

chronicles_sprint1_v2

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 $D \leq DFREE$.

Problem Analysis

Componente	Tipologia	
cargoservice	Attore	Deve essere in grado di ricevere una richiesta 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. Responsabilità di productservice
hold	Necessaria analisi del problema	Composta da 4 slot disponibili, ioport ed HOME, ma dai requisiti non è esplicito come formalizzarla. Necessita Analisi Problema
slot	Necessaria analisi del problema	Spazio dedicato al carico dei prodotti. Necessita Analisi Problema
ioport	Necessaria analisi del problema	Spazio dedicato alla consegna dei prodotti. Necessita Analisi Problema
HOME	Necessaria analisi del problema	Spazio dedicato al cargorobot in attesa di richieste. Necessita Analisi Problema
cargorobot	Attore	Utilizza il basicrobot24 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 iport. Necessita Analisi Problema
web-gui	Necessaria analisi del problema	Rappresenta dinamicamente lo stato della hold. Necessita Analisi Problema
led	Necessaria analisi del problema	Viene acceso quando il sonar misura una distanza D > DFREE per almeno 3 sec. Necessita Analisi Problema

cargoservice

cargoservice	Attore	Deve essere in grado di ricevere una richiesta 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

Request getProduct : product(ID)

risponde con un

Reply getProductAnswer: product(JsonString) for getProduct

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

Si evince la necessità di un “adapter” che faccia da ponte tra productservice e cargoservice. L’interazione tra cargoservice ed adapter sarà fissa e non soggetta a cambiamenti in futuro, anche in caso di cambiamenti apportati a productservice:

```
// 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
```

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

Il nome scelto per questo adapter è productpolice:

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.

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 è anche buona pratica esporre il minor numero di porte quando si comunica con l’esterno, minimizzando 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 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 requesttoload 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” verrà rifiutata
- La responsabilità di comunicare con l’esterno sarà delegata a dei componenti “adapter” 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.

Applicando questi principi agli altri componenti individuati nell’analisi dei requisiti delineiamo un primo modello di cargoservice basato solo sulle risposte comunicate da ogni componente 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. sonaradapter riceve una request e risponde con esito positivo o negativo
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.

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

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).

```
Request getProduct : product( ID )
Reply   getProductName: product( JSONArray ) for getProduct
```

Apprendiamo inoltre il formato dei prodotti in database e come ci verranno restituiti:

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

Sulla base delle considerazioni esposte sopra, ecco formulato un primo modello di un attore-adapter che comunica con productservice:

Per distribuire le responsabilità e per gestire al meglio le richieste è stato scelto di sfruttare un attore adapter per il database chiamato productpolice.

Questo attore è un attore di tipo dinamico (viene creato al momento da cargoservice, instaura la comunicazione e restituisce i dati, al termine del flusso si distrugge), questa scelta garantisce l'atomicità dell'operazione di comunicazione con productservice.

Ogni nuova richiesta a cargoservice implicherà la creazione di un nuovo adapter capace di comunicare con productservice.

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).

Quindi l'attore:

1. Viene creato da cargoservice
2. Riceve in ingresso un ID da parte di cargoservice
3. Richiede il prodotto corrispondente ad ID a productservice
4. Estrae il PESO e controlla che l'id restituito coincida con quello richiesto
5. Comunica il PESO o una productreplyfailed a cargoservice
6. Finito il ciclo di esecuzione si distrugge

Sonar e iport

sonar	Necessaria analisi del problema	In grado di rilevare la presenza di un prodotto entro una certa distanza in iport. Necessita Analisi Problema
iport	Necessaria analisi del problema	Spazio dedicato alla consegna dei prodotti. Necessita Analisi Problema
led	Necessaria analisi del problema	Viene acceso quando il sonar misura una distanza $D > DFREE$ per almeno 3 sec. Necessita Analisi Problema

Cominciamo analizzando il problema del sonar:

Dal testo dei requisiti si evince che questo sonar corrisponda ad un dispositivo fisico capace di misurare la distanza di un oggetto da esso. Il dispositivo sonar misura la distanza D ogni dato intervallo temporale (vincolo software o hardware) e così se $D < DFREE$ come rappresentato in figura vuol dire che un container è in attesa presso l'IOPORT.



