ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

TECNOLOGIE AD AGENTI PER PIATTAFORME DI GAMING: UN CASO DI STUDIO BASATO SU JACAMO E UNITY

Elaborata nel corso di: Sistemi Operativi

Tesi di Laurea di:
MARCO FUSCHINI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2016–2017
SESSIONE II

PAROLE CHIAVE

Unity3D

Agenti

Videogiochi

Middleware

Comunicazione

Ai miei genitori, a mia sorella, ai miei amici e a Federica,
che mi hanno sostenuto in questo percorso aiutandomi ad arrivare
fino in fondo

Indice

Introduzione

Negli ultimi decenni, le piattaforme di gaming hanno avuto uno sviluppo e un espansione notevole, diventando tecnologie all'avanguardia e toccando aspetti dell'informatica in uno spettro molto ampio, dalla programmazione concorrente alla computer grafica, dall'intelligenza artificiale ai sistemi distribuiti.

Oggi giorno le piattaforme ed engine per lo sviluppo di videogiochi, come Unity3D, vengono utilizzate anche per sviluppare software diversi dai prodotti video-ludici. Per esempio vengono sfruttate per la creazione di simulazioni real-time, come i sistemi di serious gaming, o applicazioni di realtà aumentata e realtà virtuale, come Hololens e Vive.

Indipendentemente da quale sia l'obbiettivo dell'utilizzo di tali piattaforme, un elemento importante per tutti questi tipi di applicazioni è dato da entità che devono esibire un comportamento autonomo e con un certo livello di "intelligenza". Ad esempio un BOT in un videogioco, una macchina in un sistema di simulazione del traffico, un batterio dentro una simulazione di sistemi biologici, ecc.

Un problema rilevante è dato dalla scelta di modelli, architetture e tecniche di programmazione più opportune per questo tipo di entità. Per dare una risposta a tale problema, la letteratura dell'Intelligenza Artificiale definisce un tipo di modello e architettura chiamato "ad agenti" e più in generale "Sistemi Multi-Agente".

In questa tesi è stato svolto uno studio in merito all'utilizzo di una specifica piattaforma ad agenti, chiamata JaCaMo, per la modellazione e il controllo di entità autonome in piattaforme video-ludiche e affini, prendendo come tecnologia di riferimento Unity3D.

Nel primo capitolo verranno introdotti i concetti fondamentali per lo sviluppo dei videogiochi e verranno spiegate brevemente le basi sull'utilizzo dell'ambiente di Unity3D.

Nel secondo capitolo si introdurranno le basi per il modello ad agenti e si introdurrà la piattaforma JaCaMo.

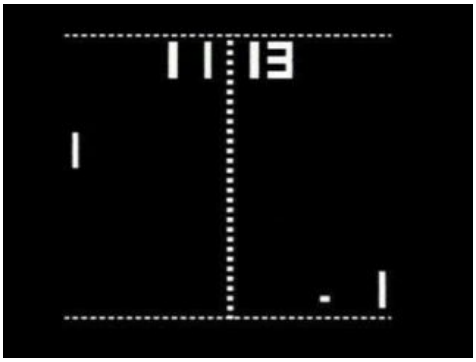
Nel terzo e ultimo capitolo verrà mostrato il processo di progettazione che si è seguito per poter far comunicare JaCaMo e Unity3D.

Capitolo 1

Ambienti virtuali

1.1 Sviluppo di ambienti virtuali

Negli ultimi 20 anni la tecnologia ha fatto progressi impressionanti. Di pari passo con gli aspetti industriali, anche il mondo dell'intrattenimento ha attinto dai progressi tecnologici, specie in campo video-ludico.
Basti pensare al primo videogioco famoso e paragonarlo a una controparte dei giorni nostri.



[Fig. 1.1]"Pong" - Rilasciato nel 1972 da Atari

[Fig. 1.2]"Overwatch" - Rilasciato nel 2016 da Blizzard

Oltre al comparto video-ludico fine a se stesso (a scopo di intrattenimento), si è pensato di sperimentare il videogioco come forma di educazione e apprendimento; è nato così il *Serious Game*

" Serious game (lett. "giochi seri") sono giochi digitali che non hanno esclusivamente o principalmente uno scopo di intrattenimento, ma contengono elementi educativi. Generalmente i serious game sono strumenti formativi e idealmente gli aspetti seri e ludici sono in equilibrio. Al centro dell'attenzione sta la volontà di creare un'esperienza formativa efficace e piacevole, mentre il genere, la tecnologia, il supporto e il pubblico variano. " [11]

Questa tipologia di videogiochi, trova applicazione in tutte quelle situazioni in cui si renda necessaria un'esperienza diretta per assimilare contenuti e comportamenti. Per esempio in campo militare o di pronto intervento, dove l'ambiente reale potrebbe mettere a rischio l'incolumità del "giocatore".

[Fig. 1.3] "Firefighting Simulator" - in fase di sviluppo da Astragon Entertainment

Inoltre gli strumenti per creare ambienti virtuali sono stati sfruttati anche per la creazione di ambienti di simulazione a scopo scientifico. In questo caso il motivo principale per l'utilizzo di ambienti virtuali è la ripetibilità dell'esperimento, così facendo si possono modificare arbitrariamente i parametri della simulazione senza il rischio di fattori esterni e imprevedibili che invalidino l'esperimento.

Cosa hanno in comune Serious Gaming, simulazioni scientifiche e videogiochi? Tutte e tre vogliono rappresentare un qualche aspetto del mondo reale in un ambiente replicabile e sicuro dal punto di vista dell'utente.

1.2 Entità virtuali

Che sia un videogioco o una simulazione scientifica, un ambiente virtuale è pressoché inutile senza delle *entità* che lo popolino.

Per entità si intende un oggetto che abbia esistenza autonoma ai fini dell'applicazione di interesse. Queste entità possono interagire tra di loro e con l'ambiente circostante; esempi di entità possono essere l'avatar del giocatore, gli oggetti che deve ottenere, gli eventuali nemici che lo ostacolino nella sua partita, ecc.

