# AMW - Agents Managed Warehouse

Paolo Baldini
paolo.baldini6@studio.unibo.it

30 maggio 2021

Rispetto a qualche decennio fa', oggi i magazzini hanno al loro interno una notevole mole di complessità. La ricerca di sempre più efficienza ha infatti portato col tempo le aziende a fornire ai loro dipendenti strumentazioni per rendere più agevole e produttivo il lavoro e, le più grandi di queste, hanno addirittura sostituito parte del lavoro dipendente con macchine totalmente automatizzate. Prendendo in considerazione quest'ultima casistica, la pratica attuale più comune consiste nell'automatizzare il deposito per mezzo di pochi PLC che si occupino della gestione del sistema nel suo complesso.

Tramite l'implementazione di un sistema in cui ogni macchina appaia come un agente robotico, si vuole fornire un'alternativa distribuita alla pratica di centralizzazione attuale. In questa visione, le macchine dovranno gestire ordini entranti, recuperare i prodotti richiesti e risultare il più flessibili possibile, anche tramite l'upgrade automatico di conoscenza procedurale nello svolgimento di attività 'non standard'.

# Indice

# 1 Obiettivi e requisiti

Il progetto mira allo sviluppo di un sistema simulato di controllo di un magazzino, il quale sarà gestito da agenti software e robotici. La scelta dell'implementazione simulata del sistema è dovuta alla mancanza di controllori fisici atti a rappresentare gli agenti robotici. Oltre a questo, la scelta permette di mettere in secondo piano le complicazioni hardware, che non rappresentano invero il focus del progetto.

Nel sistema si prevederà la possibilità di sottoporre un ordine. I prodotti che lo compongono saranno quindi raggruppati dagli agenti robotici incaricati. La suddivisione dei compiti di recupero delle parti dell'ordine sarà diretta invece da un agente 'coordinatore'.

Oltre al normale funzionamento di un magazzino, si desidera inoltre permettere agli agenti di eseguire task non precedentemente conosciuti. Lo sviluppo di questa caratteristica si delinea nella creazione di un repository di piani/abilità a cui i vari agenti possano collegarsi per ottenere la capacità richiesta.

A livello tecnologico, il progetto prevede lo sviluppo di agenti JADE[6] e Jason[3] (questi ultimi, programmati quindi nella versione estesa di AgentSpeak(L)). Gli agenti funzioneranno su di un infrastruttura distribuita JADE. Allo scopo di sviluppare la capacità di scaricamento dei piani, si ritiene infine utile la definizione di ontologie atte a descrivere il dominio del magazzino.

Come obiettivi di apprendimento, ci si pongono quelli di acquisire conoscenza relativamente al modello BDI[1], approfondire aspetti più avanzati di JADE e per quanto possibile, valutare l'utilizzo e l'impatto della rappresentazione di un dominio mediante ontologie in un sistema distribuito. Si valuta infine la possibilità di apprendimento di Docker[4] per il deployment delle componenti del sistema.

Un elenco formale dei requisiti del progetto sarà fornito di seguito.

## 1.1 Requisiti funzionali

- Implementare un sistema dinamico che permetta l'aggiunta di agenti runtime;
- Permettere l'interazione col sistema tramite un client dedicato;
- Rendere possibile l'esecuzione di task non precedentemente definiti;
- Rendere possibile la sottomissione di ordini da utenti esterni al sistema.

## 1.2 Requisiti non funzionali

- Studiare Jason e il modello BDI ed utilizzarli per la creazione degli agenti del sistema 'magazzino';
- Approfondire la conoscenza del framework JADE ed utilizzarlo come architettura base per la comunicazione degli agenti distribuiti;

[1] Beliefs, Desires, Intentions

- Definire una ontologia rappresentante il magazzino da utilizzare per la definizione di azioni e conoscenze;
- Utilizzare un approccio test-driven per monitorare il corretto funzionamento del sistema.

## 1.3 Risultati attesi

Il risultato atteso è un sistema simulante un magazzino. Questo dovrà accettare ordini entranti e adoperarsi per il loro corretto completamento.

Si prevede la presenza di una interfaccia utente tramite la quale sia possibile sottomettere ordini, eseguire task e visualizzare lo stato del sistema. Quest'ultima caratteristica non sarà tuttavia considerata un elemento obbligatorio nella soluzione finale e ci si riserva la possibilità di cancellarne l'implementazione in caso di problemi o se i tempi di sviluppo dovessero estendersi troppo.

Gli artefatti derivanti dallo sviluppo del progetto saranno possibilmente immagini docker rappresentanti le entità del sistema; un software client funzionante su JVM per l'interazione con esso.

## 1.4 Scenari d'utilizzo

Come giá accennato, il sistema AMW é simulato. Questo impedisce un suo utilizzo diretto in un ambiente reale, ma puó rappresentare un buon punto di partenza per l'identificazione delle possibili criticitá di un sistema di questo tipo.

Un sistema completo basato sulla struttura di AMW permetterebbe una gestione completa del magazzino. Non é inoltre necessariamente richiesta nessuna presenza umana nello stesso, grazie alla flessibilitá offerta dall'esecuzione di task personalizzati da parte degli agenti robotici. Questo determina un vantaggio notevole nella gestione di magazzini 'particolari' in cui sia necessario mantenere valori ambientali estremi. Un esempio potrebbe essere l'immagazzinamento di elementi particolari che richiedano temperature molto basse a cui gli umani non possono lavorare.

Gli altri vantaggi forniti dal sistema sono sostanzialmente relativi alla riduzione dei lavoratori umani (che potrebbero essere quindi sostituiti dagli agenti) e la resilienza a rotture parziali. La rottura di alcuni macchinari causa infatti, in molte realtá industriali, un fermo della produzione. Un sistema come AMW ridurrebbe invece la produttivitá, senza tuttavia bloccarla del tutto. Questo risulterebbe quindi in grado di continuare a funzionare, seppur a velocitá ridotta.

Per quanto riguarda una definizione piú pratica di uno scenario di utilizzo, questo comprende la possibilitá per gli utenti di visualizzare i prodotti disponibili, effettuare ordini e vederne lo stato di completamento. Gli amministratori possono invece visualizzare lo stato del magazzino, aggiungere prodotti in una scaffalatura e richiedere l'esecuzione di task agli agenti dello stesso.

## 1.5 Politica di autovalutazione

Il progetto è stato sviluppato, attraverso i linguaggi di programmazione Kotlin e ASL, mediante l'utilizzo dei framework JADE e Jason. Questo mira a rispettare i requisiti di chiarezza del codice, modularitá e riuso. Cerca inoltre, quando possibile, di utilizzare pattern e strutture architetturali conosciute. Si ritiene che ció debba essere incluso nella politica di autovalutazione in quanto punto focale dello sviluppo di qualsiasi sistema.

Passando ad un livello piú specifico tuttavia, l'efficacia del progetto può essere valutata analizzandolo dal punto di vista del rispetto dei requisiti. Questo copre completamente i requisiti imposti in fase di definizione del goal. Inoltre, si comporta come atteso, garantendo le proprietá richieste. Alcune imperfezioni e comportamenti non ottimali sono tuttavia presenti allo scopo di poter comunque garantire le funzionalitá definite.

# 2 Analisi dei requisiti

I requisiti di alto livello definiti nella sezione precedente ci fornisco un punto di partenza, il quale non risulta però sufficiente a spiegare la scelta tecnologica ed il design scelto. In questa sezione saranno dunque analizzate proprietà implicite che possiamo derivare dalle precedenti definizioni, in modo da ottenere e chiarire le caratteristiche richieste al nostro sistema.

Successivamente, si analizzeranno alcune tecnologie e/o paradigmi che possono risultare utili al conseguimento dei requisiti.

## 2.1 Scalabilità

Man mano che la domanda aumenta, gli elementi del sistema dovrebbero poter essere aggiunti in modo da ampliare le capacità dello stesso. Ciò è tuttavia fattibile principalmente per le entità software, mentre quelle hardware richiederebbero per questo un design dedicato allo scopo. Non volendo entrare in dettagli tanto profondi, si considererà che il design adeguato per queste sia già fornito e si modelleranno semplicemente i comportamenti ad alto livello atti a raggiungere una scalabilità teorica.

## 2.2 Resilienza

Il malfunzionamento di un entità del sistema non dovrebbe bloccare il funzionamento dello stesso. Ció é tuttavia fattibile solo fino ad un certo punto. Le entitá hardware del sistema sono infatti aggiungibili e rimpiazzabili in maniera limitata, a causa della componente fisica del sistema stesso. Limitatamente alle possibilitá disponibili, si cercherá di mantenere il sistema funzionante tramite distribuzione, indipendenza e aggiunta di nuove entitá al sistema stesso.

## 2.3 Semplicità di messa in funzionamento

Il requisito precedentemente definito cita la possibilitá di aggiungere nuove entitá al sistema. A tal scopo si ritiene necessario che, ad esempio nel caso di fallimento di un componente, questo possa essere facilmente rimpiazzato (quanto meno nel caso software).

## 2.4 Le tecnologie scelte

Diversi framework forniscono differenti modalità di programmazione. A causa di ciò, un'oculata scelta degli stessi risulta fondamentale per un corretto sviluppo del sistema. Questi devono infatti favorire il raggiungimento dei requisiti precedentemente definiti.
Di seguito verranno descritte le tecnologie scelte e le caratteristiche che hanno portato alla loro adozione.

### 2.4.1 Il framework JADE

Il framework di appoggio scelto per lo sviluppo del progetto è stato JADE: JAVA Agent DEvelopment framework. Questo permette lo sviluppo di applicativi distribuiti basati sulle astrazioni degli agenti.

Il suo utilizzo porta molti vantaggi. In primis, la sua storia di sviluppo lo rende un framework molto stabile e maturo, utilizzato in contesti aziendali da diverso tempo e quindi adeguatamente testato ed affidabile. Questo include inoltre funzionalità avanzate, di cui si è valutato l'utilizzo, quali la definizione di ontologie a livello di programma.

Il framework è stato inoltre approcciato durante il corso e la sua adozione ha permesso di focalizzare gli sforzi e gli studi sull'utilizzo di tecnologie non ancora conosciute.

Come ultimo punto, la documentazione di JADE è curata, il che agevola ulteriormente nello sviluppo di applicazioni di notevole complessità.

### 2.4.2 L'interprete Jason

Come tecnologia per lo sviluppo di alcuni degli agenti del sistema si è scelto di utilizzare il linguaggio ASL (AgentSpeakLanguage) nella sua estensione interpretata da Jason. Seppur ciò non fosse necessario allo sviluppo del sistema, le ragioni che hanno spinto a questa scelta sono legate alla volontà di apprendere il modello di programmazione BDI (Belief, Desire, Intention).

L'adozione di questa tecnologia ha anche permesso di entrare a contatto sia con lo standard di comunicazione FIPA-ACL[5] (negli agenti JADE) sia con quello KQML[2] (negli agenti Jason), dando modo di valutare ed utilizzare entrambi.

### 2.4.3 La piattaforma Docker

La tecnologia Docker permette di isolare i componenti del sistema e di eseguirli in maniera indipendente. Il suo utilizzo porta numerosi vantaggi, quali una notevole semplicità di deployment e scalabilità, a partire da una semplice immagine di base.

Docker modularizza inoltre la singola entità del sistema, permettendo una compartimentalizzazione implicita e, in questo contesto, la possibilità di simulare una distribuzione su diverse macchine. Le istanze delle immagini docker comunicano infatti su di una rete fornita dalla piattaforma, simulando quindi implicitamente una distribuzione del sistema.

Come ultimo punto, l'utilizzo di immagini dockerizzate permette di aggiungere in maniera semplice nuovi agenti al sistema durante le simulazioni.

L'utilizzo di questa tecnologia non era ovviamente necessario allo scopo base del progetto, ma la possibilità di simulare una distribuzione delle entità in rete era auspicabile e se ne è perciò deciso l'uso.

### 2.4.4 Il tool di automazione Gradle

Gradle[1] è un tool di build automatica solitamente utilizzato per applicativi Java o funzionanti su JVM. Nel progetto è stato utilizzato per la definizione di task/comandi

'user friendly' per la messa in esecuzione del sistema in tutte le sue componenti. Esso viene principalmente utilizzato per la creazione e l'avvio dei componenti dockerizzati, ma si occupa anche dell'esecuzione del client e del container principale di JADE. Quest'ultimo è infatti avviato direttamente sulla macchina, che funziona da main host per l'infrastruttura (e non è quindi dockerizzato).

