

# chronicles\_sprint1\_v3

**Il team:** Diego Bruno, Marco Crisafulli, Sebastiano Giannitti

[Link repository Github](#)

## Introduction

Il presente documento illustra le attività di analisi, progettazione e implementazione svolte durante lo **Sprint 1**. Questa fase rappresenta il naturale proseguimento dell'analisi dei requisiti condotta nello Sprint 0 e si pone l'obiettivo primario di colmare l'**Abstraction Gap** identificato tra le specifiche funzionali e la loro realizzazione tecnica .

Per giungere a una realizzazione software robusta, è stato necessario intraprendere una fase preliminare di **Analisi del Problema**. Nello specifico, tale analisi è stata necessaria per formalizzare i componenti logici che, nella fase precedente, erano stati definiti solo a livello concettuale.

Coerentemente con il piano di sviluppo incrementale stabilito, il focus di questo sprint è limitato alla **Business Logic** del servizio **cargoservice**. La **Business Logic** comprende:

- Attore principale cargoservice che riceve la requestToLoad e la gestisce delegando ove necessario ad altri attori dei compiti
- Attore cargorobot che comunica con il basicrobot
- Attore productpolice che comunica con il productservice
- Attore hold che si occupa di gestire la stiva

Approfondiremo in questo documento le scelte fatte con i relativi motivi e conseguenze sul sistema.

Si precisa che l'analisi approfondita e l'implementazione dei sottosistemi di I/O fisico (Sonar, LED) e dell'interfaccia utente (Web-GUI) sono alle iterazioni successive .

Il documento procederà dunque descrivendo la **formalizzazione architetturale** dei componenti analizzati, l'**implementazione del flusso operativo** primario e, infine, la **strategia di testing** adottata per garantire la stabilità della base di codice.

## Requirements

*The company asks us to build a software systems (named **cargoservice**) that:*

1. *is able to receive the **request to load** on the cargo a product container already registered in the **productservice**.*

*The request is rejected when:*

- *the product-weight is evaluated too high, since the ship can carry a maximum load of **MaxLoad>0 kg**.*
- *the hold is already full, i.e. the **4 slots** are already occupied.*

2. If the request is accepted, the *cargoservice* associates a slot to the product **PID** and returns the name of the reserved slot. Afterwards, it waits that the product container is delivered to the *ioport*. In the meantime, other requests are not elaborated.
3. is able to detect (by means of the *sonar sensor*) the presence of the product container at the *ioport*
4. is able to ensure that the product container is placed by the *cargorobot* within its reserved slot. At the end of the work:
  - the *cargorobot* should returns to its **HOME** location.
  - the *cargoservice* can process another load-request
5. is able to show the current state of the *hold*, by means of a dynamically updated **web-gui**.
6. **interrupts** any activity and turns on a led if the *sonar sensor* measures a distance **D > DFREE** for at least 3 secs (perhaps a sonar failure). The service continues its activities as soon as the sonar measures a distance **D <=DFREE**.

## Problem Analysis

Componente	Tipologia	
cargoservice	Attore	Deve essere in grado di ricevere una <b>richiesta</b> e di valutarla. Deve rispondere con esito positivo o negativo.
productservice	Attore	Necessaria comunicazione con il cargoservice, per verificare la presenza del prodotto in database. Fornito dal committente.
product	Classe	Ogni prodotto ha un nome, un peso e un PID associati. Responsabilita' di <b>productservice</b>
hold	Necessaria analisi del problema	Composta da 4 slot disponibili, ioport ed HOME, ma dai requisiti non è esplicito come formalizzarla. <b>Necessita Analisi Problema</b>
slot	Necessaria analisi del problema	Spazio dedicato al carico dei prodotti. <b>Necessita Analisi Problema</b>
ioport	Necessaria analisi del problema	Spazio dedicato alla consegna dei prodotti. <b>Necessita Analisi Problema</b>
HOME	Necessaria analisi del problema	Spazio dedicato al <i>cargorobot</i> in attesa di richieste. <b>Necessita Analisi Problema</b>
cargorobot	Attore	Utilizza il <i>basicrobot24</i> fornito dal committente per effettuare i movimenti di carico.

