Sprint 0

Indice

Obiettivi

In questo sprint0 i nostri obiettivi sono di analizzare e individuare sottoinsiemi di requisiti forniti dal committente e definire il nostro problema, per poi in futuro suddividere i sottoinsiemi in successivi sprint da eseguire eventualmente anche in parallelo, improntare le componenti della nostra archiettura (macrocomponenti principali & interazioni tra loro sotto forma di messaggi).

Requisiti del commitente

Requisiti del commitente

Analisi dei requisiti

Hold

È la stiva della nave, cioè l'area di lavoro e piatta dove vengono caricati i container con i prodotti. In questo progetto è una zona rettangolare con degli slot e una porta di ingresso/uscita (IOPort). Verrà utilizzato dall'attore CargoService.

Cargorobot

È il robot a guida differenziale (Differential Drive Robot) incaricato di spostare i container dentro la stiva e piazzarli nello slot assegnato. Dopo il lavoro torna sempre alla sua posizione "HOME". Ci viene gia fornito dal committente.

Products

Sono i beni/merci che devono essere caricati sulla nave. Ogni prodotto viene messo in un container di dimensioni prefissate e registrato in un sistema.

Weight

Il peso del prodotto/container. Serve per verificare che non venga superato il limite massimo di carico della nave (`MaxLoad`). Sarà un attributo del Prodotto.

CargoService

È il servizio software che gestisce la registrazione dei prodotti. Quando inserisci un prodotto specificando il peso, lui restituisce un identificativo unico (`PID`). Esso viene implementato come Attore qak.

Io-port

È la porta di ingresso/uscita della stiva. Davanti a questa porta c'è un sensore sonar che rileva se un container è presente. È il punto dove il prodotto viene consegnato prima che il robot lo carichi. Sarà un macrocomponente del nostro sistema.

Slots & Sensors

- **Slots**: sono le aree (4 in totale) all'interno della stiva dove i container vengono sistemati. Uno slot è già occupato in modo permanente, gli altri inizialmente sono liberi.
- **Sensors**: in questo caso si parla di un sonar davanti all'IOPort che rileva la presenza dei container e segnala eventuali anomalie (tipo guasto se non misura più distanze corrette).

DDR Differential Drive Robot

È il tipo di robot mobile con due ruote motrici indipendenti. Si muove facendo girare le ruote a velocità diverse (come i robot aspirapolvere), ed è quello usato come cargorobot.

Componenti fornite dal committente

Si elencano di seguito le componenti software fornite dal committente

BasicRobot

Componente software che esegue i comandi di movimento del DDR-robot. Non è a conoscenza della tecnologia con il quale il robot è stato implementato.