L'utilizzo di Gradle permette infine un avvio istantaneo di tutte le componenti del sistema multi-agente tramite l'utilizzo di pochi comandi.

# 3 Design

In questa sezione si analizza il sistema AMW da punto di vista di *struttura*, *comportamento* ed *interazione*. Si discuterà inizialmente la struttura di massima del sistema e le entità rappresentate. Successivamente, si analizzerà il loro comportamento. Infine verranno presentate le modalità di interazione.

## 3.1 Struttura

Il sistema AMW è un software distribuito basato sull'astrazione degli agenti, che definiscono la struttura e le comunicazioni di ogni singola entità del magazzino. Questi sono dunque molteplici e la loro interazione permette al behaviour complessivo del magazzino di emergere.

Nello specifico, gli agenti che concorrono al corretto funzionamento del sistema sono:

- Admin
- User
- Collection Point Manager
- Command Manager
- Order Manager
- Robot Picker
- Warehouse Mapper

Ogni agente svolge un preciso compito e interagisce con uno o più agenti differenti. Ciò permette al comportamento complessivo del sistema di emergere dall'interazione delle sue sotto componenti.

All'interno del sistema si è testata la presenza di una singola istanza di ogni agente (ad eccezion fatta per gli agenti client e robotici). Ciò non impedisce tuttavia di aggiungerne di nuovi, in quando la scoperta di questi viene effettuata runtime tramite un servizio di pagine gialle, che quindi permette una loro aggiunta in maniera flessibile.

L'unico punto contrario alla dichiarazione precedente potrebbe riguardare il Collection Point Manager e il Warehouse Mapper. Questi gestiscono infatti componenti fisiche e necessiterebbero di una ridondanza effettuata in maniera oculata.

Allo stesso modo, una gestione ottimale dell'ottenimento di informazioni richiederebbe qualche modifica nel codice client. Questa mancanza non influisce tuttavia sul funzionamento del sistema, quanto piú sulla 'user experience'.

Di seguito verrà fornita una descrizione della struttura degli agenti. A tal scopo risulta però necessario fornire una rappresentazione propedeutica dell'ambiente, definita tramite l'utilizzo di un'ontologia.

### 3.1.1 Ontologia del sistema

Un ontologia rappresenta un sistema in quanto tale, le categorie fondamentali che lo compongono e le relazioni fra esse. Questa è una componente importante per la rappresentazione della conoscenza ed è stata utilizzata nel sistema per descrivere e definire l'ambiente del magazzino e le operazioni ad esso relative.

Nello specifico, è stata definita un'ontologia relativa agli oggetti fisici o informativi ed alle loro relazioni (vedasi Figura 1); ed una ontologia relativa alle operazioni su di essi effettuabili (vedasi Figure 2 e 3).

Le entità fisiche rappresentate nel sistema magazzino sono gli *item*, ovvero gli oggetti ordinabili in esso conservati. Questi sono rappresentati da un *id* univoco e mantenuti, in differenti *quantity*, in scaffalature identificabili da una coppia *rack-shelf*.

Per quanto riguarda le entità puramente informative, si possono riscontrare i *command* e le informazioni relative agli *user* del sistema. Le prime sono relative al repository di conoscenza procedurale a cui gli agenti possono attingere; le seconde sono relative alle informazioni, quali l'*address* di spedizione, utilizzate durante la sottomissione di un *order*. Quest'ultimo contiene le informazioni relative all'ordine dell'utente, insieme al *collection point* (i.e., il punto di raggruppamento dei prodotti) e allo *status*. Si sottolinea infine come l'astrazione dell'order sia utilizzata dal solo agente *order-manager*, in quanto gli altri agenti accedono solo alle informazioni ad esso correlate.

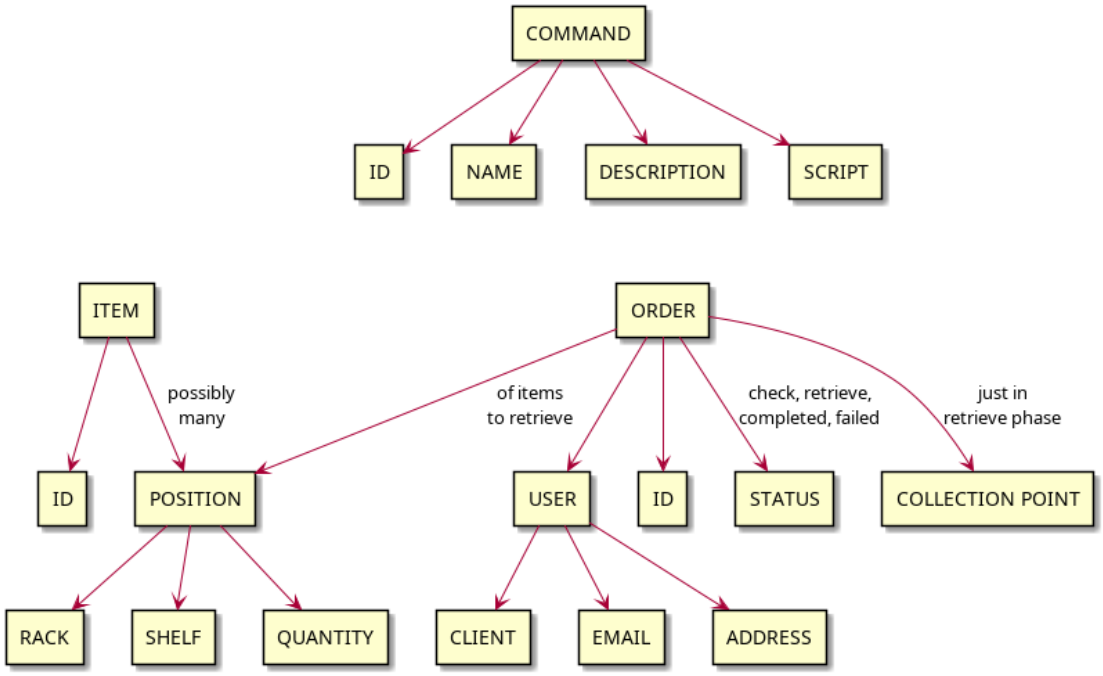


Figura 1: Schema relazionale delle entità fisiche e concettuali rappresentate nel sistema.

L'ontologia rappresentante le operazioni definisce invece le azioni che possono coin-

volgere determinate astrazioni del sistema e le informazioni per la loro esecuzione. Un banale esempio di ciò è l'operazione di piazzamento di un ordine, la quale necessita di mettere in relazione informazioni associate all'acquirente e gli oggetti richiesti da questo.

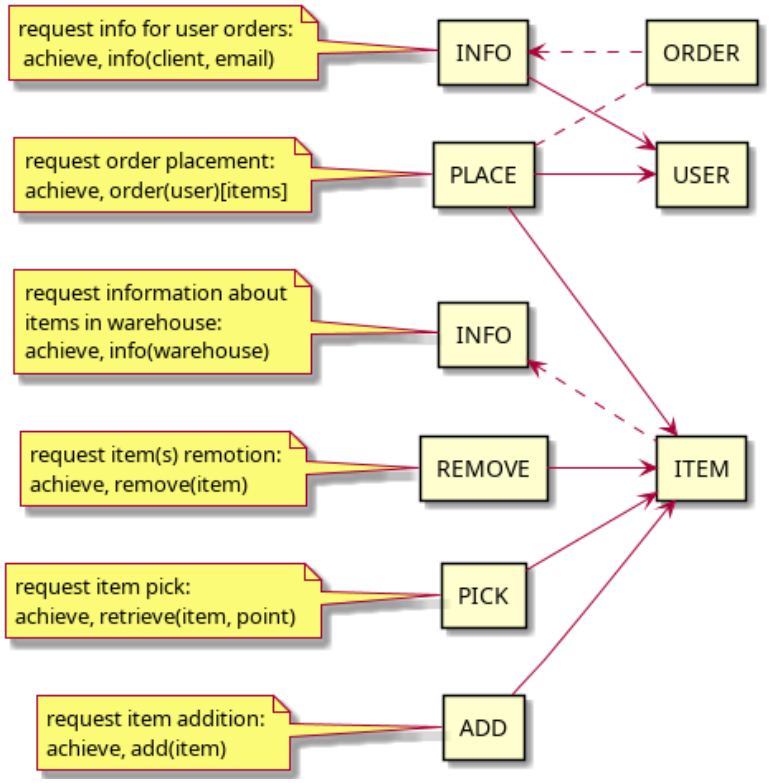


Figura 2: Parziale rappresentazione schematica delle operazioni (colonna sinistra) disponibili nel sistema e delle loro relazioni con le astrazioni (colonna destra) precedentemente definite. Questa parte mostra le operazioni relative ad ordini e prodotti.

### 3.1.2 Admin

L'agente admin è colui che si occupa di permettere l'interazione dell'amministratore col sistema. Condividendo alcuni tratti con l'agente user, si é deciso di raccogliere le caratteristiche comuni di queste due entitá tramite la definizione di un agente astratto dedito alla comunicazione col sistema (in esso, é denominato *Communicator*; vedasi Figura 4).

Le funzionalitá principali dell'agente admin sono relative alla gestione del magazzino e all'esecuzione di comandi nel sistema. Ricercando una separazione dei privilegi, questo non é in grado di effettuare ordini.

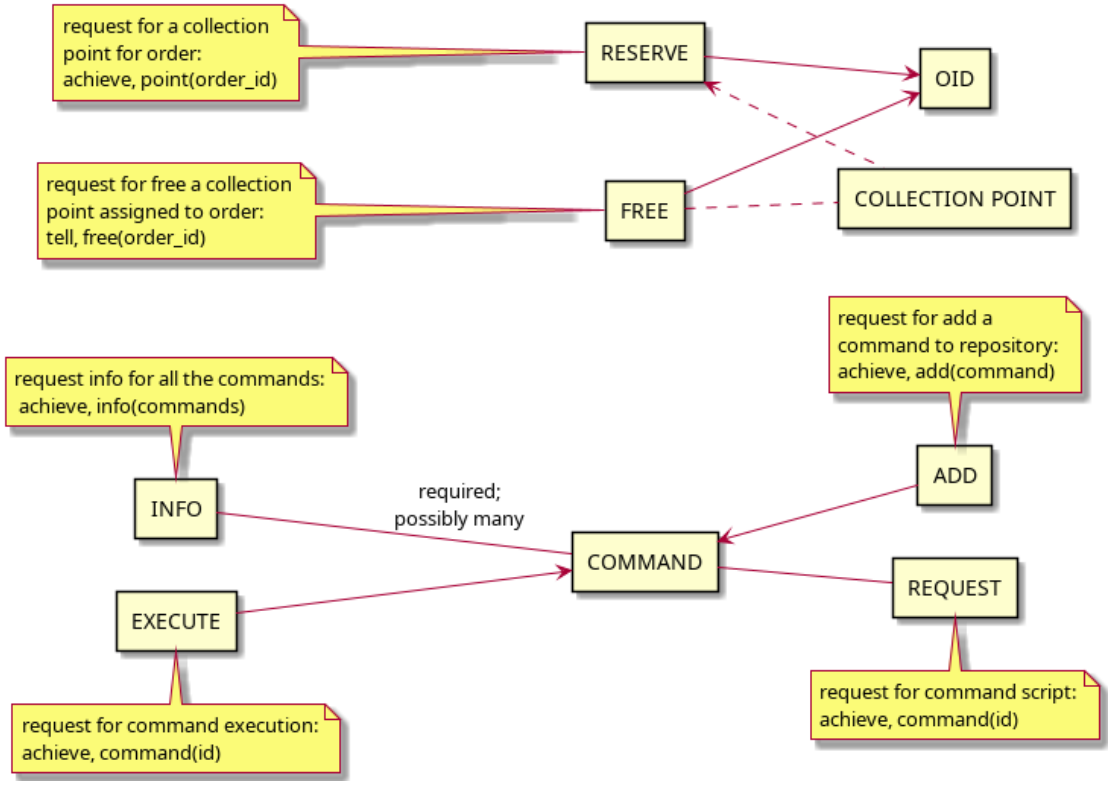


Figura 3: Parziale rappresentazione schematica delle operazioni disponibili nel sistema e delle loro relazioni con le astrazioni precedentemente definite (command, order id, collection point). Questa parte mostra le operazioni relative a comandi e punti di raccolta.