sonar	Necessaria analisi del problema	In grado di rilevare la presenza di un prodotto entro una certa distanza in ioport. <b>Necessita Analisi Problema</b>
web-gui	Necessaria analisi del problema	Rappresenta dinamicamente lo stato della hold. <b>Necessita Analisi Problema</b>
led	Necessaria analisi del problema	Viene acceso quando il sonar misura una distanza $D > D_{FREE}$ per almeno 3 sec. <b>Necessita Analisi Problema</b>

## cargoservice

cargoservice	Attore	Deve essere in grado di ricevere una <b>richiesta</b> e di valutarla. Deve rispondere con esito positivo o negativo.
--------------	--------	--

Il componente orchestratore del servizio cargoservice ha come compito principale quello di ricevere e valutare una richiesta.

Da analisi dei requisiti la richiesta presenterà questa forma:

**Request** requesttoload : requesttoload (PID)

**Reply** replyrequesttoload : replyrequesttoload (X) for requesttoload

L'attore cargoservice conosce solamente il PID di un container registrato su productservice.

La risposta sarà o positiva o negativa e il cargoservice processerà una sola richiesta per volta.

La prima responsabilità del cargoservice è quella di chiedere i dettagli relativi al container indicato dal PID a productservice.

## Clean Architecture's Cat Jump: si tratta di separare ciò che cambia da ciò che non può cambiare

Ciò che cambia o potrebbe in futuro cambiare è il productservice essendo un componente fornito dal produttore e non progettato da noi.

Non abbiamo controllo su come il productservice agisce e lo vediamo come una scatola nera che ad una richiesta restituisce un ID numerico di un container.

```
Request getProduct : product( ID )
Reply   getProductAnswer: product( JsonString ) for getProduct
```

Invece cargoservice sarà un attore immutabile che orchestra il ciclo di vita della requesttoload all'interno del sistema.

Deleghiamo perciò la responsabilità di comunicare con productservice ad un componente dedicato.

Chiameremo questo componente “**productpolice**”:

- Scambia messaggi con productservice (request/reply)
- Parsing dei messaggi ricevuti ed estrazione payload
- Scambia messaggi con cargoservice (request/reply) contenenti ID e PESO dell’oggetto in caso di esito positivo
- “Vive” nel contesto del cargoservice

Per questo motivo productpolice sarà un **ATTORE**.

L’interazione tra cargoservice e productpolice sarà fissa e non soggetta a cambiamenti in futuro, anche in caso di cambiamenti apportati a productservice:

#### Interazione cargoservice-productpolice

```
// Richieste interne da cargoservice a productpolice
Request productrequest : productrequest(PID) // from cargoservice to productpolice (PID)
Reply productreply : productreply(PESO) for productrequest// from productpolice to cargoservice with PID (PID, PESO)
Reply productreplyfailed : productreplyfailed (ARG) for productrequest
```

productpolice comunica per conto di cargoservice con il productservice e invia come risposta il peso del prodotto se registrato su productservice o una productreplyfailed se non lo è.

#### Interazione productpolice-productservice

```
Request getProduct : product( ID )
Reply getProductAnswer: product( JsonString ) for getProduct
```

L’interazione o il formato dei messaggi inviati da productpolice a productservice e viceversa potranno cambiare senza stravolgere la logica del cargoservice in sé.

Queste considerazioni permettono di implementare una logica immutabile in cargoservice che si preoccupa di gestire dati ricevuti internamente e “puliti”.

Da un punto di vista di sicurezza andiamo a minimizzare così possibili errori e riducendo il “point of failure” di questa comunicazione al solo productpolice (con l’assunzione di non avere controllo su productservice).