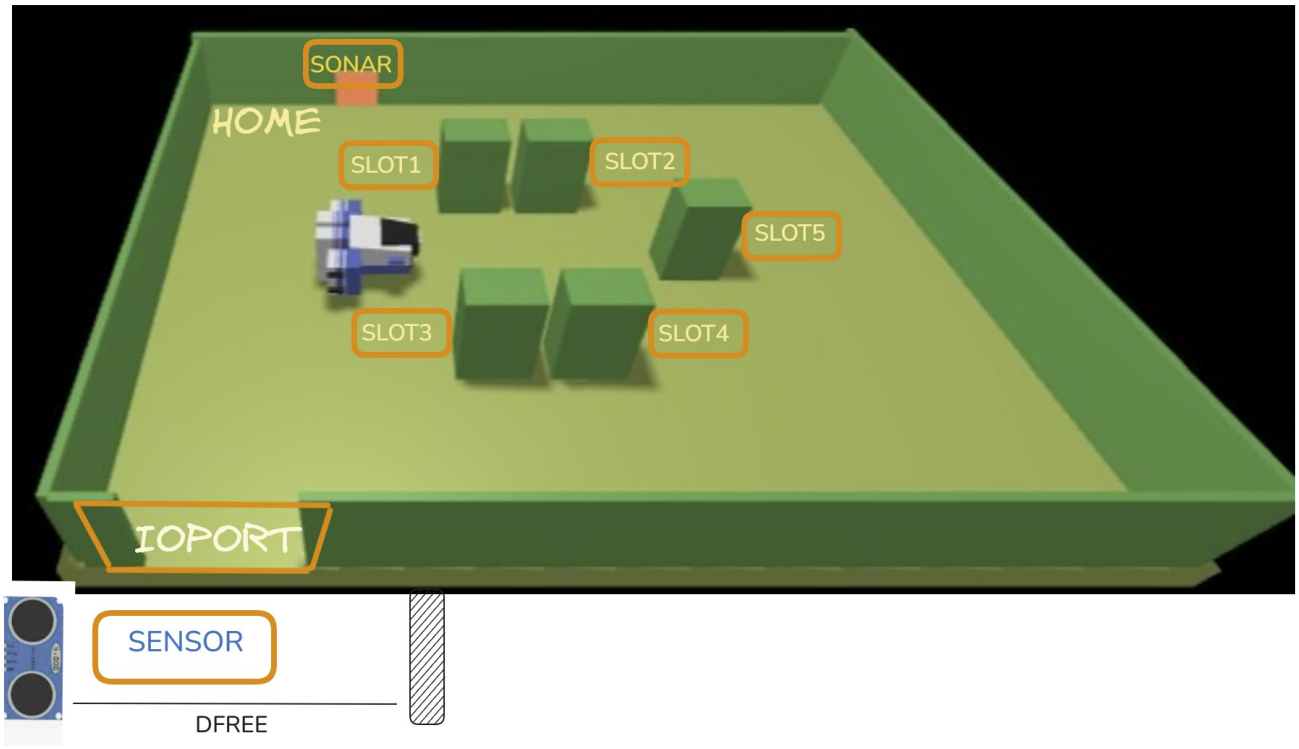
SonarLed2025

Software per la misurazione della distanza dal sonar (componente hardware) e per accendere un LED

WENV

WENV è un ambiente di simulazione software ("Web Environment") usato per testare il sistema, mostrare la stiva, lo stato degli slot e i movimenti del robot tramite un'interfaccia grafica web.

Area di Lavoro



Plain Old Java Objects (POJO)

I POJO sono essenziali per costruire il modello di dominio in DDD.
Sono usati per:
• Rappresentare entità con identità persistente (es. LoadedProduct)
• Definire value object immutabili (es. LoadRequest, ValidationResult)
• Formare aggregati, unità coerenti per la consistenza dei dati
• Incapsulare logica di business passiva e stateless

I POJO sono passivi, non gestiscono il loro stato in autonomia nel tempo in risposta a eventi esterni e non hanno code di messaggi proprie

Attori

Il modello ad Attori si basa su entità autonome che comunicano tramite messaggi. Un Attore Qak è un componente attivo con un proprio flusso di controllo autonomo e uno stato interno.

Le loro caratteristiche principali includono:

• **Gestione autonoma dello stato**: Mantengono il proprio stato interno
• **Comunicazione a messaggi**: Interagiscono esclusivamente inviando e ricevendo messaggi (Dispatch, Request, Reply, Event), promuovendo isolamento e resilienza
• **Coda di messaggi locale**: Ogni Attore ha una coda (msgQueue) per processare i messaggi sequenzialmente, gestendo naturalmente l'esigenza di non elaborare più richieste contemporaneamente
• **Comportamento come FSM**: Il loro comportamento può essere modellato come una macchina a stati finiti (FSM)
• **Adatti a sistemi distribuiti e microservizi**: Il modello Qak è specificamente pensato per la progettazione di prototipi di sistemi distribuiti, con attori che si comportano come FSM, strettamente correlati all'architettura a microservizi
• **Qak come DSL**: Il linguaggio Qak è un Domain Specific Language (DSL) che fornisce un alto livello di

astrazione per definire modelli eseguibili di sistemi basati su attori, aiutando a colmare l'abstraction gap
• **Raggruppamento in contesti**: Gli attori sono raggruppati in contesti che gestiscono le interazioni di rete tramite protocolli come TCP, CoAP, MQTT

Confronto e Complementarità

La scelta dipende dalla necessità di comportamento autonomo e stateful (Attori) o di rappresentazione dati/logica stateless (POJO)
• I POJO sono ottimi per le strutture dati e le regole di business fondamentali che operano su tali dati, fungendo da "mattoni" all'interno di un servizio o di un Attore
• Gli Attori sono ideali per gestire processi complessi e stateful, orchestrazioni e comunicazioni asincrone in ambienti distribuiti
In pratica, un sistema moderno userà spesso entrambi: gli Attori possono utilizzare i POJO per strutturare i dati che gestiscono internamente o che scambiano tramite messaggi. Questo approccio ibrido consente di beneficiare sia di una modellazione robusta del dominio sia dei vantaggi della computazione distribuita basata su messaggi.

