nello sprint 0 svolgano compiti diversi e siano completamente autonome. Questo permette di modellarle attraverso due bounded context differenti. Questi interagiranno tra loro e con gli altri componenti i cui contesti verranno specificati nei successivi sprint.

L'obiettivo dell'analisi del problema qui descritta è quello di chiarire e proporre le possibili alternative per la modellazione e la realizzazione dei seguenti punti non descritti esplicitamente dai requisiti

- Sono necessarie interazioni aggiuntive tra questi e altri componenti rispetto a quelle descritte nello sprint 0?
- Chi farà le richieste al productservice? Come e attraverso quale tipo di interfaccia queste richieste saranno fatte?
- Le richieste di carico verranno innescate dalla presenza di un container davanti all'IOPort? Se si queste richieste verranno direttamente dall'IOPort o da un'altra entità? E nel caso, come farà il cargoservice a conoscere il PID del prodotto da caricare? Se no in che modo potranno essere effettuate tali richieste?
- Per conoscere i dati (nome e peso) del prodotto da caricare, il cargoservice dovrà fare una richiesta al productservice?
- Come gestire la memorizzazione da parte del productservice su un database?
- Come manterrà il cargoservice le associazioni PID-slot?
- Come il cargoservice potrà comandare al cargorobot di trasportare il container nello slot corretto?
- Come il cargoservice verrà a conoscenza del termine dell'operazione?
- In che modo il cargoservice aggiornerà la web-gui?
- Come gestire le anomalie?

Analisi productservice

Come riportato nello sprint 0, productservice è fornito funzionante dal cliente mediante container docker e codice sorgente qak. Non è fornita nessuna documentazione, perciò analizzando il codice sorgente si può notare che:

Il servizio è raggiungibile mediante l'attore ***productservice*** che fa parte del seguente contesto

```
Context ctxcargoservice ip [ host="localhost" port=8111]
```

Eventualmente è possibile raggiungere il servizio anche localizzando il suo indirizzo presso un server eureka. In questa fase non si analizza nello specifico i dettagli già codificati di questa possibilità, eventualmente la possibilità di utilizzare eureka verrà discussa più avanti analizzando e nel caso modificando i parametri presenti nell'implementazione fornita.

Per comunicare con il productservice sono disponibili i seguenti messaggi qak:

```
Request createProduct : product(String)                    //String JSON '{"productId":31,"name":"p31","weight":311}'
Reply   createdProduct: productid(ID) for createProduct   //String JSON

Request deleteProduct  : product( ID )
Reply   deletedProduct : product(String) for deleteProduct

Request getProduct : product( ID )
Reply   getProductAnswer: product( JSonString ) for getProduct

Request getAllProducts : dummy( ID )
Reply   getAllProductsAnswer: products(  String ) for getAllProducts
```

Queste sono tutte richieste e relative risposte che possono essere inviate al servizio. Da notare come il commento indichi il formato che dovranno avere i dati al momento dello scambio: sarà necessario codificare le informazioni del prodotto da registrare in **formato json**, specificando ***productId***, ***name*** e ***weight***.

Presumibilmente questo sarà anche il formato della risposta alle richieste get per ottenere i prodotti (da verificare il formato dell'ID da inviare)

Nel file qak sono riportati anche altri messaggi: *cargoinfo* (Dispatch), *cargoevent* (Event) e *alarm* (Event). Si può notare come di questi *cargoinfo* e *alarm* non siano utilizzati (sono presenti solo in delle porzioni di codice commentato), mentre *cargoevent* è emesso solamente una volta durante l'operazione di eliminazione di un prodotto.

Alla luce di questa analisi è presumibile che gli ultimi tre eventi descritti possano essere trascurati al momento, mentre delle varie richieste, le due principali che possono essere immediatamente utili alle funzionalità richieste dai requisiti probabilmente sono ***getProduct/getProductAnswer*** e ***createProduct/createdProduct***. Le altre due richieste possono essere utili come funzionalità aggiuntive ma non sono richieste esplicitamente dai requisiti, potrebbero tuttavia tornare utili in successive anaisi.