Questo principio di clean architecture può essere applicato a tutti i componenti orchestrati da cargoservice che saranno delle “blackbox” a cui delegare delle responsabilità e che comunicheranno un esito positivo o negativo a cargoservice.

Il flusso operativo di **cargoservice** sarà quindi caratterizzato da un'esecuzione lineare:

- Ad ogni step corrisponde il successivo e solo il successivo
- In caso di fallimento di un qualsiasi step allora il sistema risponde al richiedente che la requestload è stata rifiutata
- Ogni richiesta inviata al cargoservice verrà gestita singolarmente e potrà essere ricevuta solamente quando il sistema si trova in uno stato di attesa richiesta "waitRequest"
- Ogni richiesta ricevuta quando il sistema non è nello stato "waitrequest" non verrà elaborata
- La responsabilità di comunicare con l'esterno sarà delegata a degli attori trasparenti al cargoservice (e mutabili in caso di necessità)
- Ne consegue che in caso di fallimento il sistema sarà pronto ad accogliere una nuova richiesta immediatamente.

### ProductService (PRD-1, P-1) e productpolice

L'attore productservice è fornito dal committente, sviluppato in linguaggio qak e si occupa di fornire il PID e peso dei prodotti registrati in database (anch'esso fornito dal committente).

#### Interazione productpolice-productservice

```
Request getProduct : product( ID )  
Reply getProductAnswer: product( jsonString ) for getProduct
```

Analisi del formato dei messaggi inviati da productservice a productpolice:

```
//String JSON '{"productId":31,"name":"p31","weight":311}'
```

Il formato dei messaggi di tipo JSON richiede il parsing dei messaggi dentro l'attore per estrarre i valori di peso e PID (realizzato con libreria JSON: com.googlecode.json-simple).

#### Interazione cargoservice-productpolice

```
// Richieste interne da cargoservice a productpolice  
Request productrequest : productrequest(PID) // from cargoservice to productpolice (PID)  
Reply productreply : productreply(PESO) for productrequest// from productpolice to cargoservice with PID (PID, PESO)  
Reply productreplyfailed : productreplyfailed (ARG) for productrequest
```

Quindi l'attore:

1. Viene creato da cargoservice
2. Riceve in ingresso un ID numerico da parte di cargoservice

3. Richiede il prodotto corrispondente ad ID numerico a productservice
4. Estrae il PESO e controlla che l'id restituito coincide con quello richiesto
5. Comunica il PESO o una productreplyfailed a cargoservice
6. Finito il ciclo di esecuzione si distrugge

L'attore productpolice sarà un attore dinamico che verrà creato al momento della gestione della richiesta e al termine del suo flusso di esecuzioni si distruggerà liberando risorse sul sistema. Ogni richiesta implicherà perciò la creazione/distruzione di productpolice. Questa scelta è motivata dal fatto che non è specificato nei requisiti il numero di richieste da gestire in un dato intervallo di tempo, potrebbero quindi passare ore tra una richiesta e l'altra o pochi secondi.

## Hold

hold	Necessaria analisi del problema	Composta da 4 slot disponibili, ioport ed HOME, ma dai requisiti non è esplicito come formalizzarla. <b>Necessita Analisi Problema</b>
slot	Necessaria analisi del problema	Spazio dedicato al carico dei prodotti. <b>Necessita Analisi Problema</b>
ioport	Necessaria analisi del problema	Spazio dedicato alla consegna dei prodotti. <b>Necessita Analisi Problema</b>
HOME	Necessaria analisi del problema	Spazio dedicato al <i>cargorobot</i> in attesa di richieste. <b>Necessita Analisi Problema</b>

La presente fase di analisi si pone l'obiettivo di risolvere le ambiguità strutturali emerse durante lo Sprint 0, in particolare riguardo alla definizione del componente **hold** e alla rappresentazione dell'ambiente operativo.