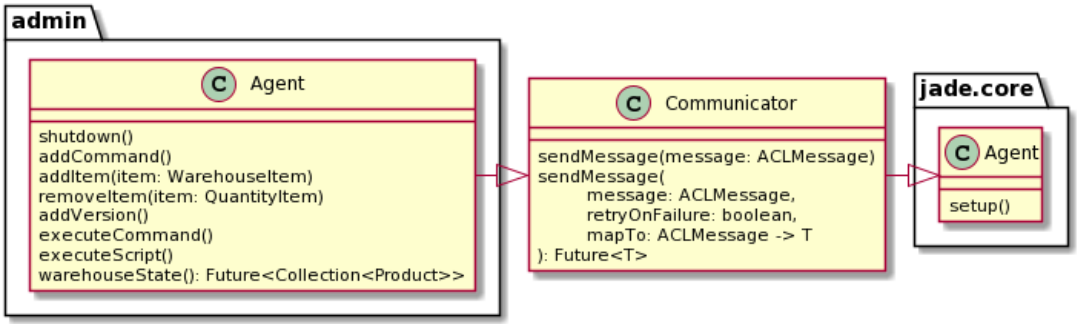


Figura 4: Diagramma delle classi dell'agente utilizzato per la comunicazione admin-sistema.

### 3.1.3 User

L'agente user si occupa di permettere l'interazione degli utenti/acquirenti col sistema. Come giá accennato, condivide alcuni tratti con l'agente admin: questi sono accomunati dall'estensione dell'agente *Communicator* (vedasi Figura 5).

Le funzionalitá principali dell'agente user consistono nella possibilitá di effettuare ordini e vederne lo stato. Non é invece dotato della capacitá di 'modificare/influenzare' direttamente il magazzino, in quanto si tratta di una capacitá disponibile al solo amministratore.

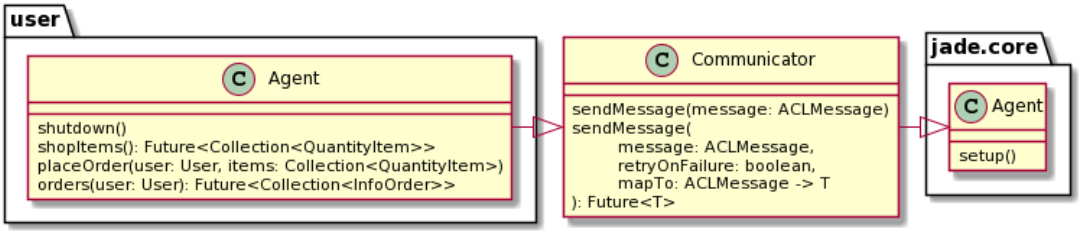


Figura 5: Diagramma delle classi dell'agente utilizzato per la comunicazione user/client-sistema.

### 3.1.4 Collection Point Manager

La struttura dell'agente gestore dei punti di raccolta é, come per molti altri agenti BDI del sistema, suddivisa piú in maniera logica che fisica. Escludendo il file principale dell'agente in cui si gestiscono import e registrazione al servizio di *Directory Facilitator*, i file che compongono la struttura dell'agente sono quelli che definiscono lo stato di conoscenza iniziale riguardo i punti di raccolta e quello che definsce le intenzioni per la gestione degli stessi e delle richieste entranti.

### 3.1.5 Command Manager

La struttura del command manager segue la stessa logica di quella del *collection point manager*. Esso é infatti suddiviso in un file che ne definisce le 'credenze' iniziali (soprattutto utile in fase di testing) e in altri che raggruppano le funzionalitá da esso fornite: aggiunta di comandi e ottenimento di informazioni su di essi.

### 3.1.6 Order Manager

La struttura dell'order manager é simile alle precedenti, senza peró essere inizializzato con uno stato. I file comprendenti le funzionalitá del seguente agente, sono quelli utilizzati per la generazione degli ID degli ordini; per la gestione delle richieste di informazioni sui loro stati; per quelle di piazzamento dell'ordine stesso e, infine, per la gestione della procedura di raggruppamento, che risulta invero piú complicata delle altre ed é stata valutata essere piú comprensibile in un file separato.

### 3.1.7 Robot Picker

Il robot picker é organizzato cercando di separare le funzionalitá di raccolta oggetti e quelle di esecuzione comandi. Le due funzionalitá sono invero collegate (non é ad esempio possibile un'esecuzione di un comando mentre é in corso il trasporto di un prodotto), ma una esigenza di chiarezza e pulizia del codice ha portato alla decisione di una loro separazione a livello di file.

### 3.1.8 Warehouse Mapper

Il gestore del magazzino prevede una struttura molto simile a quella del *collection point manager*: é infatti presente, in aggiunta a quelli di funzionalitá, un file che definisce lo stato iniziale. Oltre a questo, la logica dell'agente é suddivisa in files che determinano rispettivamente la funzionalitá di aggiunta e rimozione di prodotti dal magazzino, e quella di ottenimento di informazioni riguardo lo stato dello stesso.

## 3.2 Comportamento

Come già anticipato, gli agenti del sistema sono sviluppati tramite le tecnologie JADE e Jason. A causa di ciò, i modelli logici delle due tipologie di agenti differiscono, passando da un modello a *behaviour* ad uno *BDI*. Nello specifico, gli agenti sviluppati in JADE ed utilizzanti un modello a behaviour sono quelli relativi al client di interazione col sistema: Admin e User; gli altri agenti sono invece basati sul modello BDI.

Di seguito verranno descritti i behaviour e la logica di ogni agente del sistema. Inoltre, verrá descritto il comportamento dell'agente 'parziale' *Communicator*, il quale funge da struttura base per gli agenti del lato client.

### 3.2.1 Communicator

Quest'entità parziale permette una gestione delle possibili failure della rete. In pratica, questa si occupa del reinvio dei messaggi in caso non si riceva risposta.

L'alternaza dei behaviours é visibile nella Figura 6 e rappresenta i comportamenti necessari alla risoluzione di un possibile errore.

Si evidenzia inoltre come, in alternativa a questa forma '*reliable*' di comunicazione, sia sempre possibile utilizzarne una '*unreliable*' senza il reinvio dei messaggi.

### 3.2.2 Admin

L'agente admin non ha particolari behaviour aggiuntivi rispetto a quelli di controllo forniti dall'agente *Communicator*. Lo scopo di quest'entitá é infatti la traduzione dei messaggi dal formato basato su classi/oggetti utilizzato dal client a quello basato su *terms*, *literals*, etc. utilizzato dagli agenti del magazzino. Suo compito é infine anche la traduzione delle possibili risposte a questi messaggi.

Per chiarezza, intendiamo sottolineare come il suo compito sia la traduzione dei messaggi e delle risposte relativi alle sole funzionalitá disponibili all'amministratore. Que-

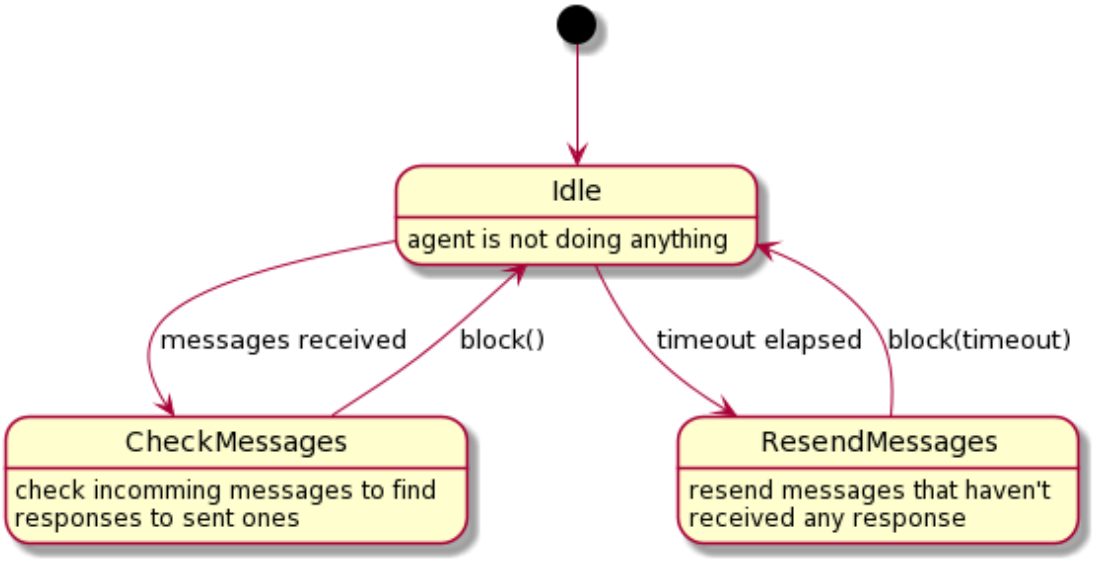


Figura 6: Diagramma degli stati del *Communicator*. Gli stati diversi da *idle* sono rappresentanti di behaviours dell'agente.

st'agente non risulta dunque in grado di comprendere le azioni e le entitá dell'ontologia utilizzate dal solo utente user.

### 3.2.3 User

Come per l'entitá admin, anche l'user non ha particolari comportamenti aggiuntivi. Il suo compito é infatti anch'esso riconducibile alla sola traduzione dei messaggi in ingresso ed in uscita.

Anche l'agente user assume un atteggiamento agnostico riguardo i formati delle comunicazioni dell'admin. Esso é infatti in grado di comprendere e tradurre le sole componenti dell'ontologia ad esso utili, e non é quindi pensato per la comprensione dei messaggi relativi all'amministratore. Questa scelta é frutto di una politica di separazione dei privilegi.

### 3.2.4 Collection Point Manager

Il comportamento dell'agente gestore dei punti di raccolta degli ordini é relativamente semplice nella sua comprensione e puó essere schematizzato dal diagramma degli stati visibile in Figura 7. Questo mostra chiaramente come l'agente permetta un'assegnazione di un punto di raccolta per uno specifico ordine quando si trovi in uno stato che ne presenti almeno uno non occupato. L'operazione di *free* si occupa invece di liberarne uno riservato ad un ordine giá completato.

Relativamente al diagramma a stati, si ritiene infine importante sottolineare come, trattandosi di un agente BDI, questo non implementi un behaviour per ogni stato in

maniera esplicita ma, piuttosto, questo schema comportamentale emerga dall'insieme delle 'credenze', 'desideri' e 'intenzioni' dell'agente stesso.

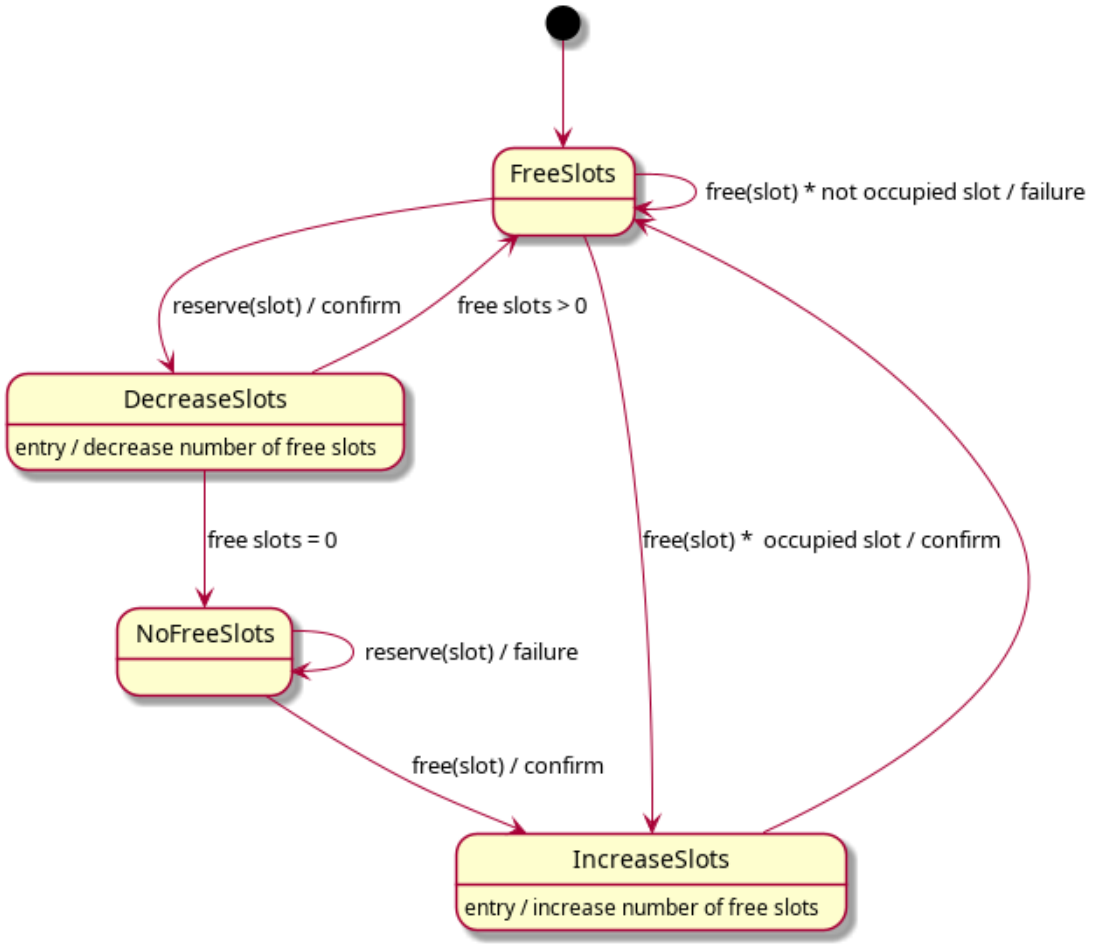


Figura 7: Diagramma a stati schematizzante il comportamento di massima dell'agente *Collection Point Manager*.