Questo dispositivo sonar andrà montato su un computer dedicato che si occuperà di gestire il sonar ed inviare dati al cargoservice. Approfondiremo questo aspetto nella sezione progetto, per il momento basta definirlo come un nodo distribuito.

Da qui possiamo dedurre l'analisi del problema relativa al LED.

Il LED sarà montato su questo computer dedicato al sonar e si accenderà quando la distanza misurata $D > DFREE$ il che indica errore del sonar.

sensor (sonar) SNS-2 Guasto: $D > DFREE$ per $\geq 3s \Rightarrow$ failure

LED LED-1 Indicatore di failure sonar

cargoservice CRS-7 Interrompe attività e accende LED se SNS-2

Come da requisiti il cargoservice dovrà interrompere ogni attività se LED e failure sonar.

Il sonar sarà inoltre montato in corrispondenza fisica dell'IOPORT.

Il nostro sistema software cargoservice è perciò un **sistema distribuito** e andrà trattato come tale.

Il sonar comunicherà con il cargoservice e invierà la distanza ogni secondo (intervallo scelto sulla base del requisito SNS-2). Sfruttando il linguaggio qak abbiamo 3 possibilità per il tipo di messaggi scambiati:

- Dispatch
- Event
- Request/reply

Request/reply è inadatto.

L'evento è locale al contesto qak ma può essere esteso ad altri contesti tramite protocollo MQTT.

Mentre il dispatch può essere trasmesso sia tramite MQTT che con CoAP.

Abbiamo ritenuto piu' corretta la comunicazione tramite eventi:

- Sul dispositivo sonar ogni attore percepisce l'evento distanza
- La verifica di failure si sposta sull'attore che gestisce il LED
- L'evento failure che accende il LED può essere trasmesso
- Possibilità di doppia verifica (sia su sonar che su cargoservice) per evento failure dato che è un evento che interrompe ogni attività e quindi critico

Introduciamo la necessità di un protocollo di comunicazione MQTT:

Il **protocollo MQTT** è un sistema di messaggistica leggero basato su pubblicazione/sottoscrizione, ottimizzato per comunicazioni affidabili su reti instabili tipiche dei sistemi distribuiti e dell'IoT. Il **broker MQTT** funge da intermediario centrale che gestisce lo scambio dei messaggi tra publisher e subscriber, garantendo scalabilità, disaccoppiamento e consegna efficiente dei dati.

La trasmissione degli eventi avverrà su una topic dedicata in modo da garantire riservatezza.

Per quanto riguarda il broker MQTT invece è possibile utilizzare un container docker che la nostra Software House offre per una soluzione plug-and-play.

Quindi sul "dispositivo sonar" avremo:

- Sonar - Sensore
- LED - Attuatore

e applicando i principi della clean architecture possiamo individuare questi attori:

- Attore che gestisce informazioni del sensore sonar
- Attore che gestisce l'attuatore LED
- Attore principale che orchestra la comunicazione tra attori
- Attore che gestirà la comunicazione con l'esterno

Invece dal lato cargoservice avremo:

- Attore sonaradapter che riceve gli eventi distanza o failure

La comunicazione asincrona pone un problema sull'affidabilità del sistema di interruzione attività, perciò la verifica su failure verrà effettuata anche a lato cargoservice sulla base della distanza ricevuta tramite evento.

Sicuramente questo approccio presenta delle criticità, per esempio la necessità di un broker MQTT esterno e l'invio costante di eventi. E' possibile sfruttare questo flusso costante di informazioni a scopi diagnostici in caso di guasti con un sistema di logging dedicato in futuro.