L'analisi dei requisiti evidenzia che la **hold** non può essere ridotta a un archivio dati passivo. Essa ricopre un ruolo critico nel mantenimento della consistenza del sistema, dovendo garantire:

### 1. Incapsulamento dello stato

La hold è l'unica proprietaria dei dati degli slot, nessun altro può modificare lo stato del magazzino.

### 2. Disaccoppiamento

Anche se la hold dovesse essere su un nodo diverso del cargoservice, l'interazione sarebbe la stessa.

### 3. Single Responsibility

La hold gestisce lo stato del magazzino

### 4. Gestione attesa richieste

Se arrivano più richieste, la hold le gestisce una alla volta

Di conseguenza, il modello architetturale adottato per la **hold** è quello di un **Attore QAK**. A tale componente è delegata la responsabilità esclusiva di monitorare lo stato di occupazione degli slot e di validare le transazioni di carico prima che queste vengano tradotte in comandi attuativi.

Il requisito funzionale relativo al posizionamento del container impone la necessità di formalizzare la topologia dell'ambiente. Le definizioni logiche (es. "Slot 1") risultano insufficienti per la navigazione di un'unità mobile.

Basandosi sulle specifiche dimensionali del robot e dell'area di lavoro , si è proceduto alla **discretizzazione dello spazio fisico** attraverso un modello a **Grid Map**. L'area operativa è rappresentata come una matrice 7×6, in cui ogni cella corrisponde all'unità di ingombro del robot.

Dall'analisi del layout fisico , si deriva la seguente mappatura univoca tra le entità logiche del dominio e le coordinate cartesiane del sistema di riferimento:

Entità Logica	Descrizione Funzionale	Coordinate Griglia (x,y)
<b>HOME</b>	Posizione di riposo e inizializzazione	(0, 0)
<b>IOPort</b>	Punto di ingresso e prelievo del carico	(4, 0)
<b>Slot 1</b>	Area di stoccaggio 1	(1, 1)
<b>Slot 2</b>	Area di stoccaggio 2	(1, 4)
<b>Slot 3</b>	Area di stoccaggio 3	(3, 1)
<b>Slot 4</b>	Area di stoccaggio 4	(3, 4)

# Cargorobot

Un nodo cruciale dell'analisi riguarda l'integrazione del software di movimentazione fornito dal committente (**basicrobot24**). L'analisi si concentra dunque su come il **cargorobot** debba interfacciarsi con esso per tradurre gli obiettivi in comandi di movimento.

Poiché le operazioni di movimento fisico richiedono tempo e devono essere atomiche (il robot non può ricevere nuove destinazioni mentre è in viaggio), l'interazione viene modellata utilizzando un pattern **Request-Reply**:

1. Il **cargorobot** deve preliminarmente acquisire l'uso esclusivo del **basicrobot** inviando una richiesta di ingaggio (**engage**). Questo previene conflitti con altri potenziali utilizzatori del robot.
2. I comandi di spostamento vengono inviati tramite richiesta **moverobot(TARGETX, TARGETY)**. Il **cargorobot** attende la risposta (**moverobotdone** o **moverobotfailed**) prima di procedere allo step successivo.

Dall'analisi delle capacità del **basicrobot**, emerge una chiara separazione dei compiti che semplifica l'architettura del **cargorobot**:

- Il **basicrobot** incapsula la complessità della navigazione. Al suo interno risiede l'algoritmo di **Pathfinding A\*** necessario per calcolare il percorso ottimale ed evitare ostacoli fissi mentre si raggiunge la coordinata (**x, y**) richiesta.
- Il **cargorobot** agisce come un livello di astrazione logica. La sua responsabilità è consultare la tabella di mapping per tradurre l'obiettivo (es. "Slot 1") nelle coordinate fisiche corrispondenti (es. **1, 1**) e inoltrarle al **basicrobot**.

Questa architettura conferma che il **cargorobot** non necessita di conoscere la topologia degli ostacoli o algoritmi di routing, ma deve solo conoscere la mappatura **Destinazione → Coordinate**.