### 3.2.5 Command Manager

Il comportamento dell'entitá del sistema relativa alla gestione del repository di conoscenza e, quindi, dei comandi é forse quello meno complesso del sistema. Questo agente si limita infatti a ricevere le richieste e a fornire le risposte. Le possibili richieste a cui deve far fronte riguardano il salvataggio di nuovi comandi (per la gestione del quale deve semplicemente verificare che non esista un altro comando con lo stesso identificativo di quello che si tenta di aggiungere); l'ottenimento della lista dei comandi esistenti; la ricerca dello script relativo al comando richiesto.

In sintesi, si puó in un certo modo dire che questo agente si comporti come un repository/database remoto.

### 3.2.6 Order Manager

Questo é probabilmente l'agente BDI piú complesso del sistema insieme al *robot picker*. A causa di ció, il suo comportamento é difficilmente rappresentabile tramite un insieme di stati. Un modello piú semplice e rappresentativo é quindi dato dalla descrizione delle attivitá che intercorrono dalla sottomissione dell'ordine al suo completamento (o rifiuto): queste sono visibili in Figura 8.

Volendone sintetizzare la logica, questa passa dalla ricezione dell'ordine, alla richiesta di ottenimento dei prodotti, passando quindi per la richiesta di un punto di recupero per terminare infine con la raccolta degli oggetti.

É importante sottolineare come la gestione di un ordine non precluda una sottomissione e un completamento contemporaneo di un maggior numero di questi. Essi sono infatti indipendenti gli uni dagli altri nel processo.

### 3.2.7 Robot Picker

La descrizione del comportamento di quest'entitá, che si occupa del raccoglimento dei prodotti e dell'esecuzione dei comandi, si puó rivelare di diversa complessitá a seconda del livello di accuratezza richiesto. Se infatti lo schema base prevede un esecuzione dei due task indipendente, la capacitá di eseguire comandi causa una possibile esplosione del numero di stati all'infinito. Ogni comando potrebbe infatti essere definito da uno script complesso, avente a sua volta un suo diagramma a stati interno. A questo scopo si é quindi deciso di rappresentare la sola alternanza dei due task di raccoglimento ordini ed esecuzione di comandi (vedasi Figura 9).

### 3.2.8 Warehouse Mapper

Il comportamento di questo agente consiste nella gestione degli items nel magazzino. Esso ne mantiene la posizione, gestisce la rimozione e la loro aggiunta.

A livello di comportamento si tratta di un agente tutto sommato semplice in quanto, nella casistica piú complicata di rimozione, si limita a controllare che le quantitá di items siano corrette e ottenere le posizioni da cui essi devono essere recuperati (vedasi Figura 10). Piú semplice é inoltre la ricerca del primo slot contenente un item dello stesso tipo o, in caso un prodotto non sia ancora in magazzino, del primo slot disponibile per il suo storing (vedasi Figura 10). La funzionalitá che permette di fornire informazioni relativamente al magazzino consiste invece in una semplice lettura del 'database' interno, rappresentato invero dall'agente.

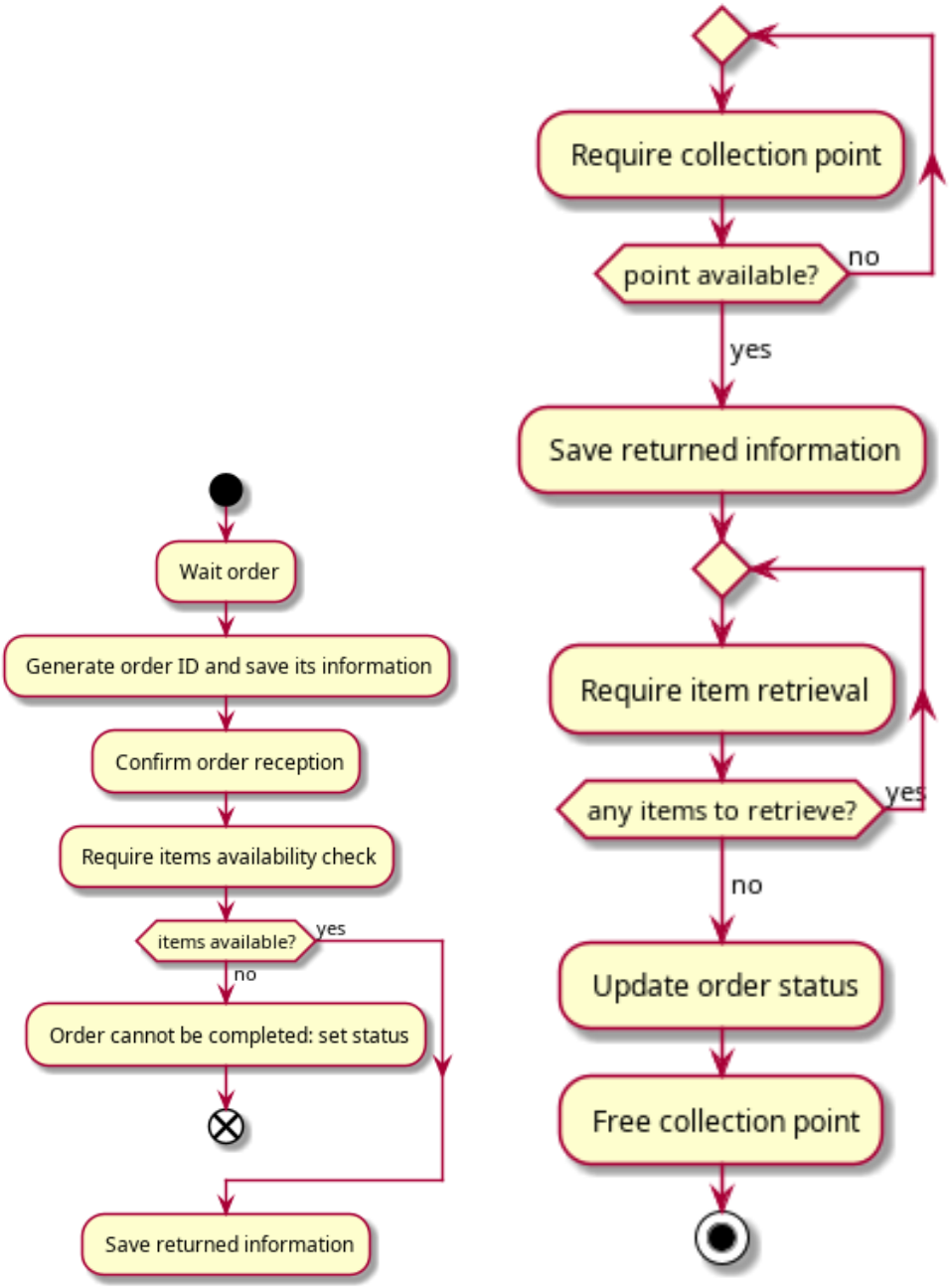


Figura 8: Diagramma delle attivitá relativo alle operazioni necessarie per la chiusura di un ordine. Si sottolinea la possibilitá per l'agente di gestire piú ordini contemporaneamente senza problemi di sorta. Il diagramma é spezzato per questioni di leggibilitá: la colonna sinistra continua in quella di destra.

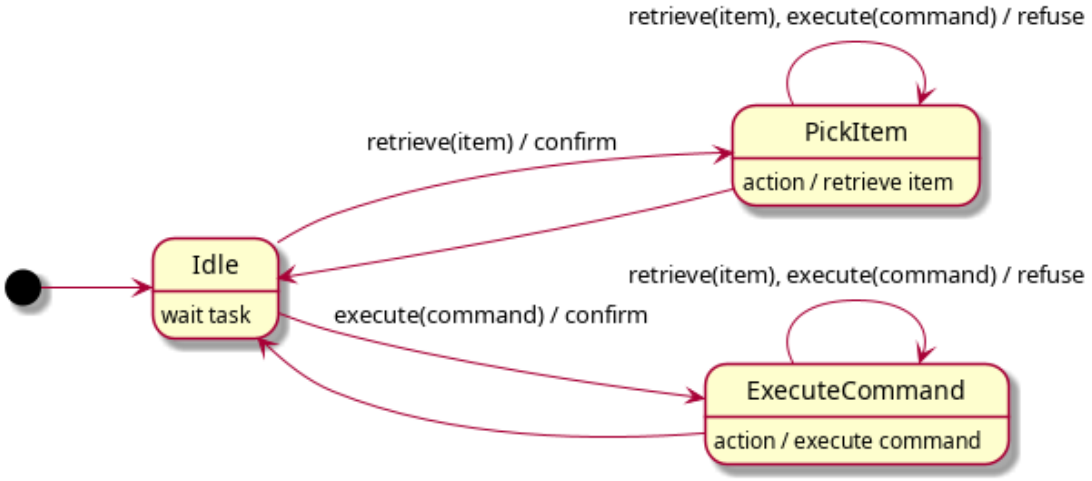


Figura 9: Diagramma degli stati di base dell'agente *robot picker*. Data la sua capacitá di esecuzione di comandi esterni, il diagramma degli stati varia a seconda del comando in esecuzione.

## 3.3 Interazione

La comunicazione è un aspetto fondamentale del sistema e determina il funzionamento dello stesso. L'analisi delle interazioni risulta perciò un elemento di fondamentale importanza nella descrizione del design e funzionamento di AMW.

Come già accennato, la comunicazione risulta in un certo qual modo complessa, in quanto richiede una conversione (effettuata automaticamente) da modello di comunicazione FIPA-ACL a KQML. Inoltre, questa deve anche tenere conto di possibili failure di rete.

Di seguito, le interazioni disponibili ed effettuate dai vari agenti saranno descritte ed analizzate.

### 3.3.1 Communicator

Abbiamo giá detto come questo agente si occupi del reivio di messaggi in caso di un supposto failure di rete. Dato che non é possibile diagnosticare un errore di questo tipo in maniera certa, questa caratteristica richiede che ogni messaggio inviato abbia un codice di conversazione univoco. Ció permette di identificare messaggi a cui si sia giá precedentemente risposto e che quindi richiedano soltanto un nuovo invio della risposta. Questa tattica é presente per la maggior parte delle trasmissioni del sistema; l'agente *Communicator* é tuttavia utilizzato soltanto per gli agenti JADE lato client.

Un esempio di gestione di failure é visibile in Figura 11.