Il **microservizio** *productservice* si deve interfacciare con un DB. Il cliente ha richiesto infatti un sistema di database per la gestione della persistenza dei prodotti registrati. A tal fine, abbiamo deciso di utilizzare **MongoDB**, un database NoSQL che rappresenta i dati sotto forma di documenti in formato JSON-like. Questa caratteristica risulta particolarmente utile per l'elaborazione dei dati all'interno del nostro sistema.

Analisi cargoservice

Dai requisiti il cargoservice si figura come un'entità che dovrà svolgere le seguenti operazioni:

- Accettare **richieste di carico** e dare una **risposta** in base allo **stato della stiva**
- **Comandare al cargorobot** il caricamento da compiere
- **Non elaborare richieste** fintanto che il cargorobot non avrà **riposto il container** nello slot giusto

- **Interrompere le operazioni** durante un malfunzionamento
- **Aggiornare la web-gui**

In sintesi, si può considerare cargoservice come l'orchestratore del sistema.

Di seguito verranno analizzate le possibilità di realizzazione di ogni funzionalità al fine di poter creare un'architettura logica completa del cargoservice.

Richieste di carico e interazione con productservice: come già evidenziato dai requisiti il cargoservice dovrà accettare dei messaggi *richiestaCarico* e rispondere in base allo stato della stiva con *richiestaCaricoAccettata* o *richiestaCaricoRifiutata*.

La richiesta può essere rifiutata nel caso in cui gli slot siano tutti occupati o nel caso in cui il peso corrente della stiva sommato a quello del prodotto il cui caricamento è stato richiesto ecceda *MaxLoad*. Posto il fatto che cargoservice dovrà mantenere lo stato corrente della stiva (come verrà analizzato successivamente), l'unico dato necessario che non conoscerà immediatamente sarà quello del peso del prodotto da caricare.

Perciò, a seguito della richiesta, il cargoservice dovrà recuperare i dati del prodotto da caricare (partendo dal PID richiesto) dal productservice tramite i messaggi ***getProduct/getProductAnswer*** (offerti dall'implementazione di productservice fornita dal committente). In questo modo, facendo il parsing della stringa json ricevuta, potrà ottenere tutti i dati necessari relativi al prodotto da caricare.

L'unica possibilità semplice e sensata è quella appena citata, non si vedono altre soluzioni che possano offrire vantaggi particolari

Stato della stiva: cargoservice dovrà necessariamente memorizzare lo stato della stiva. Nello specifico i dati di cui tenere traccia sono le associazioni slot-container (da cui è possibile ottenere informazioni sulla quantità di slot liberi) e il peso corrente della stiva. A questo punto si aprono più possibilità.

In primo luogo, bisogna scegliere se mantenere questi dati in maniera **volatile** (attraverso un qualche tipo di struttura dati come ad esempio una mappa) o **persistente** (mediante l'utilizzo di un database relazionale o non relazionale). La scelta deve considerare alcuni fattori: la possibilità di prevedere un riavvio del sistema durante la fase di caricamento della stiva, il fatto che le operazioni di caricamento della stiva siano considerabili atomiche e se dovesse essere necessario mantenere la disposizione dei prodotti nella stiva come storico.

L'associazione slot-container è mappabile in modo naturale tramite una struttura dati di tipo tabellare. Una buona strategia potrebbe essere quella di memorizzare, oltre all'associazione slot-PID, anche il peso del prodotto disposto su quello slot. In questo modo sarà possibile ricavare il peso complessivo della stiva senza necessità di mantenerlo aggiornato in una seconda struttura dati, evitando che le informazioni siano

separate e possibili errori sul calcolo e l'aggiornamento del peso ogni volta (a fronte di un modesto aumento in termini di memoria dovuto alla sola presenza di 4 slot)

Interazione con il cargorobot: il cargoservice deve avere il pieno controllo delle fasi di lavoro del cargorobot. Nello specifico dovrà poter indicare la presenza di un container all'IOPort per innescare il caricamento ed essere a conoscenza del termine di tale caricamento senza doversi preoccupare delle azioni concrete che il robot dovrà fare per eseguire l'operazione

Il posizionamento di un container davanti all'IOPort è un evento rilevabile da quest'ultima (analizzata nello specifico nello sprint successivo) che perciò emetterà come prevedibile un evento *containerRilevato*. Questo evento dovrà essere per forza rilevato dal cargoservice in quanto responsabile della gestione del processo di caricamento (infatti cargoservice è l'unico che può conoscere quale richiesta è stata fatta prima della rilevazione del container). L'evento sarà rilevante solamente in caso in cui prima ci sia stata una richiesta accettata, altrimenti non sarà possibile associare il container al prodotto.