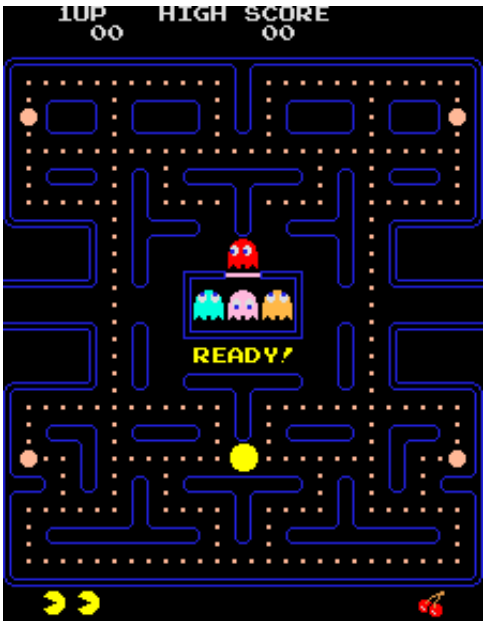
Si possono dividere concettualmente le entità in 2 tipi:

- Reattive
- Attive

Le entità **reattive** sono quelle che normalmente non cambierebbero il proprio stato, ma **reagiscono** a eventi esterni che, nella maggior parte dei casi, avvengono a seguito di un'azione delle entità attive.

Le entità **attive**, invece, sono quelle che **agiscono** per modificare il proprio stato e quello delle altre entità.

Per esempio, nel famoso gioco "Pac-Man" si possono osservare 4 classi principali di entità: 2 attive e 2 reattive.



[Fig. 1.4] "Pac-Man" - sviluppato da Namco

Le "palline" sparse per tutto il labirinto e le "caramelle" sono le entità reattive. Entrambe non si muovono e non mutano, ma quando il giocatore le tocca, le prime in-

crementano il punteggio, mentre le seconde donano un breve periodo di invincibilità al giocatore.

I fantasmi e il famoso Pac-Man invece sono le entità attive. I primi cercano di toccare il giocatore, riducendone le vite ed eventualmente ponendo fine al gioco; mentre Pac-Man è l'avatar del giocatore, con il quale cercherà di ottenere il punteggio più alto possibile.

1.3 Cos'è Unity3D

[Fig. 1.5]

Unity3D (da ora in poi chiamato semplicemente **Unity**) è un ambiente di sviluppo nato per creare videogiochi in 3D. Microsoft lo indica anche come ambiente di sviluppo per i suoi occhiali per la realtà virtuale (Hololens). Successivamente è stato esteso anche per creare giochi in 2D.

La sua espansione è sicuramente data dal fatto che il suo utilizzo è gratuito per piccole realtà (come neofiti che vogliono imparare a sviluppare videogiochi o piccole imprese), ma anche per la semplicità dell'organizzazione e gestione dei componenti.

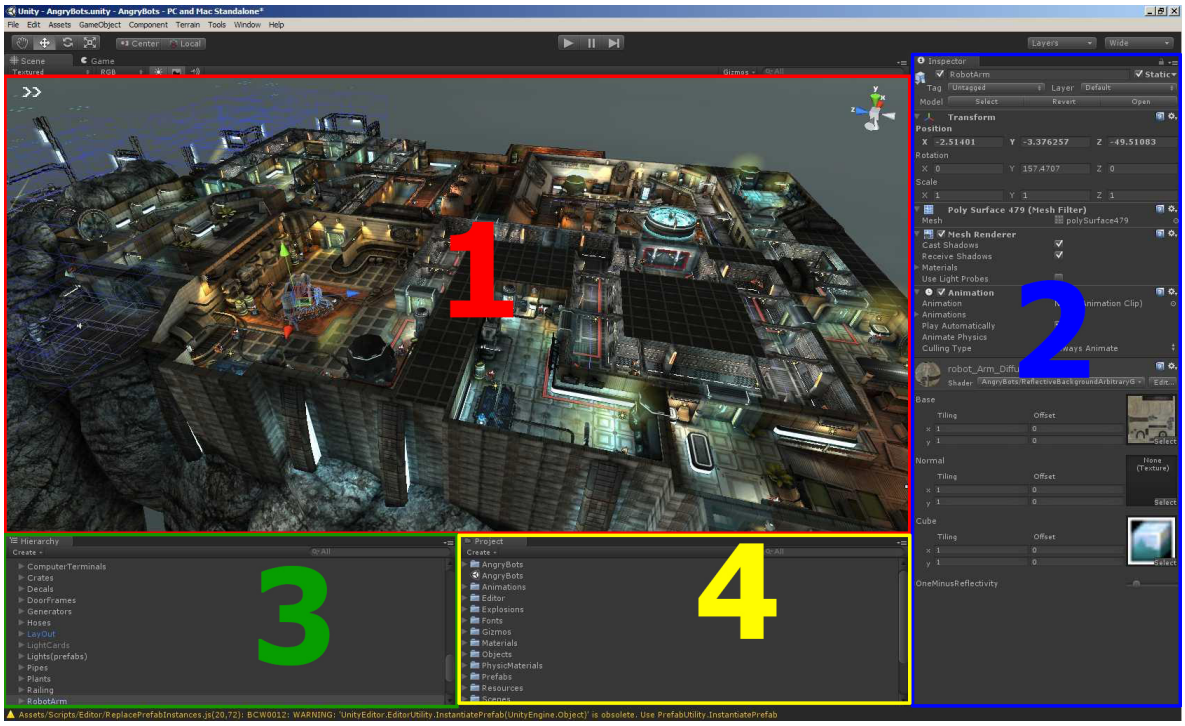
1.4 Struttura principale

L'intero sistema si basa sull'utilizzo dei *GameObject*. Le proprietà che definiscono il comportamento di un *GameObject* sono date dai suoi componenti interni, chiamati appunto *Components*. I *Components*, definiscono tutte le possibili caratteristiche dell'oggetto, per esempio la posizione spaziale, poligoni (la struttura visualizzabile dall'utente), *colliders* (che permettono di percepire se due oggetti si toccano), metodi di rendering (luci, ombre, riflessi, colori, ecc.), proprietà fisiche (come massa, attrito ed elasticità) e *scripts*(i contenitori delle logiche interne).