QAK

Qak (o Qak Actors) non è un linguaggio di programmazione generico, ma piuttosto un linguaggio di modellazione eseguibile (*DSL - Domain Specific Language*) specificamente progettato per l'analisi e la progettazione di prototipi di sistemi distribuiti. La "Q" in Qak sta per "quasi" ("quasi" un attore), indicando che è un'astrazione che mira a colmare l'abstraction gap tra i concetti di alto livello e la loro implementazione. La "k" aggiunta (Qak) si riferisce alla sua implementazione in Kotlin, senza l'uso di supporti.

Il cuore di un sistema Qak è il modello ad attori, un paradigma computazionale ispirato alla fisica, in cui ogni componente del sistema è un attore autonomo che comunica tramite lo scambio di messaggi. Un attore Qak è un componente attivo con un nome univoco e un flusso di controllo autonomo. Si comporta come un automa a stati finiti (FSM), gestendo i messaggi ricevuti in relazione al suo stato corrente e alle transizioni definite. Ogni attore possiede una coda di messaggi locale (msgQueue) per elaborare i messaggi sequenzialmente, garantendo che "altre richieste non siano elaborate" mentre una è in corso.

Gli attori Qak sono raggruppati in contesti, che fungono da nodi logici di elaborazione. Questi contesti gestiscono gli attori al loro interno e li abilitano alle interazioni via rete tramite protocolli come TCP, CoAP e MQTT. L'infrastruttura Qak e la Qak software factory sono responsabili della generazione di codice Kotlin e di altre risorse, mappando i concetti di alto livello del modello in strutture eseguibili.

Perchè QAK

Qak (o Qak Actors) è un linguaggio specifico del dominio (DSL) pensato per l'analisi, la progettazione e la prototipazione di sistemi distribuiti, in particolare quelli basati su architetture a microservizi.

Perché utilizziamo Qak:

- **Supporto ai Sistemi Distribuiti e Microservizi:** Qak si allinea all’architettura a microservizi, trattando gli attori come componenti autonomi che interagiscono tramite messaggi. Ogni attore può essere visto come un microservizio con responsabilità precise.

- **Riduzione del Gap di Astrazione:** Qak colma il divario tra le primitive dei linguaggi generici e le esigenze di alto livello della progettazione di sistemi distribuiti, permettendo di esprimere concetti indipendenti dalla tecnologia.
- **Prototipazione Rapida:** La Qak software factory genera automaticamente codice Kotlin e altre risorse dai modelli Qak, accelerando la prototipazione e supportando uno sviluppo evolutivo e incrementale.
- **Comunicazione Asincrona:** Gli attori Qak comunicano esclusivamente tramite messaggi (Dispatch, Request, Reply, Event), garantendo isolamento e resilienza. Ogni attore gestisce una propria coda di messaggi per l'elaborazione sequenziale.
- **Comportamento basato su FSM:** Gli attori sono modellati come automi a stati finiti (FSM), ideali per gestire processi complessi e interazioni asincrone.
- **Flessibilità nei Protocolli di Comunicazione:** Qak supporta vari protocolli di rete (TCP, UDP, CoAP, MQTT) tramite il concetto di Interaction.
- **Valore Didattico e Sperimentale:** L'uso di attori Qak, anche in sistemi concentrati, è utile per sperimentare il modello ad attori e affrontare le sfide tipiche dei sistemi distribuiti (concorrenza, interazioni complesse).

Principi che seguiremo nel progetto:

- **Model-Driven Engineering (MDE):** Produzione esplicita di modelli eseguibili in ogni fase di sviluppo (analisi requisiti, progettazione, implementazione), che fungono da riferimento condiviso ed evolvono nel tempo.
- **Sviluppo Agile e Incrementale (SCRUM):** Organizzazione del lavoro in sprint, affrontando la complessità per passi e consegnando sottosistemi funzionanti in modo incrementale.
- **Domain-Driven Design (DDD):**
    - *Contesti Delimitati (Bounded Contexts):* Identificazione di aree distinte del dominio con confini chiari e modelli specifici (es. Gestione Carico, Inventario Stiva, Interazione Sensori, Visualizzazione, Gestione Prodotto).
    - *Linguaggio Ubiquo:* Mantenimento di un linguaggio comune tra esperti di dominio e sviluppatori.
    - *Aggregati, Entità, Value Objects:* Modellazione della struttura interna e garanzia di consistenza dei dati.
    - *Eventi di Dominio:* Comunicazione asincrona tra microservizi per ottenere consistenza finale.
    - *Orchestrazione vs. Coreografia:* Scelta tra coordinamento centralizzato o distribuito dei flussi di lavoro.
- **Architettura Pulita / Esagonale / Inversione delle Dipendenze:** Separazione della logica di business dai dettagli tecnologici, promuovendo modularità e testabilità.
- **12 Factor App Principles:** Principi per la costruzione di applicazioni SaaS robuste e scalabili (gestione configurazione, logging, concorrenza, ecc.).
- **Principi SOLID:** Fondamentali per la progettazione orientata agli oggetti (Responsabilità Singola, Aperto/Chiuso, Sostituzione di Liskov, Segregazione delle Interfacce, Inversione delle Dipendenze).
- **Test Continuo e Osservabilità:** Definizione di piani di test automatizzati fin dall'analisi per garantire copertura dei requisiti e individuazione precoce dei difetti. L'applicazione deve essere osservabile per la verifica automatizzata dei suoi effetti.
- **Progettazione Top-Down:** Si parte dall'analisi dei requisiti e del problema, progredendo verso progettazione e implementazione, per affrontare la complessità in modo sistematico.

- **Distinzione tra Interazioni H2M e M2M:** Riconoscimento delle differenze tra interazioni Uomo-Macchina e Macchina-Macchina, per progettare interfacce, protocolli e strategie di test adeguate.

Link alla documentazione ufficiale Qak

Macrocomponenti

Gli attori consentono di gestire la distribuzione o la concentrazione dei componenti sui nodi della struttura. In questa fase dello sviluppo non siamo ancora nella condizione tale da poter stabilire il grado di distribuzione di tutta l'archiettura.

Le componenti che rappresentano l'entità fisica sappiamo gia che risiederanno sul nodo dedicato (`Sonar` & `Led`). Stessa cosa vale per il componente che sostituirà `Basicrobot`.

In aggiunta alle Componenti del committente, si svilupperanno i seguenti Macrocomponenti:

CargoService

Il CargoService rappresenta il nucleo della logica di business del sistema, allineandosi al concetto di "Gestione Carico" (Loading Management) identificato come Bounded Context

Le sue responsabilità principali sono:

• **Gestire le richieste di carico**: Riceve le richieste esterne per il carico di un prodotto e ne gestisce l'intero ciclo di vita.
• **Orchestrare il processo**: Agisce come orchestratore della saga di caricamento. Decide se accettare o rifiutare una richiesta dopo aver eseguito le necessarie validazioni, come il controllo del peso massimo e la disponibilità di slot nella stiva
• **Coordinare gli altri componenti**: Interagisce con gli altri macro-componenti per portare a termine il processo. Ad esempio, richiede informazioni sul prodotto (come il peso) a un servizio dedicato, verifica la disponibilità di slot interrogando il componente `hold`, e comanda al cargorobot di eseguire la movimentazione.
• **Garantire l'elaborazione sequenziale**: Assicura che una sola richiesta di carico venga processata alla volta, mantenendo lo stato dell'operazione corrente

Cargorobot

Il cargorobot è il componente software che astrae e controlla il robot DDR (Differential Drive Robot) fisico o virtuale, che opera come un carrello di trasporto (transport trolley).

Le sue funzioni principali sono:

• **Esecuzione di comandi di movimento**: Funge da puro esecutore di comandi di basso livello inviati da un componente di livello superiore (come il CargoService). Questi comandi possono essere movimenti elementari (avanti, indietro, ruota), step di durata definita, o piani di movimento complessi (`doplan`).
• **Indipendenza dalla tecnologia**: Offre un'interfaccia software che permette di operare con il robot indipendentemente dalla sua realizzazione specifica (reale o virtuale), nascondendo i dettagli tecnologici.
• **Interazione con l'ambiente**: Può essere una fonte di informazioni, emettendo eventi relativi al suo stato o a ciò che percepisce, come i dati di un sonar (sonardata).

Sonar

Il sonar è un agente software situato che si interfaccia con il sensore a ultrasuoni fisico (es. HC-SR04), tipicamente montato su un dispositivo come un Raspberry Pi.