Il cargoservice comanderà allora al cargorobot il caricamento della stiva tramite una richiesta *richiestaCaricamentoSlot,* in seguito alla quale esso inizierà compiere i movimenti per trasportare il container allo slot specificato. A quel punto il cargorobot risponderà alla richiesta con *slotCaricato* (o *caricamentoFallito* in caso di problemi), in questo modo cargoservice sarà a conoscenza del termine dell'operazione e potrà accettare una nuova richiesta.

Nel lasso di tempo che intercorre tra l'invio della risposta positiva *richiestaCaricoAccettata* e la ricezione della risposta alla richiesta *richiestaCaricamentoSlot,* cargoservice non deve processare altre richieste di carico.

Dall'incontro con il committente per la discussione dello sprint 0, si è venuti a conoscenza che le richieste debbano essere accodate; la gestione dell'accodamento è però di semplice realizzazione in quanto integrata direttamente nel linguaggio QAK.

Interruzione operazioni: **c**argoservice interrompe le sue operazioni quando il sonar rileva una distanza maggiore di DFREE e le riprende al momento in cui tornerà ad essere rilevata una distanza <= di DFREE.

Come già riportato nello sprint 0 è intuibile la presenza e la necessità di due eventi *rilevazioneAnomalia* e *risoluzioneAnomalia* generati dall'IOPort. Cargoservice dovrà essere sensibile a tali eventi e interrompere/riprendere le sue operazioni in corrispondenza di essi. Non si tratteranno oltre tali eventi in quanto inerenti all'IOPort che verrà trattata nello sprint successivo.

Aggiornamento web-gui: cargoservice deve permettere l'aggiornamento dinamico di una web-gui che mostri lo stato della stiva. Tale informazione può essere vista come

una risorsa dinamica che può essere aggiornata dal cargoservice e recuperata da un cliente esterno (es. la web-gui). Essa si mappa bene nella struttura dati che verrà utilizzata per memorizzare l'associazione slot-prodotti. A livello di realizzazione due possibilità valide per offrire un accesso alla risorsa potrebbero essere il protocollo CoAp o una REST API. Tutto ciò che riguarda la web-gui verrà trattato negli sprint successivi.

Analisi cargorobot

Come già menzionato, *cargorobot* è il componente incaricato di gestire le operazioni del **DDRobot** all'interno della stiva, il suo funzionamento generale è quello di entrare in azione sotto comando e controllo di *cargoservice*, che specifica le azioni concrete che esso deve compiere. Il cargorobot può essere visto come un attuatore del sistema comandato da cargoservice

Interazione con cargoservice: l'interazione cargorobot-cargoservice è già stata trattata durante l'analisi di cargoservice quindi non verrà riproposta nuovamente

Interazione con basicrobot: il cargorobot sfrutta il servizio basicrobot fornito dal committente per interfacciarsi con un certo tipo di robot. Nello specifico una volta ricevuta la richiesta di carico, esso sfrutterà la funzionalità di trovare un percorso tra due punti offerta dal basicrobot per fare i seguenti movimenti: HOME-IOPort, IOPort-slot, slot-HOME, pungoGenerico-IOPort. Il cargorobot sfrutterà anche la funzionalità di fermare il movimento del robot mandando l'evento *alarm* a cui il basicrobot è sensibile. Questa può risultare utile in due casi: nel caso della rilevazione/risoluzione di un'anomalia (per ottenere un comportamento analogo a quello descritto per il cargoservice) e per interrompere un movimento in caso arrivi una richiesta in un momento successivo al caricamento del prodotto all'IOPort ma precedente a quello del raggiungimento della HOME.