Inoltre ogni *GameObject* può essere composto a sua volta da altri *GameObject*, generando degli oggetti composti che permettono la creazione di strutture complesse, definendo per ogni oggetto dell'agglomerato i propri componenti.

Come un sito internet è suddiviso in pagine, un applicazione prodotta in Unity è suddivisa in scene. Ogni scena conterrà *GameObject* che potranno interagire tra di loro; sarà possibile anche cambiare scena all'avvenimento di determinati eventi, per esempio:

il menù principale di un qualsiasi videogioco è una scena di Unity, quando il giocatore premerà "nuovo gioco", la scena cambierà, lanciando quella relativa al primo livello di gioco. Al raggiungimento della fine del livello, si aprirà la seconda scena di gioco e così via.



[Fig. 1.6]

L'interfaccia utente è così divisa (rispetto alla numerazione di Fig.1.6):

1. Pannello *scena* - qui è possibile avere una visione 3D della scena e inoltre è possibile interagire con i vari elementi posizionati, cliccandoli, trascinandoli ecc. Ci si può muovere liberamente nella scena come se si possedesse un corpo virtuale, così facendo è possibile raggiungere tutti gli elementi anche se si fosse creata una scena molto estesa.

2. Pannello *proprietà* (*Inspector*) - qui vengono visualizzati ed è possibile modificare tutti i componenti collegati al *GameObject* selezionato (compresi i campi `public` degli script derivanti da `MonoBehaviour`).

3. Pannello *gerarchia* - qui sono presenti tutte le istanze dei *GameObject* presenti nella scena, vengono visualizzate in modo da capire quando un *GameObject* è composto da altri *GameObject*.

4. Pannello *progetto* - qui si trovano i file prodotti da/per l'applicazione, per esempio *script*, *prefab* (vedi sotto) e i file delle varie scene (aprendone uno si aggiornano tutti gli altri pannelli).

Una volta creato un *GameObject* e configurato a piacere, è possibile salvarlo come *prefab*. In questo modo è possibile istanziare più volte lo stesso *GameObject* senza doverne definire più volte gli stessi valori.

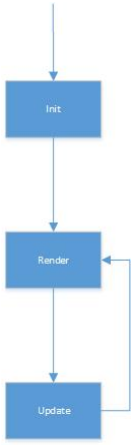
1.5 Logiche interne

1.5.1 gli *scripts*

Ogni *GameObject* può essere un'entità (re)attiva, cioè può cambiare il proprio stato autonomamente o col verificarsi di un determinato evento. La logica che determina quando e come modificare lo stato di un determinato *GameObject* è incapsulata negli script, che vengono poi "attaccati" agli oggetti su cui devono interagire.

Uno dei vantaggi di Unity è che gli sviluppatori non devono preoccuparsi del motore che muove l'intero sistema, concentrandosi solo su cosa devono fare i propri oggetti e quale deve essere l'azione scatenante.

Il motore che muove le applicazioni di Unity è quello classico per la creazione di videogiochi:



[Fig. 1.7] Loop

Prima di tutto il sistema chiama tutti i metodi *init* dopodiché inizia quello che viene chiamato *IL LOOP*, cioè chiama tutti i metodi *render* e poi tutti i metodi *update* in successione. Ogni *GameObject* estende la classe *MonoDevelop*, la quale implementa già i metodi *render* e *update* chiamati dal *LOOP*.

- *Render* è incaricato principalmente di fornire una rappresentazione grafica dell'oggetto in questione.
- *Update* serve ad aggiornare lo stato del relativo *GameObject*, per esempio muoverlo nello spazio, distruggerlo, ruotarlo, ecc.

Oltre ai classici render e update, gli sviluppatori hanno a disposizione dei metodi ben definiti per far agire i propri oggetti all'avvenimento di un determinato evento (es: alla creazione, alla collisione con un altro oggetto, ecc.)

Gli script possono essere scritti in 2 linguaggi: C# e Javascript. Ai fini di questa tesi, è stato adottato il linguaggio C#.

Gli script che si attaccheranno ai *GameObject* come componenti, devono derivare la classe `MonoBehaviour`.

Seguono alcuni dei metodi più utili (oltre al `update`) ai fini di questa tesi:

- `Start` - viene chiamato la prima volta che viene abilitato uno script (nella maggior parte dei casi, quando viene abilitato l'intero oggetto al quale è attaccato);
- `OnDestroy` - viene chiamato come ultimo metodo, prima che venga distrutto lo script;
- `OnCollisionEnter` - viene chiamato quando un altro GameObject entra in collisione (cioè tocca) con questo GameObject;

- `OnCollisionExit` - come OnCollisionEnter, ma avviene all'uscita della collisione;

- `OnCollisionStay` - viene chiamato a ogni frame finchè i due GameObject si toccano;

- `OnTriggerEnter/OnTriggerExit/OnTriggerStay` - stesso principio degli omonimi `OnCollision`..., la differenza sta se l'oggetto che collide è configurato come *Trigger* o meno (configurazione particolare per i componenti *collider*).

Per una consultazione di tutta l'interfaccia della classe `MonoBehaviour` riferirsi alla pagina uffciale dell'API di Unity https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

1.6 Intelligenza Artificiale nei Videogiochi

I videogiochi divertono perché pongono delle sfide ai giocatori, per le quali devono riuscire a **ingegnarsi** per poterle superare.

" ingegno - facoltà dello spirito di intuire, penetrare e giudicare le cose con prontezza e perspicacia; capacità inventiva applicata sia alla creazione di opere d'arte, sia all'esecuzione di opere anche manuali, sia a **trovare le vie, i modi e i mezzi per risolvere problemi, per eliminare le difficoltà che ostacolano la riuscita di un lavoro o di un'impresa** " [16]

Queste sfide, possono essere date dal design del livello, ma soprattutto da altre entità attive il cui scopo è ostacolare il giocatore. Queste entità devono poter "leggere" l'ambiente, lo stato del giocatore e lo stato delle altre entità, per **agire** secondo lo scopo per cui sono state programmate (es: uccidere l'avatar del giocatore).