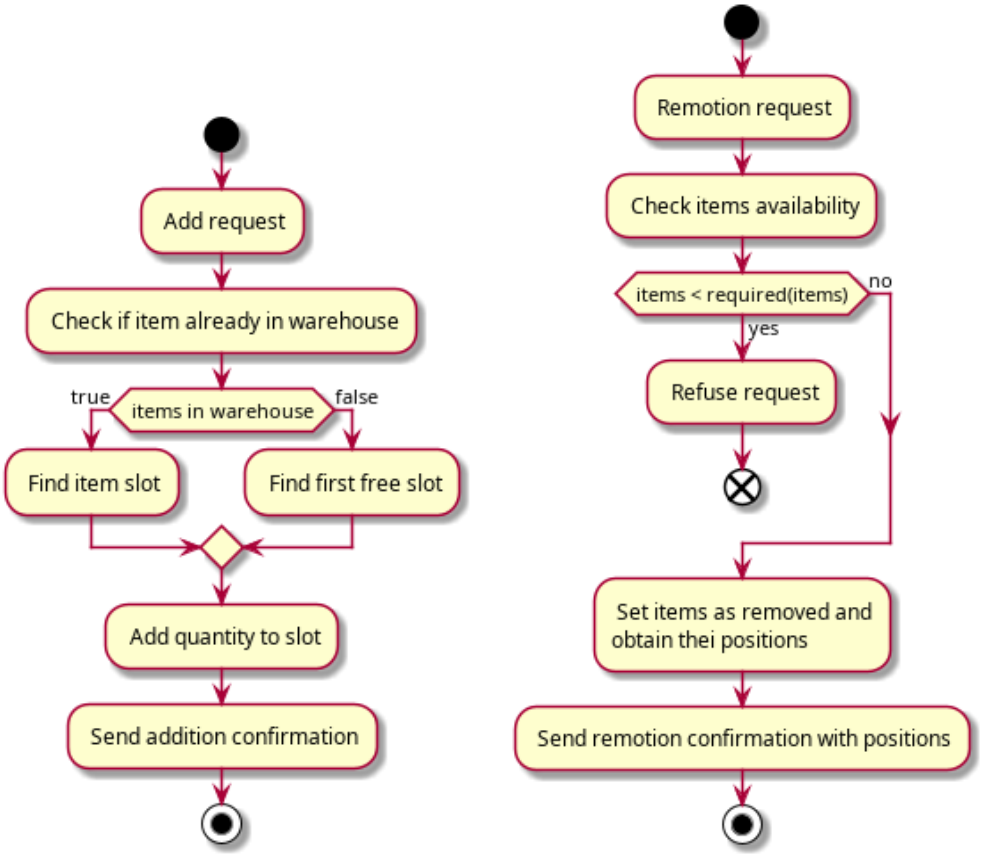


Figura 10: Diagrammi delle attivitá di aggiunta e rimozione dei prodotti dal magazzino.

### 3.3.2 Admin

Questo agente permette l'utilizzo delle funzionalitá di amministratore. A questo proposito, esso necessita di interagire con piú entità del sistema, fra cui: *warehouse mapper* per lo storing e la rimozione degli elementi dal magazzino; *command manager* per tutte le funzionalitá relative ai comandi. Inoltre, può potenzialmente dialogare con tutti gli agenti allo scopo di lanciare l'esecuzione dei comandi.

Un esempio relativo all'interazione necessaria all'esecuzione dei comandi é visibile in Figura 12.

### 3.3.3 User

Come giá detto il compito dell'user agent é relativo alla gestione degli ordini (vedasi Figura 13). A causa di ció, la principale interazione di questo é con l'agente gestore degli ordini *order manager* e con quello incaricato della gestione del magazzino, *warehouse mapper*, per la visualizzazione degli articoli disponibili.

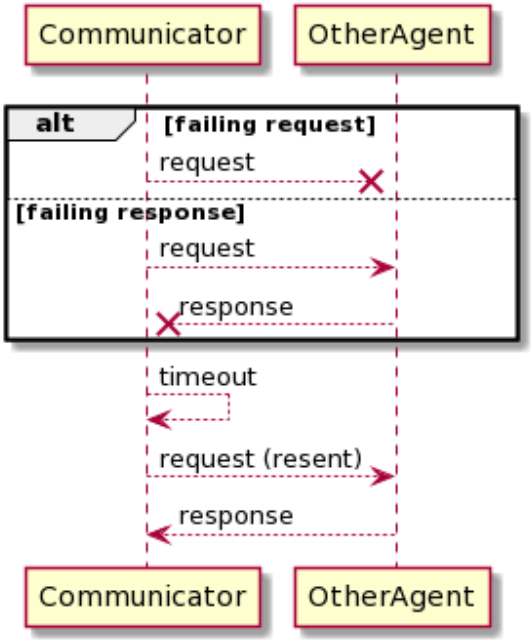


Figura 11: Esempio di gestione di failure di rete dell'agente *Communicator*. Come precedentemente detto, per evitare una doppia 'risoluzione' della richiesta, é necessario che ogni messaggio abbia un identificativo che ne permetta il riconoscimento da parte (nell'esempio) di *OtherAgent*.

### 3.3.4 Collection Point Manager

L'interazione del *collection point manager* avviene solo con l'*order manager*, il quale comincia la comunicazione con lo scopo di acquisire un punto di raccolta per gli ordini (vedasi Figura 13). Questa richiesta puó sia andare a buon fine, sia fallire in caso non ci siano points disponibili. In questo caso é dunque compito dell'order manager inviare una richiesta successiva per farsi assegnare uno dei punti di raggruppamento.

### 3.3.5 Command Manager

Questo agente interagisce potenzialmente con qualunque altro nel sistema, in quanto repository comune di piani (vedasi Figura 12). Allo stato attuale tuttavia, l'unico agente disponibile per l'esecuzione di comandi risulta essere il *robot picker*, il che implica che esso sia il solo a richiedere al *command manager* gli script da eseguire. Al di fuori del magazzino comunque, il repository dialoga con l'agente admin, permettendo a quest'ultimo di ottenere l'elenco dei piani disponibili.

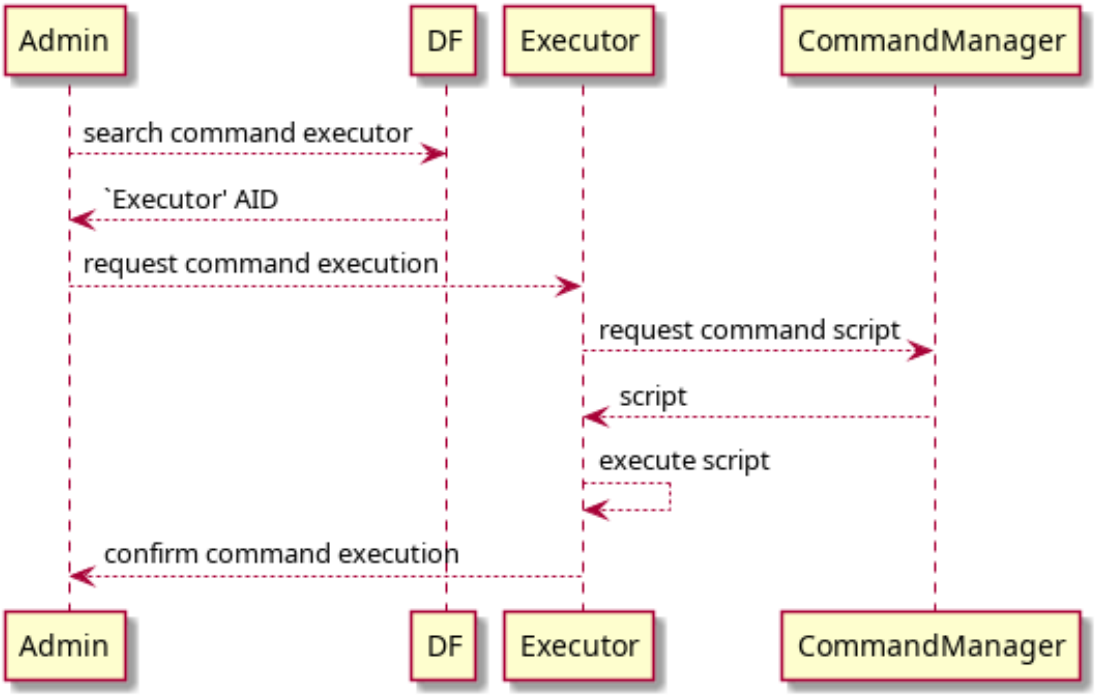


Figura 12: Diagramma di sequenza di una richiesta di esecuzione di un comando da parte dell'admin. In questo, si ignorano failure case per non complicare la descrizione, inoltre si assume che l'admin conosca in anticipo l'ID del comando da eseguire.

### 3.3.6 Order Manager

L'*order manager* é uno dei punti nevralgici del sistema. Questo si occupa invero di organizzare tutte le procedure necessarie al completamento degli ordini (vedasi Figura 13). Accetta innanzitutto le richieste di ordini dai clienti e le inoltra quindi al *warehouse manager* per la conferma di presenza dei prodotti. Si occupa poi di concordare col *collection point manager* un punto di raccolta per i componenti dell'ordine. A questo punto gestisce l'organizzazione dei *robot picker* per il raggruppamento dei prodotti. Infine, esso permette il salvataggio dello storico ordini e la visualizzazione dello stato di completamento di questi (vedasi Figura 14).

### 3.3.7 Robot Picker

Come giá anticipato, il *robot picker* ha due tipologie di funzionamento diverse: la raccolta di oggetti; l'esecuzione di comandi. Ognuna di queste due modalità richiede la comunicazione con agenti diversi del sistema.

La prima di queste accetta le richieste dall'agente gestore degli ordini e permette al robot di far parte del processo di completamento di uno di essi (vedasi Figura 13). In

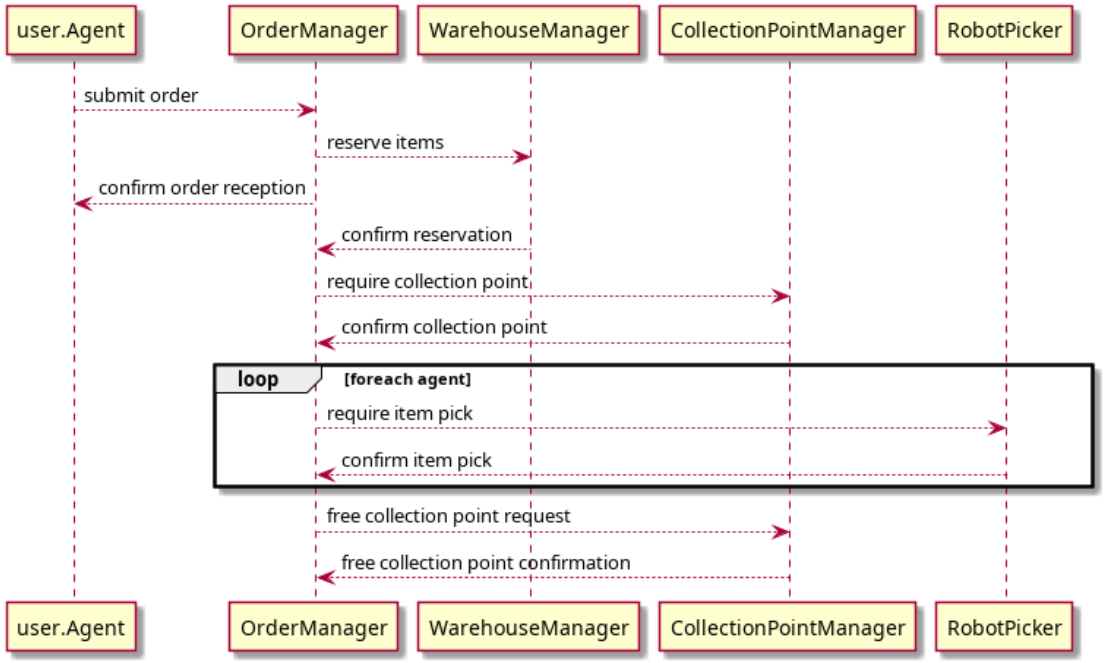


Figura 13: Diagramma di sequenza di una gestione di un ordine. In questo, si ignorano i failure case per non complicare la descrizione, inoltre si assume che l'user conosca in anticipo i prodotti presenti all'interno di un magazzino e che siano quindi ordinabili.

questa casistica il *robot picker* comunica col solo agente *order manager*.

Nella seconda modalità il robot dialogherá con due tipologie di agenti differenti: l'*admin* e il *command manager*. Dal primo otterrá infatti la richiesta di esecuzione, mentre dal secondo otterrá lo script ad essa necessario (vedasi Figura 12).

### 3.3.8 Warehouse Mapper

L'interazione di questo agente consiste sia in una componente informativa sia in una di esecuzione di task.

La prima di queste entra banalmente in gioco quando l'*user* richiede al gestore del magazzino le informazioni riguardo i prodotti disponibili. Allo stesso modo, questa interazione informativa ha luogo quando l'*admin* richiede le informazioni riguardo la distribuzione dei prodotti nelle varie scaffalature.

Il secondo tipo di interazione é invece relativo alla richiesta di verifica di disponibilitá e posizione di prodotti per la loro raccolta. In questa circostanza, il gestore del magazzino comunica invece col gestore ordini (vedasi Figura 13).