Così come la presenza del broker MQTT "semplifica" l'integrazione di altri attuatori o sensori se dovesse sorgere la necessità.

Lato cargoservice - Attore cargoservice:

Il sonaradapter è visto come una blackbox che risponde positivamente o negativamente quando interpellata.

Request sonarReq : sonarReq(X)

Reply sonarReply : sonarReply(X) **for** sonarReq

Reply sonarReplyFailure : sonarReplyFailure(X) **for** sonarReq

Per modellare il comportamento di failure e ripresa a funzionare servono perciò 2 stati:

1. Richiesta a sonaradapter - Si occuperà di mandare una sonarReq a sonaradapter per conoscere se il prodotto è arrivato alla IOPORT. In caso di esito positivo allora prosegue il flusso operativo. In caso di esito negativo - failure appunto - si passa allo stato di gestione errore.
2. Gestione errore sonar - Si occuperà di mandare nuovamente una sonarReq a sonaradapter e attende risposta - in caso di esito negativo si ripete. Questo stato sfrutta il flusso costante di eventi (contenti la distanza D) che sonaradapter riceve per riprestinare il sistema nel minor tempo possibile.

Lato cargoservice - l'attore sonaradapter invece:

- Conosce DFREE come valore hardcoded (Si può valutare anche la comunicazione di questo valore)
- Ha un flag booleano GUASTO
- Riceve una sonarReq da cargoservice al momento opportuno
- Valuta la distanza D ricevuta come evento 2 volte, a distanza di 2 secondi
- Esito positivo: risponde con esito positivo a cargoservice tramite sonarReply
- Esito negativo: Se $D > DFREE$ allora flag GUASTO = true e invia sonarReplyFailure a cargoservice
- Il sistema rimane in attesa di nuove richieste

Hold

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

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. La verifica del vincolo **MaxLoad**
2. L'accesso alle risorse limitate (i 4 slot)

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)
HOME	Posizione di riposo e inizializzazione	(0, 0)
IOPort	Punto di ingresso e prelievo del carico	(4, 0)
Slot 1	Area di stoccaggio 1	(1, 1)
Slot 2	Area di stoccaggio 2	(1, 4)
Slot 3	Area di stoccaggio 3	(3, 1)

Slot 4	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**:

- 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.
- 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
Carico Standard	<code>requesttoload(PID_VALIDO)</code>	Stiva vuota, Peso < MaxLoad	Ricezione <code>loadaccepted</code> , assegnazione slot, avvio robot.
Rifiuto per Sovrappeso	<code>requesttoload(PID_HEAVY)</code>	Stiva vuota, Peso > MaxLoad	Ricezione <code>loadrefused(reason=overweight)</code> . Nessuno slot riservato.
Rifiuto per Stiva Piena	<code>requesttoload(PID_VALIDO)</code>	4 slot occupati/riservati	Ricezione <code>loadrefused(reason=full)</code> .
PID Non Valido	<code>requesttoload(PID_NOT_FOUND)</code>	N/A	Ricezione <code>loadrefused(reason=unknown_pid)</code> . Fallimento <code>productpolice</code> .

Fallimento Robot	Simulazione guasto	Robot non risponde	Ripristino stato coerente e ritorno a <code>waitRequest</code> .
-------------------------	--------------------	--------------------	--

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`).

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.

Implementazione dei Componenti

`cargoservice`

L'attore principale implementa una FSM che coordina il flusso sequenziale. Un punto saliente è la gestione dell'interazione con servizi esterni potenzialmente lenti o inaffidabili tramite la **creazione dinamica di attori**.

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

```
create productpolice _ "p1"
requestbycreator productrequest : productrequest($ID)
```

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

productpolice

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.

```
var parser = JSONParser()
var jsonObject = parser.parse(ANSWR) as JSONObject
var productId = (jsonObject["productId"] as Long).toInt()
WEIGHT = (jsonObject["weight"] as Long).toInt()
```

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`) 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)`

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