" agire - Fare, operare, traducendo in azioni la parola o il **pensiero** " [17]

" pensiero - La facoltà relativa alla formazione di contenuti mentali " [18]

Quindi queste entità attive, per portare a termine il loro scopo, devono avere l'intelligenza necessaria a comprendere cosa sta succedendo attorno a loro e agire di conseguenza. Si parla in questo caso di *Intelligenza Artificiale (I.A.)*.

In Unity, per poter definire l'I.A. di un oggetto, è necessario scriverla con un paradigma ad oggetti dentro a 1 o più script.

Ma esistono modi più efficienti/eleganti per poter definire un I.A.?

Capitolo 2

Agenti

Negli anni l'I.A. sta diventando sempre più parte integrante della vita quotidiana. Non si pensi erroneamente che per I.A. si intendano solo le forme di vita digitali a cui il panorama cinematografico ci ha abituato.

[Fig. 2.1] " L'intelligenza artificiale (o IA, dalle iniziali delle due parole, in italiano) è l'abilità di un computer di svolgere funzioni e ragionamenti tipici della mente umana. " [12]

Quando parliamo di intelligenza artificiale, intendiamo quindi macchine (e programmi) capaci di "ragionare" e che possano fare scelte per portare a termine un compito alle quali

è stato assegnato. Queste scelte devono essere prese in base allo stato attuale dell'ambiente (e delle entità nel quale vivono).

I linguaggi di programmazione classici sono volti alla definizioni di algoritmi dove il programmatore deve prevedere tutti (o quasi) i possibili scenari che la propria entità può riscontrare; mentre un'entità dotata di intelligenza artificiale può decidere **autonomamente** le azioni da compiere per fare il suo lavoro.

A tale scopo, si stanno studiando nuovi *paradigmi* di programmazione dedicati all' I.A. tra i quali troviamo il *paradigma ad agenti*.

2.1 Paradigmi

" paradigma - un insieme di idee scientifiche collettivamente accettato per dare un senso al mondo dei fenomeni " [1]

Con l'evoluzione della programmazione si è sentita la necessità di distinguere fra diversi modi di intendere la programmazione e i paradigmi offrono una chiave di classificazione dei linguaggi. La più vecchia accezione di programmazione è stata chiamata *programmazione imperativa*:

" la programmazione imperativa è un paradigma di programmazione secondo cui un programma viene inteso come un insieme di istruzioni (dette anche direttive o comandi), ciascuna delle quali può essere pensata come un 'ordine' che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato. Da un punto di vista sintattico, i costrutti di un linguaggio imperativo sono spesso identificati da verbi all'imperativo (es: read, print, go) " [13]

In seguito è stata introdotta la *programmazione procedurale*, la cui peculiarità è il suddividere il codice sorgente in blocchi delimitati, chiamati *sottoprogrammi* o *subroutines*. Facendo ciò era possibile il riutilizzo parziale del codice, ottimizzando le risorse a disposizione.

Negli anni '60 si è introdotto il concetto di *programmazione funzionale*, dove l'esecuzione del programma non è più scandita da istruzioni, ma bensì da funzioni matematiche. Così facendo si ignorano gli effetti collaterali introdotti dalle funzioni.

Arrivando ai giorni nostri, il paradigma più diffuso è quello *a oggetti* (*O.O.P. - **O**bject **O**riented **P**rogramming*).

2.2 Cos'è un agente

Quando si progettano nuovi paradigmi, bisogna poter isolare le astrazioni su cui il paradigma si deve fondare:

- P. Procedurale - i sottoprogrammi
- P. Funzionale - le funzioni matematiche
- P. a Oggetti - gli oggetti

Nel caso dell'intelligenza artificiale era necessario trovare un'astrazione che modellasse un'entità che potesse compiere **autonomamente** e **proattivamente** azioni per portare a termine il proprio scopo, ma anche l'abilità di **reagire** alle situazioni in cui si trovasse e **interagire** con le altre entità.

È stato modellato così il concetto di *agente*.

" Agente - Che agisce, che provoca un determinato effetto " [10]

Le caratteristiche principali che un agente deve avere sono:

- Autonomia
  Le decisioni che un agente può prendere per portare a termine il proprio obbiettivo sono solo sotto il suo controllo.
- Proattività
  Il comportamento degli agenti è votato al raggiungimento di un proprio scopo, un obbiettivo.

- Reattività

  L'agente deve poter reagire ai cambiamenti dell'ambiente; deve poter cambiare i propri piani in risposta a eventi percepiti.

- Abilità sociali

  Gli agenti devono poter interagire tra di loro e scambiarsi informazioni.

Sono stati proposti diversi modelli concettuali per lo sviluppo degli agenti, ma uno dei più interessanti è il *B.D.I* (**B**elief **D**esire **I**ntetion).

2.2.1 Belief Desire Intetion

Concetto

" State aspettando il bus che vi dovrà portare in aeroporto: è in ritardo e iniziate a essere preoccupati dal tempo atmosferico. Decidete di abbandonare il vostro piano di viaggiare con il bus e chiamate un taxi. L'autista del taxi non conosce il terminal dal quale deve partire il vostro aereo e neanche voi. Mentre vi avvicinate all'aeroporto, telefonate al numero del centro informazioni stampato sul vostro biglietto informandovi che il vostro volo parte dal terminal 1. Il taxi vi lascia al terminal 1 e mentre entrate, notate le lunghe file presenti ai check-in. Realizzate che, non avendo un bagaglio da imbarcare, potete usare un punto di check-in express, risolvendo la questione in qualche minuto e ottenendo la carta d'imbarco. Procedete al check della sicurezza, ma il metal-detector suona. Realizzate di avere ancora in tasca il cellulare, lo togliete ma suona ancora. Togliete anche cintura e portafoglio, al che riuscite a passare. Notando che avete ancora tempo, comprate un caffè e un panino. Mentre mangiate il panino aprite il vostro laptop per controllare la mail, cancellate numerose mail di spam e controllate se ci siano nuove email importanti. Una voce vi segnala che il vostro gate apre. Quindi finite il vostro panino, chiudete il laptop e vi imbarcate. " [2]

Questo breve testo sarebbe un racconto su una normale partenza in aereo. Ma entrano in gioco numerose capacità umane che per una macchina non sarebbero poi così "normali".