Le sue responsabilità sono:

• **Astrazione dell'hardware**: Nasconde i dettagli di basso livello dell'interazione con il sensore fisico.
• **Rilevamento e misurazione**: Misura la distanza da eventuali ostacoli e fornisce questi dati al sistema. Nel contesto specifico del sistema cargo, il suo ruolo cruciale è rilevare la presenza di un container nell'area di I/O (`IOPort`).
• **Fornitura di dati filtrati**: Invia i valori di distanza rilevati dopo averli filtrati per renderli significativi per il livello applicativo (es. valori interi entro un certo range).
• **Emissione di eventi**: Comunica le informazioni rilevanti al resto del sistema emettendo eventi, come sonardata o un più specifico evento di business come obstacle.
• **Controllo**: Può essere controllato tramite comandi remoti come sonarstart e sonarstop per attivarne o disattivarne il funzionamento.

Led

Il led è un altro agente software situato con la funzione di attuatore. È responsabile del controllo di un LED fisico, tipicamente connesso a un pin GPIO di un Raspberry Pi.

• **Fornire feedback visivo**: La sua funzione primaria è quella di segnalare visivamente lo stato del sistema o di un particolare sottosistema.
• **Esporre un'interfaccia di controllo**: Deve offrire un'interfaccia semplice per essere comandato da altri componenti, accettando messaggi come `turnOn` e `turnOff`.

Hold

Il componente hold è un modello software che rappresenta lo stato della stiva della nave (cargo hold). Questo si allinea con il Bounded Context "Inventario Stiva" (Hold Inventory).

• **Gestire l'inventario della stiva**: È la fonte autorevole (source of truth) per lo stato di occupazione della stiva. Mantiene una mappa degli slot, registrando quali sono liberi e quali sono occupati.
• **Tracciare i prodotti**: Per ogni slot occupato, memorizza quale prodotto (identificato da un PID) vi è stato collocato.
• **Fornire informazioni di stato**: Risponde a interrogazioni da parte di altri servizi, come il CargoService (che ha bisogno di sapere se ci sono slot liberi) e la web-gui (che deve visualizzare la mappa della stiva).

Web-gui

La web-gui è l'interfaccia uomo-macchina (HMI) del sistema, concepita come un "dispositivo di I/O evoluto". Corrisponde al Bounded Context "Visualizzazione Stiva" (Hold Visualization).

• **Visualizzazione dinamica**: Fornisce una rappresentazione grafica e in tempo reale dello stato della stiva.
• **Aggiornamento automatico**: Deve aggiornarsi automaticamente per riflettere i cambiamenti di stato nel backend senza che l'utente debba ricaricare la pagina. Questo suggerisce l'uso di tecnologie push come le WebSocket.
• **Interazione utente**: Permette agli utenti di interagire con il sistema, ad esempio inviando comandi per

avviare nuove richieste di carico.
• **Architettura disaccoppiata**: Può essere realizzata come un microservizio a sé stante, disaccoppiato dalla logica di business principale, con cui comunica attraverso protocolli di messaggistica come MQTT o tramite API.