## Test Plans

In questa sezione definiamo la strategia di verifica adottata per validare la business logic del **cargoservice**. L'obiettivo è garantire che il sistema rispetti i requisiti funzionali e gestisca correttamente le anomalie prima di procedere all'integrazione con l'hardware reale.

Per eseguire i test in modo automatizzato e ripetibile, isolando la logica di business dalle dipendenze esterne, abbiamo predisposto un ambiente composto da:

- **ctxusermock**: Un contesto QAK dedicato che ospita un attore *mock* (simulatore cliente). Questo attore invia sequenze predeterminate di **requesttoload** e verifica che le risposte del sistema (**loadaccepted/loadrefused**) corrispondano alle attese.

- **productservice**: Il database è popolato con un set di prodotti noti (es. PID validi, PID inesistenti, PID con peso eccessivo) per stimolare deterministicamente le diverse ramificazioni del codice.

Sulla base degli *Edge Cases* identificati in fase di analisi , abbiamo formalizzato la seguente suite di test funzionali.

Scenario	Input	Condizione del Sistema	Risultato Atteso
<b>Carico Standard</b>	requesttoload(PID_VALIDO)	Stiva vuota, Peso < MaxLoad	Ricezione <b>loadaccepted</b> , assegnazione slot, avvio robot.
<b>Rifiuto per Sovrappeso</b>	requesttoload(PID_HEAVY)	Stiva vuota, Peso > MaxLoad	Ricezione <b>loadrefused(reason=overweight)</b> . Nessuno slot riservato.
<b>Rifiuto per Stiva Piena</b>	requesttoload(PID_VALIDO)	4 slot occupati/riservati	Ricezione <b>loadrefused(reason=full)</b> .
<b>PID Non Valido</b>	requesttoload(PID_NOT_FOUND)	N/A	Ricezione <b>loadrefused(reason=unknown_pid)</b> . Fallimento <b>productpolice</b> .
<b>Fallimento Robot</b>	Simulazione guasto	Robot non risponde	Ripristino stato coerente e ritorno a <b>waitRequest</b> .

# Project

La fase di progettazione ha tradotto le decisioni analitiche in un'architettura software concreta basata sul meta-modello QAK. L'implementazione si articola su un sistema distribuito che separa la logica di dominio (**ctxcargoholdservice**) dai servizi esterni (**ctxcargoservice**, **ctxbasicrobot**).

Applicando questi principi agli altri componenti individuati nell'analisi dei requisiti delineiamo un primo modello di **cargoservice** basato solo sulle risposte comunicate da ogni ATTORE senza preoccuparci della logica interna di ognuno di essi:

1. **productpolice** riceve un ID e restituisce un PESO
2. **hold** riceve un PESO e un ID e restituisce uno SLOT e un ID
3. **Un Attore Mock** (sonaradapter) che simula il comportamento di un componente che gestisce il sonar. Riceve una request e risponde con esito positivo o negativo (per il momento è un mock, quindi solo esiti positivi)
4. **cargorobot** riceve uno SLOT e risponde con esito positivo o negativo

Il costrutto fornito dal linguaggio QAK della request/response è perfetto per questo tipo di scambi di informazione.

**Il sonar e la logica di gestione di I/O localizzata verrà gestita nello sprint2, perciò al momento il sonaradapter restituisce solo esiti positivi.**

## Architettura Logica

Il sistema è strutturato in contesti che isolano le responsabilità:

- **ctxcargoholdservice**: Contiene gli attori proprietari (**cargoservice**, **hold**, **cargorobot**) per massimizzare la coesione e minimizzare la latenza interna.
- **ctxcargoservice**: Ospita il **productservice** (esterno).
- **ctxbasicrobot**: Ospita il software di pilotaggio robot.
- **ctxusermock**: Contesto dedicato al testing.

Ogni contesto utilizza la primitiva `ip [host="discoverable"]` e una funzione custom `register()` per integrarsi con il servizio di **Discovery Eureka**, garantendo la localizzazione dinamica dei servizi distribuiti.