Iniziare con un **obbiettivo** in mente (prendere l'aereo) e **decidere** un **piano d'azione** (prendere il bus). Constatare che il bus è in ritardo per poi scegliere di **cambiare piano** e

chiamare un taxi. **Non sapere** quale gate è relativo al vostro volo e decidere di venirne a **conoscenza** chiamando un altra entità (il centro informazioni) la quale sapete possedere l'informazione. **Scegliere** se fare la coda più lunga al check-in o adottare un metodo più veloce (express check-in) visto di possederne i requisiti. Sfruttare un attesa per poter **compiere altre azioni** (controllare la mail e mangiare un panino) non necessariamente utili alla vostra causa (prendere l'aereo), ma utili per alleggerire un possibile carico di lavoro futuro.

Il modello

Questo modello prende spunto dal modello comportamentale umano, cercando di replicare il ragionamento pratico degli esseri umani.

" Il ragionamento è il processo cognitivo che, partendo da determinate **premesse**, porta a una **conclusione**, facendo uso di **procedimenti logici**. " [14]

" Il ragionamento pratico è una questione di soppesare considerazioni conflittuali pro e contro opzioni concorrenti, dove le considerazioni rilevanti sono fornite da cosa l'agente desidera/valuta/importa e a cosa l'agente crede " [3]

Già solo analizzando il nome del modello ci si può fare un idea dei concetti base a cui fa riferimento:

- Belief (letteralmente *credenze*) - è la rappresentazione delle conoscenze di un agente, quindi le informazioni che ha del mondo sulle quali si baserà per le proprie decisioni.
- Desire (*desideri*) - sono gli obbiettivi che l'agente vuole portare a termine, ma non necessariamente agisce per compierli direttamente. Sono le potenziali influenze che un agente può avere. Notare che un agente potrebbe avere due o più desideri incompatibili tra loro.
- Intention (*intenzioni*) - sono le opzioni che l'agente ha a disposizione per compiere il suo lavoro. Possiamo considerarle una lista di piani d'azione da cui può scegliere.

Riassumendo: un **agente** vuole raggiungere il compimento di un proprio **desiderio**, attuando in successione una serie di **intenzioni**, scegliendole da una lista di piani d'azione in base a delle proprie conoscenze/**credenze**.

2.3 Paradigma ad agenti

Il paradigma ad agenti si basa sul concetto di *agente* come unità fondamentale. Però il caso di un sistema che utilizzi un solo agente è molto raro. Il caso più comune è un sistema che contenga molteplici agenti. I sistemi con un implementazione multi-agente, vengono chiamati *M.A.S. (**M**ulti **A**gent **S**ystem)*.

Questi MAS compiono un lavoro, sfruttando l'interazione tra gli agenti, i quali possono essere istanziati e distrutti *runtime* cioè con il sistema già in esecuzione.

In un MAS la forza risiede non solo nelle capacità di un singolo agente, ma anche nell'interazione e nello scambio di informazioni tra i vari agenti che lo popolano.

L'agente viene quindi implementato con la possibilità di memorizzare un insieme di conoscenze sotto forma di predicati, per esempio: `orarioDiPartenza(10.30)` indica che l'orario di partenza sarà alle 10:30 del mattino oppure `voto(Matematica,9)` significa che il voto in Matematica è 9.

L'agente può memorizzare i propri scopi/obbiettivi con la possibilità di aggiungerne di nuovi e togliere quelli completati/non più necessari/non più completabili.

L'agente possiede una lista (anche nulla) di piani d'azione che può essere ampliata runtime, da cui sceglie dinamicamente l'azione più congeniale al momento. La lista dei piani d'azione sono i piani (plans), mentre il piano scelto ed eseguito diventa l'intenzione.

Adesso che abbiamo definito gli agenti, possiamo notare una lacuna nel modello che è stato appena definito: non è stato ancora definito un modo per interagire con l'ambiente. Per questo motivo, allo stesso livello degli agenti, sono stati introdotti gli *Artefatti*.