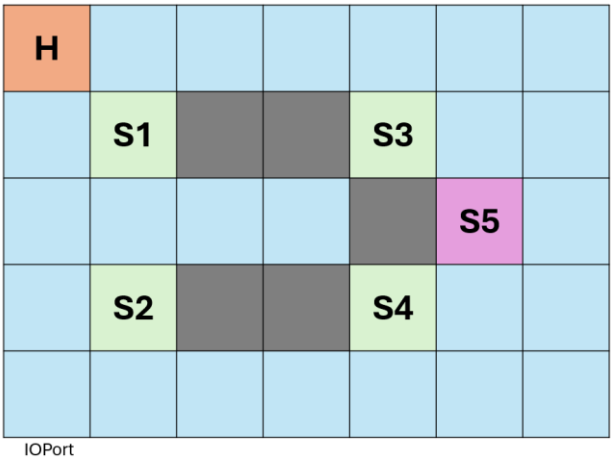
Modello della mappa: nel complesso, cargorobot utilizza basicrobot per agisce come un componente intelligente capace di pianificare la logica di navigazione all'interno della stiva. Il *cargoservice* si limita a comunicargli lo slot assegnato, mentre *cargorobot*, conoscendo soltanto la destinazione, calcola autonomamente il percorso più efficiente per:

- guidare il **DDRobot** fino all'**IO-Port** per il recupero del prodotto,
- trasportarlo allo slot indicato,
- riportare il robot alla posizione di default (**Home**).

Il basicrobot risulta fondamentale per queste operazioni in quanto fornisce la logica di navigazione e mantiene nativamente un modello sia della mappa della stiva sia della posizione del robot in essa.

Modello della mappa: In seguito all'analisi dei movimenti fondamentali abbiamo ritenuto di rilevante importanza riportare di seguito il modello della stiva dove si muoverà il robot, i punti di interesse sono:

- H (Home): angolo in alto a sinistra in cui il robot deve sostare quando è in stato di inattività.
- S {1,..,4} (Slots): aree designate in cui il robot deve depositare i container.
- S5: area permanentemente occupata in cui non possono essere lasciati i container.
- I/O Port: apertura situata nel bordo inferiore sinistro, da cui il robot preleva i container.



Utilizzando questo modello è quindi possibile legare ogni posizione a delle coordinate cartesiane nel seguente modo (x indica la riga e y la colonna):

- HOME = (0, 0)
- S1 = (1, 1)[1, 0]; S1 = (3, 1)[3, 0]; S1 = (1, 4)[1, 5]; S1 = (3, 4)[3, 5];
- IOPort = (4, 0)

NOTA: i valori tra parentesi quadre indicano le coordinate in cui il robot si dovrà fermare per effettuare il caricamento dello slot associato

Riepilogo messaggi

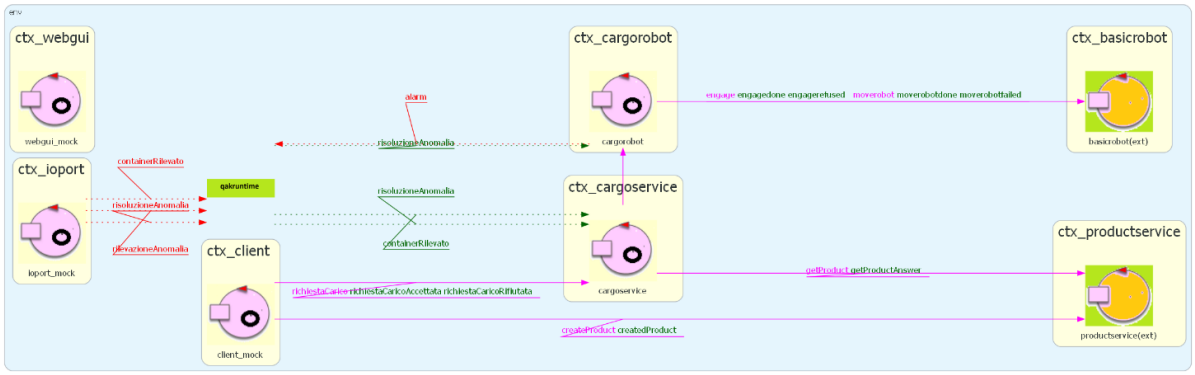
Riepilogo dei messaggi che si prevede di utilizzare:

Messaggio	**Tipo**	**Inviato da**	**Ricevuto da**	**Parametri**
RichiestaCarico	Request	*Cliente Esterno**	Cargoservice	PID
RichiestaCaricoAccettata	Reply (to Richiesta carico)	Cargoservice	*Cliente Esterno**	
RichiestaCaricoRifiutata	Reply (to Richiesta carico)	Cargoservice	*Cliente Esterno**	Motivo
RilevazioneAnomalia	Event	IOPort	Cargoservice Cargorobot	
RisoluzioneAnomalia	Event	IOPort	Cargoservice Cargorobot	
GetProduct	Request	Cargoservice	Productservice	PID
GetAnswere	Reply (to *GetProduct*)	Productservice	Cargoservice	Dati Prodotto
CreateProduct	Request	*Cliente Esterno**	Productservice	Dati Prodotto
CreatedProduct	Reply (to *CreateProduct*)	Productservice	*Cliente Esterno**	Esito Creazione
RichiestaCaricamentoSlot	Request	Cargoservice	Cargorobot	Slot
SlotCaricato	Reply (to *Richiesta CaricamentoSlot)*	Cargorobot	Cargoservice	
CaricamentoFallito	Reply (to *Richiesta CaricamentoSlot)*	Cargorobot	Cargoservice	
ContainerRilevato	Event	IOPort	Cargoservice	

Per brevità non vengono riportati i messaggi tra cargorobot e basicrobot in quanto si tratta semplicemente dei comandi relativi alle funzionalità offerte dal basicrobot.

L'entità denominata *Cliente Esterno** non è chiaramente definita. Il committente stesso ha informato che le richieste al cargoservice e al product service in termini di richiesta di caricamento e di registrazione di prodotti verranno fatte da altri entità non oggetto di questo progetto. Sarà quindi sufficiente definire l'interfaccia tramite la quale queste entità esterne potranno interagire con tali componenti

Nuova architettura logica



cargorobotArch

Progettazione

A seguito dell'analisi, data la natura ad attori/servizi dei due componenti analizzati, si è deciso di realizzare cargorobot e cargoservice come due microservizi. In questo modo si potranno ottenere numerosi vantaggi:

- I due servizi eseguiranno in maniera indipendente permettendo una maggiore separazione di responsabilità e un minore accoppiamento tra i due macrocomponenti
- In questo modo le interazioni sono facilmente realizzabili attraverso lo scambio di messaggi
- I due software che verranno realizzati potranno eseguire potenzialmente su più macchine diverse, permettendo sia i vantaggi legati all'utilizzo di un sistema distribuito, sia il dislocamento fisico, utile nel caso in cui ad esempiosi voglia eseguire il servizio del cargorobot sul DDR robot e il cargoservice su un server dedicato

Per fare ciò si è deciso di continuare a sfruttare il linguaggio QAK sia per questioni di tempi di sviluppo, sia per una più semplice integrazione con i sistemi già esistenti (basicrobot e productservice).

A seguito della scelta di QAK, viene naturale, per motivi di compatibilità con il DSL, scegliere i linguaggi Java/Kotlin per lo sviluppo di altri pezzi software che saranno necessari durante lo sviluppo (come oggetti POJO e test)