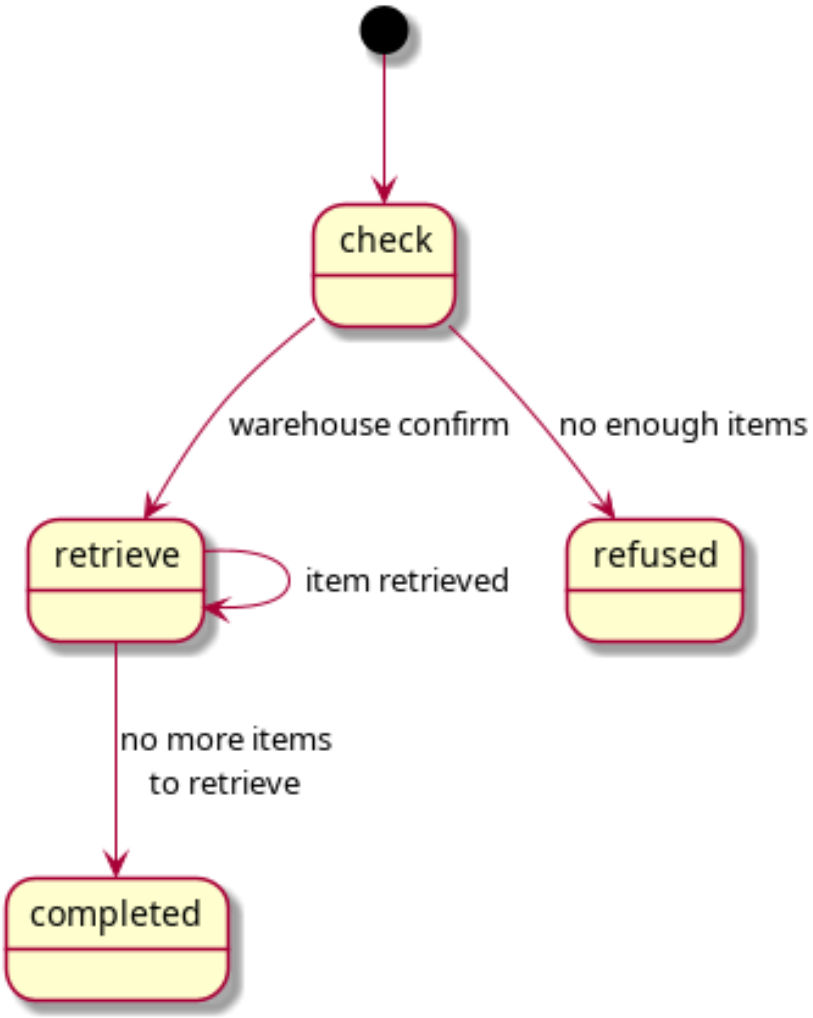


Figura 14: Diagramma dei possibili stati che un ordine puó assumere.

## 3.4 Considerazioni sul design scelto

In questa sezione si ritiene interessante analizzare alcune scelte di design e le loro conseguenze. Nello specifico, si descriveranno alcuni approcci inizialmente intrapresi ma poi scartati o modificati nel corso del progetto. Inoltre, si descriveranno alcuni ragionamenti successivi allo sviluppo.

### 3.4.1 Ontologia

Il design dell'ontologia differiva inizialmente da quello descritto precedentemente nel capitolo, principalmente riguardo la struttura dei comandi. Questi erano definiti da un

design molto piú complicato comprendente 'dipendenze' per l'esecuzione, 'interfacce' e versioni (vedasi Figura 15).

I molti problemi legati a questa struttura (si pensi ad esempio alla risoluzione delle dipendenze cicliche) hanno poi portato alla scelta di una semplificazione della stessa, in quanto troppo complessa per il livello richiesto per l'esame.

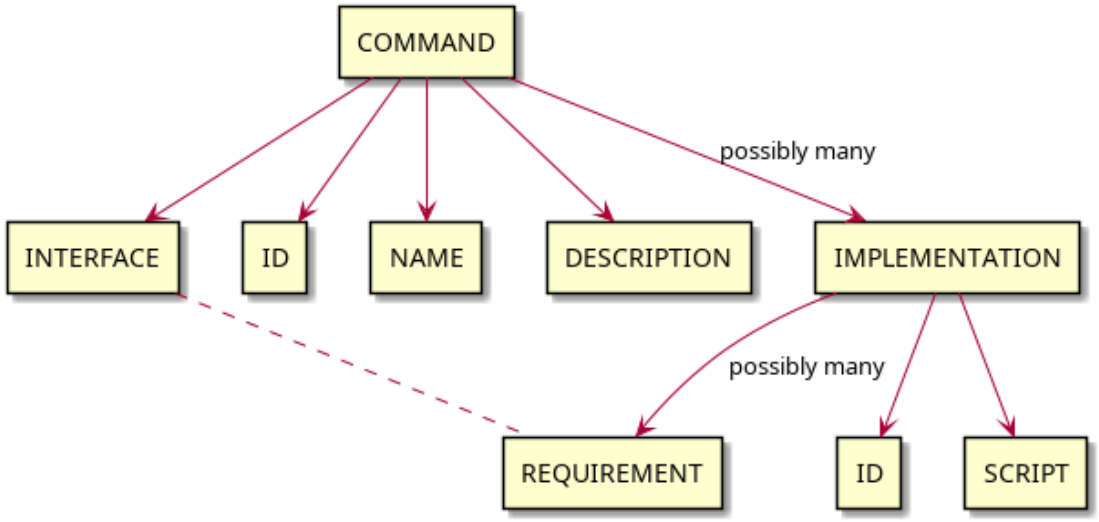


Figura 15: Design iniziale dei comandi. Ognuno consisteva di piú implementazioni, le quali potevano dipendendere da alcune funzionalitá che l'agente doveva conoscere o reperire tramite una ricerca nel repository di piani.

### 3.4.2 Il modello di comunicazione

Come giá detto, uno scopo di questo progetto era l'apprendimento dell'utilizzo del modello di comunicazione a scambio di messaggi, in alternativa a quello basato su spazi di tuple visto a lezione. Questo modello si é invero rivelato molto flessibile nel suo utilizzo, tuttavia si é anche notato come per alcuni task sarebbe probabilmente stato piú adatto un modello basato sullo spazi di tuple. In particolare, si é visto come tasks quali l'assegnamento di un oggetto da recuperare siano invero meglio rappresentati da un modello a spazio di tuple, dove gli agenti, una volta disponibili al compito, possano autonomamente accettare il task di recupero precedentemente pubblicato. Si nota infine come per altri task quali la sottomissione di un ordine, risulti invece piú coerente il recapito diretto dei messaggi al singolo agente interessato.

In conclusione, non si considera la scelta fatta un errore, ma si ritiene comunque la considerazione sopra effettuata interessante ai fini della corretta comprensione del sistema e dei suoi punti deboli e di forza.

# 4 Dettagli Implementativi

Il progetto é stato documentato tramite Javadoc all'interno del codice. É peró importante evidenziare come la documentazione sia relativa a solo alcune componenti pubbliche delle classi. Questa scelta deriva dalla convinzione, adottata da molti sviluppatori, secondo la quale un buon codice debba essere 'parlante' e non necessiti quidi descrizioni se non relativamente alle funzionalitá che possano essere chiamate esternamente e non siano immediatamente comprensibili.

## 4.1 Traduzione dei messaggi

Una funzionalitá che si intende descrivere é relativa al marshalling ed un-marshalling dei messaggi dal client JADE agli agenti Jason.

Come giá descritto, é stata definita una ontologia del sistema per la corretta descrizione delle operazioni e dei loro 'target'. La 'traduzione' dei messaggi non é peró logicamente parte dell'ontologia. Allo scopo di creare una struttura piú logicamente corretta si é quindi crata una classe di utilities utili alla trasformazione dei messaggi dal formato utilizzato all'interno del client (basato su classi/oggetti) a quello utilizzato dagli agenti Jason (basato invece su Terms, Literals, etc.). Queste utilities sono quindi richiamate ogni qualvolta sia necessario inviare un messaggio dal client ad un agente del magazzino o, viceversa, quando una risposta venga ricevuta. Uno schema della logica appena descritta é visibile in Figura 16.

## 4.2 Reinvio dei messaggi

Come giá detto, il sistema permette un reinvio dei messaggi in caso di un possibile errore di rete. Allo scopo di ottenere questa caratteristica é stato creato un behaviour (all'interno dell'agente di utility Communicator) che periodicamente si occupa di reinviare i messaggi per cui non si sia ricevuta risposta. Questa funzionalitá richiede quindi di salvare i messaggi di cui si sia tentato l'invio, insieme al loro timestamp, per permetterne un reinvio in caso una risposta non giunga entro il tempo limite. Il sistema si occupa dunque di considerare il parametro 'in-reply-to' per validare la ricezione della risposta ad un messaggio.

## 4.3 Formato degli script

Il sistema prevede l'esistenza di un repository di piani/comandi utilizzabili dagli agenti a richiesta. Questi sono, come giá detto, definiti da script che contengono *beliefs*, *desires* e *intentions* degli stessi. Relativamente all'implementazione di questo meccanismo di utilizzo di conoscenza procedurale, si sono dovuti tuttavia prendere alcuni accorgimenti. I piani aggiunti tramite gli script necessitano infatti di un processo di '*labeling*' allo scopo di permettere la loro rimozione al completamento del task. Questa etichettatura non avviene in maniera automatica (in quanto alcuni piani negli script potrebbero giá avere delle label assegnate) ma richiede la loro aggiunta da parte del programmatore (o admin, in questo contesto). Risulta quindi necessario aggiungere una label nel formato

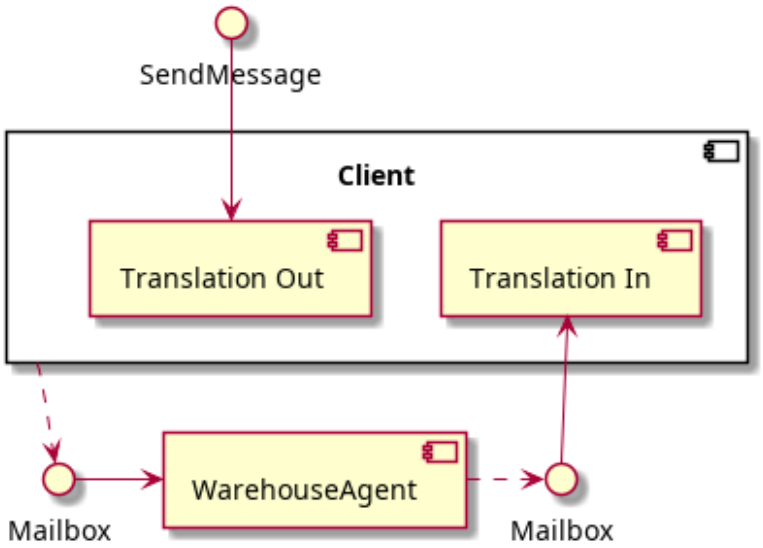


Figura 16: Rappresentazione schematica del processo di traduzione dei messaggi in ingresso ed in uscita. Quando un messaggio deve essere inviato dall'agente lato client, questo viene innanzitutto tradotto/marshallato dal sistema per poi essere inviato. Viceversa, alla ricezione di un messaggio, questo viene tradotto dal formato utilizzato nel magazzino a quello utilizzato dal client, per poi essere infine utilizzato dal sistema.

`@lN`, con `N` intero incrementale e $\in [0, \text{inf}]$, ad ogni piano definito nello script. Se ció non dovesse essere fatto, lo script verrebbe eseguito correttamente ma i piani non verrebbero rimossi alla terminazione dell'esecuzione, potendo causare problemi in futuro.

## 4.4 Raccolta oggetti

Un ordine prevede il raggruppamento di prodotti in un collection point. Questo task viene eseguito dal *robot picker* sotto richiesta dell'*order manager*. Si ritiene importante sottolineare peró come questo task di recupero sia invero solo simulato. Una sua effettiva implementazione richiederebbe infatti di conoscere una struttura hardware del sistema in esame e dei trasduttori da esso posseduti. Si ritiene che questa conoscenza esuli dal focus dell'esame ed é quindi stata data per conosciuta.

# 5 Autovalutazione / Validazione

Per la validazione del progetto non sono stati utilizzati specifici criteri formali, bensí un'insieme di test automatizzati e in grado di coprire una vasta quantitá di casistiche. Ció é stato principalmente dovuto alla modalitá di sviluppo scelta, seguente l'approccio **test driven**.

I test sviluppati mirano a valutare sia il funzionameto dei singoli agenti e delle loro funzionalitá di base, sia il funzionamento del sistema nel suo complesso. Lo sviluppo di questi é cominciato a seguito della definizione della logica di interazione del sistema ed ha quindi guidato lo sviluppo e la validazione di ogni singola entitá.