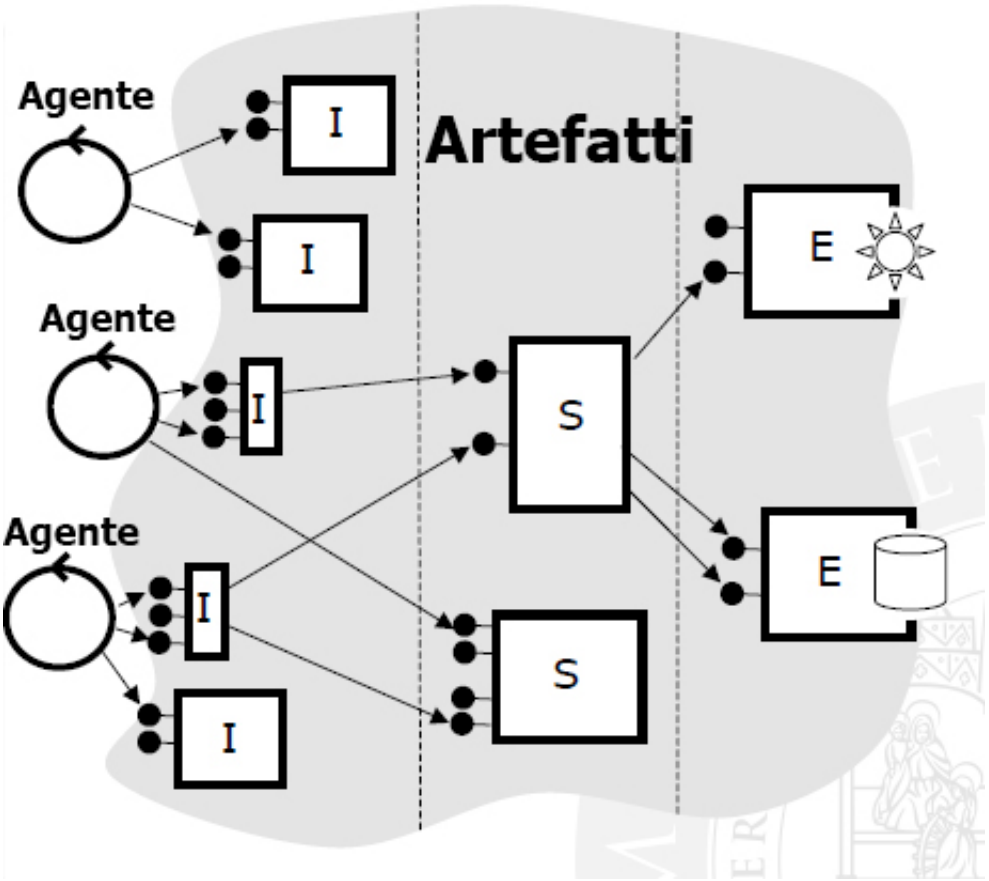
2.4 Artefatti

Se consideriamo un essere umano come agente, gli artefatti sono i surrogati degli oggetti che ci circondano: mouse e tastiera, schermi, archivi, tavoli, cellulari, trapani, ecc. Se non esistessero, la nostra esistenza si limiterebbe all'interazione con altri esseri umani senza la possibilità di interagire con l'ambiente e non avremo il modo di compiere azioni altrimenti

impossibili (parlare a chilometri di distanza, volare, vedere immagini di luoghi e persone poste all'altro capo del mondo).

La funzione degli artefatti è proprio questa: ampliare le funzionalità di un agente e la capacità di "cambiare il mondo" in cui risiede.

Se però gli agenti sono entità attive, gli artefatti sono entità passive, in attesa di servire richieste degli agenti o di altri artefatti che a loro volta sono stati interpellati dagli agenti.



[Fig. 2.2] Tassonomia a livelli

" Tassonomia a livelli

- Artefatti **Individuali** (I) - gestiscono una singola interazione di un agente
- Artefatti **Sociali** (S) - gestiscono le interazioni di molteplici agenti/artefatti
- Artefatti **Ambientali** (E) - gestiscono le interazioni tra il MAS e l'ambiente

" [4]

Gli artefatti forniranno delle interfacce per rendere possibile agli agenti di utilizzarli, nascondendo la reale implementazione agli stessi.

2.5 JaCaMo

Come caso di studio si è deciso di utilizzare un framework per programmazione multi-agente chiamato JaCaMo. Questo framework combina 3 tecnologie già affermate da diversi anni per lo studio e la produzione di software multi-agente. Le tecnologie che combina sono:

- **Ja**son - per la programmazione degli agenti
- **Ca**rtago - per la programmazione di artefatti
- **Mo**ise - per la programmazione dell'organizzazione multi-agente

Gli sviluppatori e ideatori del framework sono:

- Oliver Boissier - EMSE, Francia
- Rafael H. Bordini - INF-UFRGS, Brasile
- Jomi F. Hubner - DAS-UFSC, Brasile
- Alessandro Ricci - Università di Bologna, Italia
- Andrea Santi - Università di Bologna, Italia

Per l'implementazione di questa tesi sono stati sfruttati però solo la parte Jason e Cartago.

2.5.1 Jason per gli Agenti

Jason è un interprete del linguaggio per agenti, *AgentSpeak*, basato su *Java*.

" Jason è un interprete per una versione estesa di AgentSpeak. Implementa la semantica operazionale di quel linguaggio e fornisce una piattaforma di sviluppo per multi-agent systems, con molte funzionalità user-customisable. " [6]

Questa tesi non vuole coprire le basi su come programmare in AgentSpeak e più precisamente con l'interprete Jason, ma per chiarezza dei capitoli successivi viene mostrato qualche semplice esempio.

- Conoscenze - sono rappresentate da *literals*(cioè un insieme di caratteri con un valore fissato nel codice sorgente) con la seguente sintassi (semplificata):
  `predicato(arg1,arg2,arg3,...)` con un numero di argomenti ≥ 0.
  esempio:
  - `color(cube,red)`
  - `wake`
  - `loopCount(15)`

  Possono essere immagazzinate in *variabili* logiche. Queste variabili sono indicate da una lettera maiuscola iniziale, esempio:
  Persona indica una *variabile*.
  persona indica una *conoscenza* (che può essere salvata in una variabile).
- Obbiettivi - sono rappresentati da *literals* preceduti da un *!*.
- Piani - ogni piano è diviso in 3 parti:
  - evento scatenante - come suggerisce il nome, è la causa che fa prendere in considerazione il piano relativo. Può essere l'acquisizione (se preceduta da `+`) o la cancellazione (se preceduta da `-`) di una conoscenza o di un obbiettivo.
  - contesto - pone delle condizioni per le quali il piano può essere applicato oppure no, è composto da *literals*. Il caso più comune è il verificare se un agente ha o meno una data conoscenza.Viene però usato anche per richiamare determinate *conoscenze* mediante le *variabili*.
  - corpo - un insieme di istruzioni che l'agente esegue in successione. Queste istruzioni possono essere: *aggiunte* o *rimozioni* di obbiettivi e/o conoscenze, oppure comandi da chiamare su artefatti collegati a quell'agente.

  uniti dalla seguente sintassi:
  `evento_scatenante : contesto <- corpo`

  esempio:

```
+!sete: bicchiere(pieno) $<-$ bevi.
+!sete: bicchiere(vuoto) $<-$ !rimepiBicchiere.

+!riempiBicchiere $<-$ -bicchiere(vuoto); +bicchiere(pieno); +!sete.
```

`+!sete: bicchiere(pieno)` $<-$ `bevi.`
all'acquisizione dell'obbiettivo `!sete`, se il bicchiere è pieno, allora l'agente beve

`+!sete: bicchiere(vuoto)` $<-$ `!rimepiBicchiere.`
se all'acquisizione dell'obbiettivo `!sete` il bicchiere è vuoto, viene assegnato un nuovo obbiettivo `!riempiBicchiere`