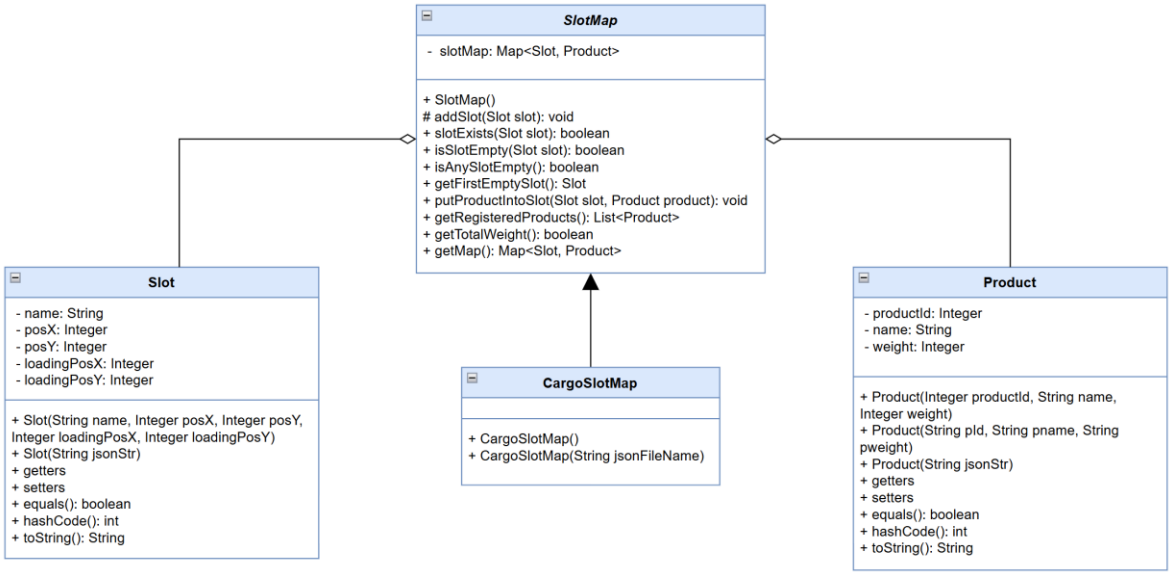
Come già anticipato, cargoservice e cargorobot possono essere modellati e progettati come attore, nello specifico come automi a stati finiti che interagiscono tramite messaggi. Fortunatamente il linguaggio QAK offre proprio queste caratteristiche permettendo di definire già ad alto livello la struttura specifica degli stati. Perciò, per la progettazione specifica dei dettagli del comportamento dei due servizi e delle interazioni tramite messaggi si rimanda direttamente al codice QAK che mostra in maniera schematica e diretta la struttura degli attori in gioco. Inoltre l'abstraction gap

colmato da QAK alleggerisce la progettazione di molti dettagli già gestiti dal runtime del DSL (come gestione dei thread o della comunicazione).

Di seguito verranno comunque analizzate alcune scelte progettuali ritenute rilevanti.

Classi POJO

Oltre agli attori QAK puri che verranno analizzati successivamente, per la realizzazione dei servizi saranno necessarie anche delle classi POJO che permetteranno di realizzare le strutture dati necessarie per il corretto funzionamento dei componenti. Di seguito è riportato un diagramma UML che mostra tali classi.



Product e Slot sono due semplici classi destinate a contenere i dati relativi alle due entità che modellano. Entrambe permettono di serializzare e deserializzare i dati in formato JSON (necessario per la comunicazione anche con i componenti già forniti), nello specifico Product è pienamente compatibile con productservice.

SlotMap è una classe astratta che gestisce l'associazione slot-prodotto dei container caricati in una stiva. CargoSlotMap invece, specializza la classe appena descritta per lo specifico caso della stiva della nave oggetto dei requisiti, preimpostando gli slot come descritti in precedenza o attraverso un file di configurazione in formato JSON e non

permettendone l'aggiunta di ulteriori. Questa scelta è legata a un'eventuale futura possibilità di riutilizzo della classe SlotMap in futuri nuovi contesti simili.

Di seguito è riportato un esempio di file di configurazione *slotmap-conf.json*

```
{
  "slots": [
    {
        "name": "Slot1",
        "posX": 1,
        "posY": 1,
        "loadingPosX": 1,
        "loadingPosY": 0
    },
    {
        "name": "Slot2",
        "posX": 3,
        "posY": 1,
        "loadingPosX": 3,
        "loadingPosY": 0
    },
    {
        "name": "Slot1",
        "posX": 1,
        "posY": 4,
        "loadingPosX": 1,
        "loadingPosY": 5
    },
    {
        "name": "Slot1",
        "posX": 3,
        "posY": 4,
        "loadingPosX": 3,
        "loadingPosY": 5
    }
  ]
}
```