Volendo fornire qualche valore quantitativo riguardo ad i test effettuati, possiamo dire che sono stati creati ed eseguiti **116 test**, con una **coverage vicina al 100%** per gli agenti e le classi fondamentali (non sono ad esempio state testate le view e classi di scarsa importanza ai fini del progetto) e con **passing rate del 100%**. Essendo tuttavia gli agenti eseguiti su flussi di controllo diversi e non volendo bloccare il sistema a tempo indeterminato, é teoricamente possibile che alcuni test non passino. Ció non é peró dovuto ad un mancato funzionamento del sistema quanto ad una scelta implementativa del test. Questa possibilitá, seppur da tenere in considerazione, non si é comunque mai manifestata nel processo di validazione.

## 5.1 Il framework di testing

A causa della difficoltá di testing diretta degli agenti Jason, é stato necessario effettuarne una validazione 'a scatola chiusa', considerandoli come *black box* e valutandone il behaviour percepito esternamente e il cambio di stato tramite comunicazione. Ció é stato possibile grazie alla creazione di un piccolo framework di testing in grado di valutare il funzionamento sia tramite lo scambio dei messaggi (vedasi Listing 1), sia tramite i log prodotti ed attesi dagli agenti (vedasi Listing 2).

Lo sviluppo di questo é stato invero arduo, in quanto ha richiesto sia uno studio accurato dell'architettura JADE e Jason, sia un processo di *reverse engineering*. Il testing di un agente comprende infatti la comunicazione e non é quindi possibile testarne il funzionamento se esso non é il solo che possa manifestare comportamenti inattesi. A causa di ció é dunque ovviamente necessario che tutte le altre entitá coinvolte nel testing abbiano un comportamento deterministico. Ció é stato ottenuto tramite l'avvio dell'agente testato come agente Jason e la simulazione degli altri agenti coinvolti. La capacitá di far ció ha peró richiesto un *reverse engineering* del sistema in modo da sfruttare le classi e le funzionalitá non normalmente intese per l'utilizzo da parte degli utenti.

Listing 1: Esempio di test 'basato sulla comunicazione' permesso dall'utilizzo del framework.

```
@Test fun executionRequestShouldCauseARequestToCommandManager() =
    test {
        JADE.commandManager  // initialization of simulation agent
        agent .. REQUEST + """command(id("Command1"))""" - "123" >
```

```
            ASL.robotPicker  // send request to asl agent

        // expect (assertion) a request from the robotPicker agent
        JADE.commandManager <= REQUEST +
            """command(id("Command1"))[${mid(1)}]"""
    }
```

Listing 2: Esempio di test 'basato sul logging' permesso dall'utilizzo del framework.

```
@Test fun commandPropagation() = test(filterLogs = true) {
    robotPicker; commandManager

    recordLogs = true       // start to record system logs
    Thread.sleep(250)       // empirical df-registering time

    agent .. REQUEST + """command(id("Command1"))""" - "abc" >
        robotPicker         // request command execution

    // expected logs
    -"[ROBOT␣PICKER]␣request␣command␣execution"
    -"[COMMAND␣MANAGER]␣request␣command␣script"
    -"[ROBOT␣PICKER]␣command␣script␣obtained"

    Thread.sleep(2000)      // empirical execution time
    recordLogs = false      // stop to record system logs

    // expected execution confirmation
    agent <= CONFIRM + """command(id("Command1"))"""
}
```

# 6 Istruzioni per il deployment

La procedura di deployment utilizza sia il tool di automazione Gradle che quello di creazione ed esecuzione di immagini Docker. Se per il primo non é richiesta installazione, il secondo necessita invece di essere presente sul computer di avvio.

Gradle é stato scelto per la sua comoditá di utilizzo, necessitando solo di pochi comandi per eseguire il sistema. Docker é stato scelto sia per la possibilitá di dividere logicamente le entitá del sistema durante il testing su di una macchina, sia con lo scopo di creare un immagine che, teoricamente, possa semplicemente essere eseguita per aggiungere un nuovo agente al sistema.

Per ogni agente del magazzino viene prodotta un'immagine, la quale viene quindi eseguita per poi connettersi al container principale del sistema. In questa struttura, solo il main container JADE e il client di sistema non posseggono immagini specifiche. Per il primo, ció é dovuto alla inutilitá di creare un container docker per lanciare un jar. Per il secondo, la scelta é dovuta alla complessitá di creazione di un'immagine di un applicazione GUI.

Si ritiene inoltre importante sottolineare come il sistema non sia stato testato approfonditamente su macchine Windows. L'ambiente in cui é stato sviluppato é stato infatti un sistema Linux con la presenza di Java e Docker. Seppur i plugin utilizzati siano teoricamente funzionanti anche su Windows, non si da garanzia di ció.

Infine, per quanto riguarda il processo di build e avvio degli artefatti, si rimanda al README del progetto. In questo sono invero descritti tutti i comandi e le funzionalitá piú utili del sistema.

# 7 Esempi d'uso

Come giá descritto, il sistema permette l'utilizzo di due tipologie di utenti: User e Admin. In quanto non richiesto e non importante ai fini dell'esame, l'avvio del client nelle due modalitá é effettuato semplicemente tramite il passaggio o meno del flag 'admin' alla funzione di avvio `main`. Ció é possibile sia tramite l'utilizzo di un ide, sia tramite il comando Gradle `start_client -Padmin` (richiede che il main container sia giá in esecuzione).

L'interfaccia grafica non é stata studiata per essere responsiva ed é possibile riscontrare blocchi in attesa, ad esempio, della ricerca degli agenti nel *DirectoryFacilitator*. Inoltre, possibili failure come errori di rete sono visibili solo tramite l'analisi della riga di comando e dunque non visibili dall'interfaccia. Tutte queste semplificazioni alla GUI sono state effettuate in modo da concentrarsi principalmente sullo sviluppo dell'infrastruttura ad agenti senza perdere tempo nell'implementazione di funzionalitá grafiche di poco conto.

## 7.1 User

Relativamente all'User, i casi d'uso principali sono:

1. sottomissione di un ordine;
2. visualizzazione dello stato degli ordini passati.

L'interfaccia si avvia sulla schermata dello shop, dalla quale é possibile selezionare gli *items* richiesti e piazzare un ordine (vedasi Figura 17).

L'ordine é poi preso in carico dall'*order manager*, il quale fará partire la procedura di completamento degli ordini. É possibile visualizzare lo stato di quest'ultima nella schermata dello storico ordini (vedasi Figura 18).

Non trattandosi di un processo automatico, la pressione del pulsante refresh é infine necessaria per l'aggiornamento dei dati nella schermata.

## 7.2 Admin

Relativamente all'Admin, i casi d'uso principali sono:

1. visualizzazione dello stato del magazzino;
2. aggiunta e rimozione di *items*;
3. visualizzazione dello stato del *repository* di comandi;
4. aggiunta di un comando al *repository*;
5. richiesta di esecuzione del comando da parte di un agente del sistema preposto allo scopo (allo stato attuale, solo i robot).


Figura 17: Interfaccia User per la sottomissione di ordini.

L'interfaccia si avvia sulla schermata del magazzino, dalla quale é possibile aggiungere, rimuovere e visualizzare la disposizione degli oggetti al suo interno (vedasi Figura 19). L'aggiunta permette di inserire un oggetto in uno slot che che sia vuoto o contenga un prodotto dello stesso tipo. La rimozione permette invece di rimuovere una specifica quantitá di un elemento senza peró la possibilitá di specificarne la posizione. Cliccando su di un *rack* é invece possibile visualizzare i prodotti in esso contenuti (vedasi Figura 20).

La schermata dei comandi (visibile in Figura 21) permette la visualizzazione dei *tasks* salvati e presenti nel *repository* di piani; la loro modifica e successivo salvataggio; la possibilitá di richiederne l'esecuzione. Perché un comando sia eseguibile é necessario che questo abbia i *plans* che lo compongono preceduti da una label nella forma `@lN`, dove `N` rappresenta un indice incrementale che parte da 0. Questa 'etichettatura' é necessaria per la corretta esecuzione dei piani ed in futuro potrebbe essere possibile applicarla in automatico allo *script* sottomesso. Inoltre, per l'esecuzione é necessario che il piano sia stato precedentemente salvato nel *repository*.

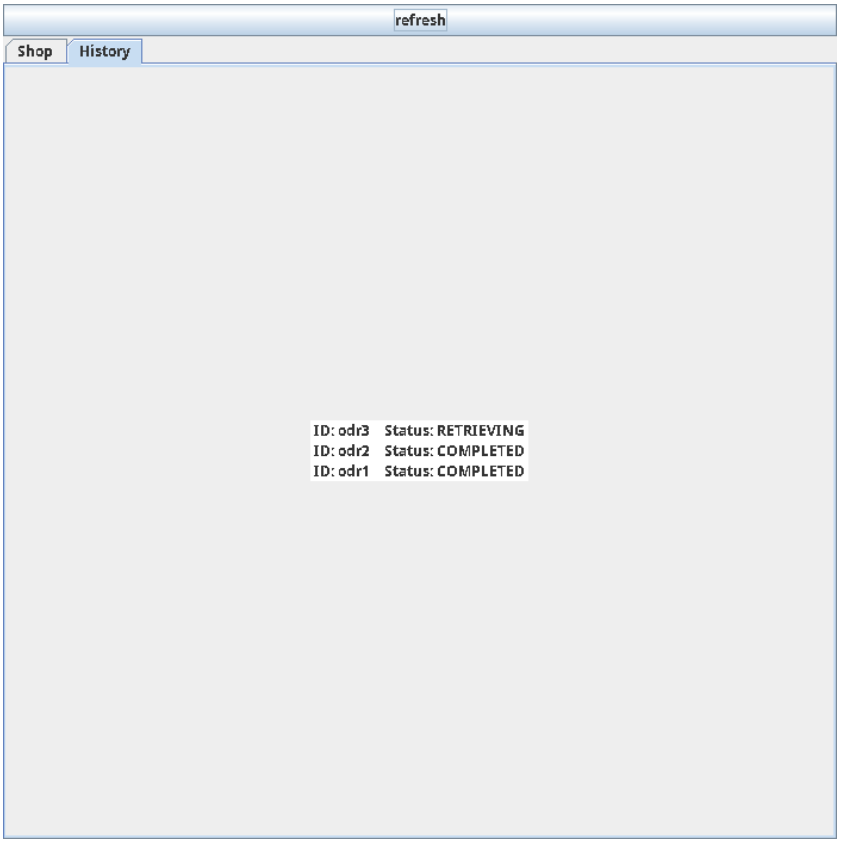


Figura 18: Interfaccia User per il controllo degli stati degli ordini.

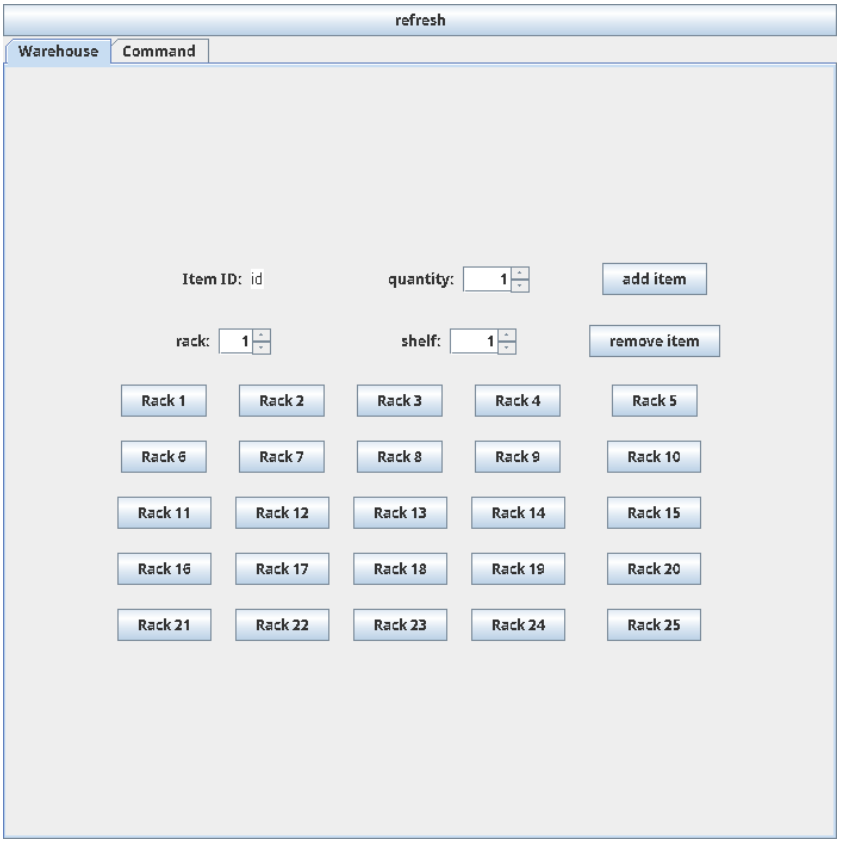


Figura 19: Interfaccia Admin per il controllo dello stato del magazzino.