`+!riempiBicchiere` $<-$ `-bicchiere(vuoto); +bicchiere(pieno); +!sete.`
all'acquisizione dell'obbiettivo `!riempiBicchiere`, si sostituisce la credenza `bicchiere(vuoto)` con `bicchiere(pieno)` e si torna all'obbiettivo iniziale `!sete`

2.5.2 Cartago per gli Artefatti

" CArtAgO (Common ARTifact infrastructure for AGents Open environments) è un framework / infrastruttura 'general purpose' che rende possibile programmare ed eseguire ambienti virtuali - detti anche ambiente virtual/application/software- per sistemi multi-agente. Cartago è basato sul meta-modello A&A (Agents & Artifacts)[...] gli agenti come entità computazionali che compiono qualche tipo di attività che mira a uno scopo e gli artefatti come risorse e strumenti costruiti dinamicamente, usati e manipolati dagli agenti per supportare/realizzare le loro attività. " [7]

Cartago è un framework specializzato negli artefatti, ma non è legato a nessun modello di agente o piattaforma. Nel caso dell'integrazione in JaCaMo, gli artefatti saranno implementati come *classi Java* che estendono la classe *Artifact* fornita da Cartago.

I metodi forniti dalla classe di un artefatto, potranno essere annotati come `@OPERATION`, indicando un metodo eseguibile da un agente che abbia l'accesso a tale artefatto.

Le operazioni di un artefatto, possono avere dei parametri di ingresso e dei parametri di uscita (quelli di ingresso sono forniti dall'agente all'artefatto, quelli di uscita dall'artefatto all'agente). La sintassi per un *operazione* è la seguente:

```
@OPERATION
public void nome_operazione(tipoParamIn inParam1,tipoParamIn inParam2,...,
                            tipoParamOut outParam1,tipoParamOut outParam2,...)
{
corpo dell'operazione...
}
```

dove i tipi dei parametri di ingresso sono gli stessi di Java(`int`, `String`, `float`, ecc.), mentre i tipi per i parametri di uscita devono essere "wrappati" nella classe
OpFeedbackParam< *tipoParametroUscita* >

dentro al corpo dell'operazione, i parametri di ingresso vengono utilizzati come classici parametri Java, mentre per impostare un parametro di uscita, bisogna chiamare il metodo *set* sul wrapper del parametro.
Per esempio se si volesse un'operazione che prenda due numeri in ingresso e restituisca la somma dei due numeri in uscita, il codice sarebbe il seguente:

```
@OPERATION
public void somma(int num1,int num2,OpFeedbackParam<int> somma){
somma.set(num1+num2);
}
```

2.5.3 Moise per l'organizzazione

" Moise è un modello organizzativo per multi-agent systems basato sulle nozioni di ruoli, gruppi e missioni. Abilita un MAS ad avere specifiche esplicite per la sua organizzazione. Queste specifiche sono usate sia dagli agenti per ragioni inerenti la loro organizzazione, sia da una piattaforma organizzativa che si assicuri che gli agenti seguano le specifiche. " [8]

" L'aspetto organizzativo e sociale delle agenzie, ha raggiunto un bel numero di lavori teorici in termini di modelli formali e teorie. Comunque, la concezione e l'ingegnerizzazione di proprie infrastrutture organizzative che incorporano tali modelli e teorie, sono ancora un problema aperto. L'introduzione di normative che interessano i requisiti di apertura e adattamento, sollecitano questo problema. Il meccanismo conseguente dalle infrastrutture correnti sembra non essere appropriato per gestire organizzazioni distribuite. C'è ancora il bisogno di astrazioni e strumenti che facilitino le applicazioni ad agenti a prendere parte nel monitoraggio dell'organizzazione da un lato e nell'adattamento e definizione dell'organizzazione nella quale sono situati dall'altro. " [9]

Moise permette di definire una gerarchia di ruoli con autorizzazioni e missioni, da assegnare agli agenti. Questo permette ai sistemi con un organizzazione forte, di guadagnare proprietà di apertura e adattamento.

" apertura - essenzialmente, la proprietà di lavorare con un numero e una diversità di componenti che **non è imposta** una volta per tutte " [5]

Fondamentalmente è la proprietà di un sistema, di non definire preventivamente i componenti che andrà ad utilizzare, ma può operare con componenti in continua evoluzione (numerica e funzionale).

Questa parte del framework JaCaMo non è stata utilizzata per l'implementazione della tesi, ma potrebbe comunque venire usata per la programmazione degli agenti.

Capitolo 3

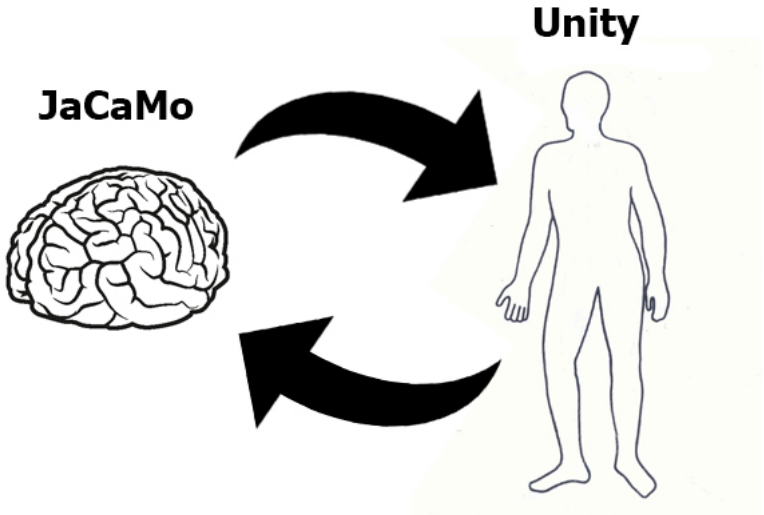
Agents On Unity

3.1 Obbiettivo

Esiste un ambiente di sviluppo molto diffuso, per la creazione di videogiochi e simulazioni, ma l'implementazione di una I.A. è ancora lasciato a modelli di programmazioni non specifici a tale scopo.