Progettazione cargoservice

Gestione associazionei slot-prodotti: Partendo dalle classi POJO appena descritte è stato scelto di mantenere lo stato della stiva attraverso una mappa (nello specifico la classe SlotMap utiliza una HashMap) in modo volatile. In questo modo è possibile avere una gestione più semplice ed efficiente delle associazioni slot-prodotti in un contesto in cui l'operazione di carico della stiva avviene in un periodo contiguo di tempo. L'unico problema legato a un malfunzionamento della macchina su cui girerà cargoservice può essere risolto in altre maniere ripristinando lo stato a seguito di un riavvio automatico del servizio. La memorizzazione in manira persistente potrebbe essere un requisito aggiuntivo che il cliente potrebbe richiedere esplicitamente in futuro.

Evento rilevazione di un container sull'IOPort: La natura di tale evento è stata descritta durante l'analisi. Qui si vuole specificare come tale evento verrà percipito solamente da parte del cargoservice e non dal cargorobot. In questo modo, a fronte di una comunicazione aggiuntiva tra cargoservice e cargorobot (infatti il primo invierà *richiestaCaricamentoSlot* solo a fronte di una rilevazione del container al secondo), sarà possibile permettere a cargoservice di controllare la corretta esecuzione della

procedura: richiesta -> posizionamento sull'IOPort -> caricamento. In caso contrario il cargorobot potrebbe iniziare il caricamento anche nel caso in cui prima non ci sia stata una richiesta (tale avvenimento potrebbe essere evitato tramite appunto una comunicazione tra cargorobot e cargoservice, rendendo però la comunicazione inutilmente più complicata)

Progettazione cargorobot

Gestione del movimento del robot: Come già detto in fase di analisi i movimenti concreti del robot verranno realizzati direttamente dal componente basicrobot, il cargorobot si limiterà a indicare quali movimenti fare. Il basicrobot sposta il DDR robot da un punto di partenza a uno di destinazione un passo alla volta. Ogni passo è considerato dal basicrobot come una casella della mappa riportata in precedenza, ma la lunghezza di tale passo dovrà essere decisa dal cargorobot al momento della richiesta. Osservando attentamente la mappa della stiva, si è definito quindi come valore opportuno per un passo quello di 360 unità di tempo di movimento del robot. Tale valore è frutto di un'osservazione empirica dei movimenti del DDR robot.

Gestione degli allarmi: A seguito degli opportuni eventi da parte dell'IOPort il robot dovrà fermarsi e successivamente riprendere il suo lavoro dal punto in cui era arrivato. Ciò può essere realizzato in QAK gestendo l'evento come un interrupt, ma rimane comunque un punto aperto: bisogna distinguere il caso in cui il robot si blocca mentre è in movimento dal caso in cui è in una situazione di attesa. Nel secondo caso non c'è nessun problema rilevante. Nel primo invece, è necessario che il cargorobot interrompa il lavoro di basicrobot tramite l'opportuno messaggio *alarm*. In tal caso però sarà fondamentale mantenere in memoria le coordinate della destinazione per poterle riprendere in un secondo momento. Basicrobot infatti, ricevuto l'*alarm,* interrompe il percorso che stava facendo considerandolo come terminato (con un fallimento). Sarà allora essenziale, una volta risolto il malfunzionamento, chiedere nuovamente tale movimento al basicrobot.

Gestione della richiesta di carico prima di tornare alla HOME: Questo caso si presenta simile al precedente: sarà necessario interrompere tramite *alarm* il movimento alla HOME e successivamente ordinare quello verso l'IOPort.

Deployment

I due compoenenti software verranno rilasciati come due microservizi, ***cargoservice*** e ***cargorobot,*** attraverso due container docker. Questi due servizi avranno bisogno, oltre ai futuri servizi IOPort e web-gui (per ora realizzabili solamente come mock), anche di ***productservice*** e ***basicrobot*** ai quali si fa riferimento nel qak come ExternalActor.