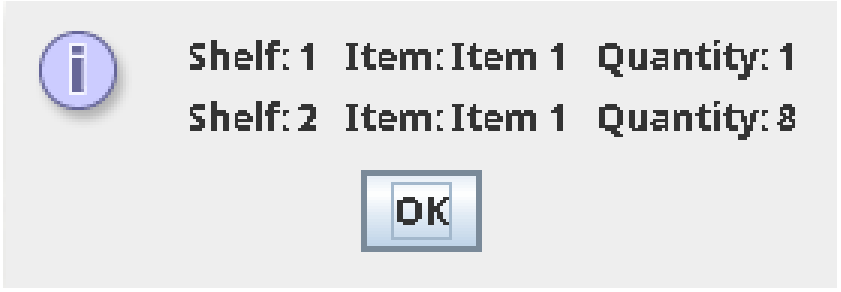


Figura 20: Dialog dell'interfaccia Admin per il controllo dello stato di un rack del magazzino.

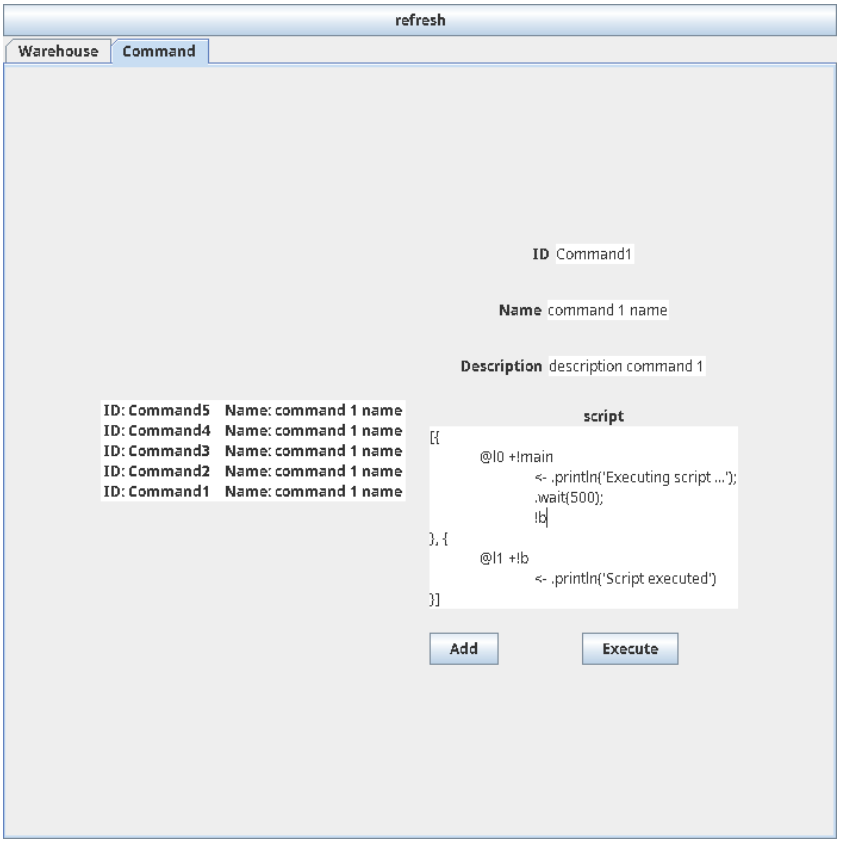


Figura 21: Interfaccia Admin per il controllo dello stato del *repository* di comandi.

# 8 Conclusioni

La realizzazione del progetto si é focalizzata sul raggiungimento degli obiettivi inizialmente definiti. Dato lo stato del sistema al momento della consegna, questi si ritengono interamente completati. Con riferimento al progetto, il sentimento é dunque di generale soddisfazione. Seppur con qualche imperfezione infatti, il raggiungimento totale degli obiettivi posti e la tutto sommato buona qualitá del sistema prodotto (seppur con tutte le sue semplificazioni) mi portano ad esprimere un sentimento positivo sul lavoro svolto. L'unico lato veramente negativo del processo é stato relativo alla realizzazione, che ha richiesto un ammontare di tempo ben superiore a quello previsto. Ció é stato dovuto ad una moltitudine di fattori, sia organizzativi ma soprattutto collegati a documentazioni non sempre esaustive e meccanismi di testing non ortodossi, che hanno spesso richiesto implementazioni ad-hoc e processi di reverse engeneering sui framework utilizzati.

La realizzazione del sistema ha dunque richiesto una notevole quantitá di lavoro. Questo é iniziato con un processo logico per identificare le entitá del sistema e capire come dovessero comportarsi e come dovessero logicamente interagire. Ció non é stato facile in quanto ha richiesto uno studio approfondito del dominio.

Il secondo punto dell'implementazione é consistito nello studio delle funzionalitá dei sistemi JADE e Jason su 'tests giocattolo' e documentazioni. Ció é servito per ottenere una corretta comprensione delle caratteristiche di questi. In questo processo ricade anche lo studio del paradigma BDI e di quello di programmazione logica, effettivamente mai visti prima dell'inizio del progetto.

Si é quindi passati allo sviluppo delle prime entitá del sistema che avevano giá uno schema logico chiaro e definito. In parallelo a questo processo si é dunque iniziato a definire alcune metodologie di testing. Queste sono invero state una delle parti di implementazione piú onerose (a livello di tempo) di tutto il sistema. Non si sono infatti trovate tecnologie adatte alla validazione delle entitá prodotte e, essendo lo sviluppo seguente il modello 'test driven', é stato necessario che queste fossero implementate in autonomia, in quanto fondamentali. Il processo di definizione dei test ha dunque richiesto un processo di reverse engeneering sui due framework (specialmente Jason) per riuscire a creare un interfaccia di testing quanto meno utilizzabile. Questo si é concretizzato nella creazione del 'framework di testing'.

A seguito di ció si é proseguito nell'implementazione del sistema e nella definizione della modalitá di deployment. Questa ha richiesto l'analisi e lo studio di diverse tecnologie (principalmente Gradle e Docker) e l'analisi di diverse possibilitá allo scopo di trovare una soluzione che fornisse una buona facilitá di utilizzo. Anche questa fase si é rivelata piú 'time consuming' del previsto.

L'ultima parte dello sviluppo é stata quindi orientata ad una risoluzione delle criticitá e debolezze che ancora caratterizzavano alcune parti del sistema.

## 8.1 Lavori futuri

Fin dal principio, questo progetto si é delineato come un sistema simulato, intenzionalmente crato con alcune semplificazioni allo scopo di focalizzarsi sulle caratteristiche

dei sistemi ad agenti per cui si era mostrato interesse. A causa di ció, futuri lavori sul progetto non sono attualmente previsti.

Tuttavia, possibili punti di interesse per futuri lavori esistono. Nello specifico, l'analisi della funzionalitá di ottenimento di conoscenza da un repository remoto risulta, in mia opinione, interessante. Come giá annunciato infatti, la complessitá del sistema ha richiesto una sua semplificazione in corso d'opera. Il piano iniziale risulta tuttavia comunque interessante e di possibile interesse per studi sulla capacitá di agenti di inferire autonomamente le conoscenze necessarie (e quindi possibilmente ad ottenerle) per lo svolgimento di compiti non inizialmente definiti dal programmatore. Un esempio basilare di ció potrebbe essere il riuscire ad inferire le necessitá di riuscire a muoversi e di 'afferrare' per un compito di trasporto oggetti. Il raggiungimento di capacitá di questo tipo sarebbe senza dubbio un notevole passo in avanti nelle potenzialitá del progetto.

## 8.2 Conoscenze acquisite

Seppur il progetto abbia alcune mancanze e debolezze dovute alla mancanza di tempo, questo ha permesso una notevole acquisizione di conoscenze. Alcune di esse verranno di seguito elencate.

### 8.2.1 Linguaggi e paradigmi di programmazione

L'apprendimento del modello di programmazione BDI é stato senza dubbio uno dei punti fondamentali del progetto. La scelta delle azioni da intraprendere e la loro combinazione per il raggiungimento di un obiettivo, a partire dalle conoscenze dell'agente, é un approccio allo sviluppo di applicazioni che risulta in un certo modo nuovo rispetto all'esperienza pregressa. Allo stesso modo, il linguaggio ASL permette un approccio di programmazione logico che, non essendo mai stato visto prima dell'inizio del progetto, ha permesso un notevole incremento della conoscenza e si é rivelato utile per la comprensione di corsi successivi.

### 8.2.2 Comunicazione

Come giá annunciato, lo sviluppo del progetto ha permesso una migliore comprensione delle differenze, dei vantaggi e degli svantaggi delle metodologie di comunicazione basate su scambi di messaggi e su spazi di tuple. Si é infatti notato come per alcuni task questi metodi possano risultare piú o meno indicati. Si sono quindi chiarite le casistiche in cui la scelta di un approccio rispetto all'altro possa portare a comunicazioni piú efficaci.

Sempre relativamente alla comunicazione, si é direttamente sperimentato il problema dell'imposibilitá di garantire contemporaneamente disponibilitá e coerenza in caso di partizionamento del sistema (CAP thorem). A questo riguardo é stato necessario effettuare una scelta. Il sistema si impegna dunque a garantire in primo luogo la coerenza dei dati a discapito della loro disponibilitá.

Con l'obiettivo di garantire comunque un certo grado di disponibilitá di servizio anche in caso di failure di qualche entitá del sistema, si é deciso di fare in modo che alcuni

agenti gestissero solo sottosezioni di questo. Un esempio é dato dal *collection point manager*. Da design infatti, la duplicazione di questo richiede che esso gestisca solo un sottoinsieme dei punti di raccolta (non gestiti da altri agenti). Questo permette di garantire coerenza (nessun punto puó essere assegnato a piú ordini) e un certo grado di disponibilitá (se un agente o la rete fallisce, il servizio puó continuare a funzionare, seppur senza la possibilitá di assegnamento dei punti dell'agente fallito). Altri agenti seguono lo stesso principio logico.

La presenza di riferimenti ad entitá ed agenti fisici nel sistema porta comunque alcuni problemi. La possibilitá di aggiunta di nuovi agenti é infatti talvolta limitata (si pensi ai *robot picker* o ad i *collection point manager*), causando una possibile inabilitá del sistema in caso nessuno di questi risulti raggiungibile. Questa si é rivelata una debolezza del sistema stesso che non é stato possibile risolvere.

### 8.2.3 I framework JADE e Jason

Ovviamente le conoscenze acquisite riguardo questi due framework sono state notevoli. Anche se molte funzionalitá non sono state necessarie (e quindi non utilizzate) per lo sviluppo del sistema, il suo design ha richiesto uno studio approfondito di queste due tecnologie. Si ritiene dunque di poter dire, grazie a questo progetto, di aver acquisito una buona esperienza e conoscenza riguardo entrambe.

### 8.2.4 Design di ontologie

Lo sviluppo del sistema ha anche richiesto lo studio delle ontologie. Non essendo mai state viste nel percorso di studi e data la loro indubbia utilitá nella rappresentazione della conoscenza, si ritiene l'abilitá acquisita riguardo ad esse ed al loro design molto utile e di valore. Queste rappresentano invero una conoscenza importante a prescindere dal sistema che ci si approccia ad implementare, non essendo infatti legate ad una specifica tecnologia.

### 8.2.5 Conoscenze generali sui sistemi ad agenti

Da un punto di vista piú generale, si ritiene di aver esteso ed acquisito competenze riguardo ai sistemi ad agenti. Ció risulta importante in quanto si ritengono questi una componente fondamentale dei sistemi informativi del futuro, se non addirittura attuali.

In sintesi, grazie a questo progetto é stato possibile approfondire e comprendere appieno gli aspetti trattati nel corso, con riferimento a questo nuovo paradigma di programmazione.

## Riferimenti bibliografici

[1] Hans Dockter, Adam Murdoch, Szczepan Faber, Peter Niederwieser, Luke Daley, Rene Gröschke, Daz DeBoer, Steve Appling, and Others. Gradle.

[2] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language.

[3] Jomi F. Hübner and Rafael H. Bordini. Jason.

[4] Docker Inc. Docker.

[5] Stefan Poslad. Specifying protocols for multi-agent system interaction.

[6] Telecom Italia SpA. Java agent development framework.