Architettura di Riferimento

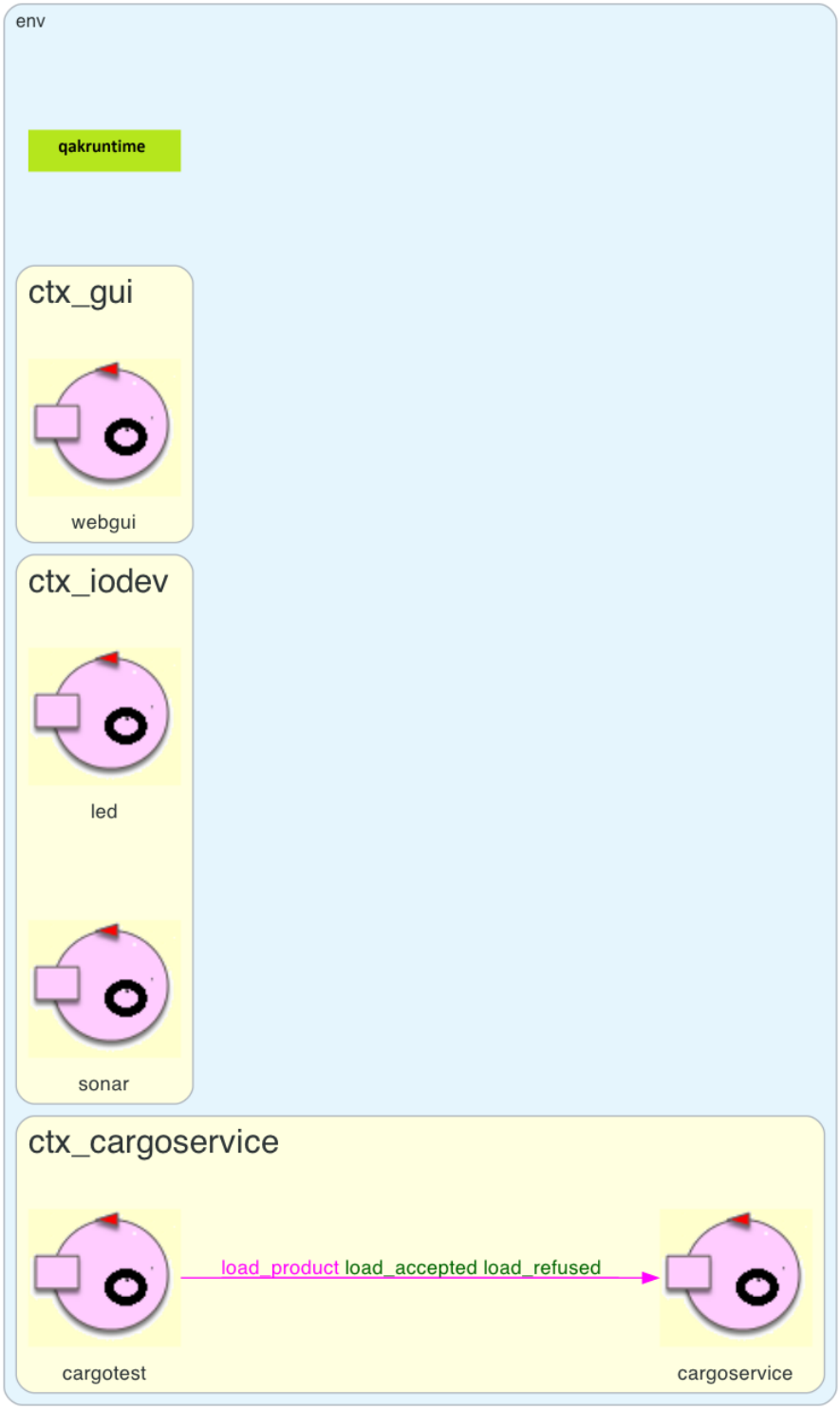
Modello di comunicazione a Messaggi

Il modello ad attori sfrutta la comunicazione tramite messaggi e dai requisiti forniti siamo in grado di comprendere alcune delle interazioni che avvengo tra gli attori. Si indicano di seguito i messaggi che sono in grado di scambiarsi tra di loro.

- `Request load_product : load_product(PID)` richiesta di carico di un prodotto con PID
- `Reply load_accepted : load_accepted(SLOT) for load_product` restituisce lo slot assegnato
- `Reply load_refused : load_refused(CAUSA) for load_product` ritorna la causa del mancato carico

Serviranno successive decisioni per la modellazione e l'implementazioni di messaggi tra attori per ulteriori funzionalità.

Schema dell'architettura



sprint0Arch

Piano di Test

In questa prima fase i test servono a controllare che i prototipi dei componenti interagiscano come richiesto dal committente.

- tentativo accettato di carico
- tentativo rifiutato di carico per troppo peso
- tentativo rifiutato per mancanza di slot

```
State richiesta {
    println("[cargotest] Invia una nuova richiesta") color yellow
    // Invio della richiesta
    request cargoservice -m load_product:load_product(1)
    request cargoservice -m load_product:load_product(1)
    request cargoservice -m load_product:load_product(1)
}
Goto waiting_for_response

State waiting_for_response {
}

Transition t0
    whenReply load_accepted -> loadAccepted
    whenReply load_refused  -> loadRefused

State loadAccepted {
    println("[cargotest] risposta arrivata") color blue

    onMsg(load_accepted : load_accepted(SLOT)) {
        [# val Msg = payloadArg(0).toInt() #]
        println("[cargotest] Richiesta accettata, slot n. $Msg ") color
yellow
    }
}
Goto waiting_for_response

State loadRefused {
    onMsg(load_refused : load_refused(CAUSA)) {
        [#
        var Msg = payloadArg(0)
        #]
        println("[cargotest] Richiesta rifiutata causa : $Msg ") color
yellow
    }
}
```

Piano di Lavoro

Successivi allo sprint0 si distinuguono i seguenti sprint operativi del nostro processo Scrum

1. Sprint 1 (30h)
    - CargoService (core buisness del sistema)
    - Cargorobot
2. Sprint 2(20h)
    - Hold
3. Sprint 3(20h)
    - Sonar
    - Led
4. Sprint 4(10h)
    - Web Gui