Per caricare i container su docker:

1. Creare l'immagine docker (una solva volta oppure di nuovo quando cambia il progetto). Entrare nella cartella del progetto di cui si vuole fare l'immagine e fare:
   a. ***./gradlew distTar***
   b. ***docker build -t <nome>:1.0 .***

   dove <nome> è il nome del progetto, ovvero quello dopo System nella prima riga del qak.

2. Entrare nelle varie cartelle dove ci sono i file yaml e lanciare il comando docker-compose giusto

   docker-compose -f nomefile.yaml up

NOTA: cargorobot dovrà essere fatto partire dopo il basicrobot altrimenti potrebbe dare problemi durante la fase di ingaggio.

Per quanto riguarda il cargoservice è necessario impostare il parametro MAX_LOAD come variabile d'ambiente.

Piano di test

Per la realizzazione dei test è stato deciso di utilizzare java mediante classi main di test e test automatici JUnit. Questa scelta è stata fatta per avere un maggiore controllo sui messaggi e sull'ordine dei test, nonché per poter automatizzarli attraverso JUnit, anche se in maniera limitata per via della natura complessa ed eterogenea del sistema.

Per il corretto funzionamento del sistema nell'architettura logica erano riportati tre componenti mock: IOPortMock, webGuiMock e clientMock. Per via della natura dei messaggi inviati da IOPortMock e clientMock, per avere un maggiore controllo sull'ordine dei messaggi e sul loro funzionamento nei test, essi non sono stati realizzati in qak, ma ne sono stati simulati i messaggi direttamente in java attraverso la libreria unibo.basicomm23 fornita. WebGuiMock è invece stata trascurata in quanto non necessaria per il corretto funzionamento di cargorobot e servicerobot.

Per il productservice non abbiamo pensato ad un piano di test perché il componente è già stato fornito e quindi i vari test sono a carico del committente.

Abbiamo suddiviso il piano di test in base alla tipologia di verifica che si può fare sui vari componenti. Questo in quanto i test che prevedono la presenza del cargorobot necessitano di un riscontro visivo sui movimenti effettivi compiuti dal DDR robot che non è possibile ottenere attraverso test automatici. Attraverso i test automatici è invece stato possibile verificare la correttezza delle risposte del cargoservice alle richieste.

- **Test Non Automatici**: riguardano le parti del sistema che non sono testabili direttamente attraverso dei JUnit Test.
    - Simulare un allarme di anomalia e ci aspettiamo che cargoservice e robot si debbano interrompere e quando interrompiamo l'allarme devono ripartire.
    - Dare indicazioni manuali del robot per uno slot tra il momento in cui ha lasciato il product e il momento in cui torna alla home e verificare che non torni alla home completamente ma accetti la nuova richiesta.
    - Verificare che il cargorobot non accetti altre richieste mentre ne sta finendo un'altra
- **Test Automatici:**
    - **TestRichiestaCaricoOK**(): si occupa di testare se la richiesta di carico è andata a buon fine.
    - **TestRichiestaCaricoNoSlotLiberi**(): si occupa di testare il fatto che la richiesta non può essere accettata a causa degli slot pieni.
    - **TestRichiestaCaricoOltrePesoMax**(): si occupa di testare il fatto che la richiesta non può essere accettata perché è stato raggiunto il peso massimo nella stiva.
    - **TestRichiestaCaricoProdNonPres**(): si occupa di testare il fatto che la richiesta non può essere accettata perché il prodotto non è presente.

Conclusioni

Partendo dall'analisi iniziale effettuata durante lo Sprint0, questa iterazione si è concentrata sull'identificazione e sull'implementazione dei requisiti legati al **core business** dell'applicazione.

Sono stati implementati i servizi principali del business: **cargorobot**, che si occupa della creazione e della cancellazione dei prodotti, e **cargoservice**, che coordina il processo di caricamento verificando correttamente i vincoli di capacità. Tutte le attività pianificate sono state completate nei tempi stimati e i punti chiave previsti per lo Sprint1 sono stati raggiunti con successo.

Questo sprint ha quindi posto le basi per le fasi successive dello sviluppo, che prevedono:

- **Sprint2** – gestione dell'**ioport**
- **Sprint3** – sviluppo della **web-GUI**