**cargoservice**

Alla ricezione di `requesttoLoad(PID)`, il `cargoservice` non chiama direttamente il servizio esterno, ma istanzia un attore temporaneo `productpolice`.

Questo approccio mantiene il `cargoservice` reattivo e incapsula la logica di adattamento (parsing JSON) in un componente "usa e getta".

L'attore dinamico riceve la risposta JSON grezza dal `productservice`, esegue il parsing utilizzando `org.json.simple` e normalizza i dati prima di rispondere al creatore.

Al termine dell'operazione, l'attore esegue `terminate 0`, liberando immediatamente le risorse.

## hold

La **hold** non è un semplice database, ma un Attore attivo che incapsula lo stato dei 4 slot tramite strutture dati Kotlin (array paralleli **slotState**, **slotPid**, **slotPeso**)

```
// 4 slot S1..S4: tre array paralleli
var slotState = intArrayOf(FREE, FREE, FREE, FREE)
var slotPid   = intArrayOf(0,    0,    0,    0)
var slotPeso  = intArrayOf(0,    0,    0,    0)
```

e funzioni helper (**findFree**, **reserve**).

1. Riceve **holdrequest(PESO, PID)**.
2. Aggiorna le variabili interne (**invalid**, **overweight**, **full**).
3. Emette un **autodispatch msg1**.

La transizione successiva utilizza il pattern matching sulle guardie per determinare lo stato di arrivo:

Transition t1

```
whenMsg msg1 and [# invalid #] -> invalidPayload
whenMsg msg1 and [# overweight #] -> overweight
whenMsg msg1 and [# full #] -> full
whenMsg msg1 and [# !invalid && !overweight && !full #] ->
reserve
```

## cargorobot

Questo attore implementa la tabella di traduzione definita nell'analisi spaziale.

Utilizza un'espressione **when** di Kotlin per convertire l'indice logico dello slot in coordinate fisiche per il **basicrobot**:

```
var coords = when(slot) {
    1 -> Pair(1, 1)
    2 -> Pair(4, 1)
    3 -> Pair(1, 3)
    4 -> Pair(4, 3)
    else -> Pair(0, 0)
}
```

**Protocollo di Movimento:** Implementa una sequenza robusta di interazione con il **basicrobot**:

1. **Engage:** **request basicrobot -m engage : engage(\$MyName, 330)**
2. **Move Sequence:** **IOPort → Target → Home.**

### 3. **Disengage:** `forward basicrobot -m disengage : disengage(ok)`

Abbiamo optato per un protocollo di **engagement dinamico** del robot per ogni singola request, trattando il `basicrobot` come una **risorsa condivisa**. Questo approccio non solo evita la monopolizzazione della risorsa, permettendo l'interoperabilità con altri servizi del sistema, ma assicura che il robot venga rilasciato e torni disponibile in caso di completamento o di errori critici dell'orchestratore.

## Protocollo dei Messaggi

Le interazioni sono state formalizzate nel file `.qak` con messaggi tipizzati per garantire la validazione semantica:

- **Ingresso Sistema:** `Request requesttoload : requesttoload(PID)`
- **Interfaccia Hold:** `Request holdrequest : holdrequest(PESO, PID)`
- **Comando Movimento (Interno):** `Request moverobotinternal : moverobot(SLOT)`
- **Comando Movimento (Esterno):** `Request moverobot : moverobot(TARGETX, TARGETY)`

Questa formalizzazione garantisce che ogni componente riceva solo i dati di sua competenza (es. la `hold` riceve Peso/PID, il `robot` riceve Slot/Coordinate).

## Testing

## Maintenance

1. **Sprint 2**  
IOManager, in questa fase si svilupperanno ottimizzazioni, la parte di gestione del sonar, il rilevamento del product container nella porta di IO e l'interfacciamento hardware dei dispositivi fisici di sonar e led con il sistema.(30h/uomo)
2. **Sprint 3**  
Realizzazione della GUI (20h/uomo)