Esiste un ambiente di sviluppo basato su 3 piattaforme affermate per la programmazione ad agenti, ma il suo utilizzo è limitato ancora solo in ambito di ricerca.

Ancora non c'è la possibilità di far interagire questi due mondi; questo è l'obbiettivo finale della tesi, permettendo a un modello sviluppato apposta per l'I.A. di essere applicato a un contesto dove l'I.A è fondamentale.



[Fig. 3.1]

3.2 Progettazione

3.2.1 Idea di fondo

Unity viene utilizzato per sviluppare ambienti virtuali (principalmente 3D) e popolarli di entità. Queste entità possono essere viste come **corpi virtuali**.

JaCaMo invece è nato per programmare e gestire gli agenti, in questo caso possiamo considerarli delle **menti virtuali**.

Bisogna creare un canale di comunicazione per questi due "mondi": un *middleware*.

" In informatica con middleware si intende un insieme di **programmi informatici che fungono da intermediari tra diverse applicazioni e componenti software**. Sono spesso utilizzati come supporto per sistemi distribuiti complessi con architetture multitier. L'integrazione dei processi e dei servizi, residenti su sistemi con tecnologie e architetture diverse, è un'altra funzione delle applicazioni middleware. " [15]

Fondamentalmente un **corpo** deve eseguire **azioni** e a seguito di determinati eventi deve trasmettere le proprie **percezioni** alla mente, mentre la **mente** deve elaborare le **percezioni** per decidere quali **azioni** deve far svolgere al proprio corpo.

La prima domanda da porre è inerente alla cardinalità del collegamento fra corpi e menti:
Una mente, quanti corpi può gestire? E invece un corpo, da quanti menti può essere comandato?

La risposta è:
Una mente può comandare solo un corpo e un corpo è comandato da una sola mente.

In realtà questo vincolo è solo una semplificazione concettuale, come si vedrà in seguito; la comunicazione 1:N (da una mente a più corpi) è possibile e potrebbe servire nel caso di "corpi composti", ma renderebbe molto più pesante la progettazione delle logiche mentali e potenzialmente meno performante l'esecuzione delle I.A. Inoltre la componibilità dei *gameobjects* di Unity (un *GameObject* può essere composto da più *GameObjects*), rende superflua questa funzionalità.

La comunicazione N:1 (da più menti a un solo corpo) invece non ha risvolti pratici,

renderebbe solo più complicato descrivere il comportamento che dovrebbe avere un corpo dividendo le logiche mentali su più menti. Inoltre, se per qualche ragione risultasse necessaria la suddetta funzionalità, siccome gli agenti comunicano tra di loro, basterebbe rendere un agente "responsabile" di tutte le azioni di un corpo, mentre le decisioni potrebbero comunque essere prese da un agglomerato di agenti.

Un'altra domanda, meno immediata della prima, è relativa alla distribuzione dell'intero sistema:

Il sotto-sistema ad agenti e l'ambiente virtuale, risiedono nella stessa macchina o sono distribuiti su macchine diverse?

La domanda subito successiva è:

Possono essere presenti più sotto-sistemi ad agenti paralleli e/o più ambienti virtuali?

La risposta, in prima battuta, è: **dipende**.

Lo scopo di questa tesi è riuscire a fare interagire un MAS con un ambiente virtuale, permettendo a sviluppatori di videogiochi/simulazioni di inserire nei loro "mondi virtuali" un implementazione di un modello ad hoc per l'intelligenza artificiale. Quindi, per lasciare la possibilità di queste decisioni agli sviluppatori delle applicazioni finali, si dovrebbero porre meno vincoli possibili a tale scelta, strutturando l'intero sistema come se si potessero avere più sotto-sistemi ad agenti e più ambienti posti su macchine diverse.

Ragionandoci su, però gli scenari N:N (più ambienti collegati a più MAS), possono essere ricondotti a scenari con 1 solo sistema ad agenti collegato a 1 solo ambiente.

Possiamo osservare 4 casi generici, considerando il collegamento *JaCaMo:Unity* :

- **1:N** - 1 MAS può comunicare con più ambienti virtuali.

  Dividendo il "mondo virtuale" in più sotto-ambienti si perderebbe la possibilità di avere le interazioni tra le entità presenti in un "pezzo" con quelle presenti nell'altro. Se si avesse necessità di avviare in parallelo 2 "mondi virtuali" che non hanno bisogno di interagire tra di loro, non esisterebbe la necessità che siano accomunati dallo stesso "sistema di menti". Se per qualche ragione si avesse comunque questa necessità, basterebbe implementare un sistema di scambio di informazioni tra 2 MAS che sarebbe una funzionalità non dipendente dal middleware sviluppato in questa tesi, tornando a 2 sistemi 1:1 che possono scambiare messaggi tra loro sul livello degli agenti.

- **N:1** - Diversi sistemi MAS che controllano le entità di 1 singolo ambiente virtuale.

Potrebbe volersi implementare un sistema del genere, per scaricare il carico computazionale dei MAS su un sistema distribuito, dove ogni MAS dovrebbe essere presente su un nodo diverso del sistema distribuito. Così facendo si perderebbe la possibilità di comunicazione tra agenti presenti su due MAS diversi, se non implementando una funzionalità di comunicazione come al punto sopra. Se disponibile la funzionalità comunicativa appena citata, allora si potrebbe comunque avere un sistema 1:1 dove il MAS collegato all'ambiente virtuale abbia tutte le entità che possano comandare dei corpi virtuali e possa comunicare con altri N MAS distribuiti, i quali presentino le logiche "pesanti" che hanno reso necessario una distribuzione del carico computazionale.

- **N:N** - Più sotto-sistemi ad agenti che comandino corpi virtuali presenti in diversi ambienti scollegati tra loro.

  Rimangono valide le ipotesi fatte per i due punti precedenti, quindi se fosse necessario implementare un sistema N:N, con qualche accorgimento sarebbe comunque possibile implementarlo come sistema 1:1.

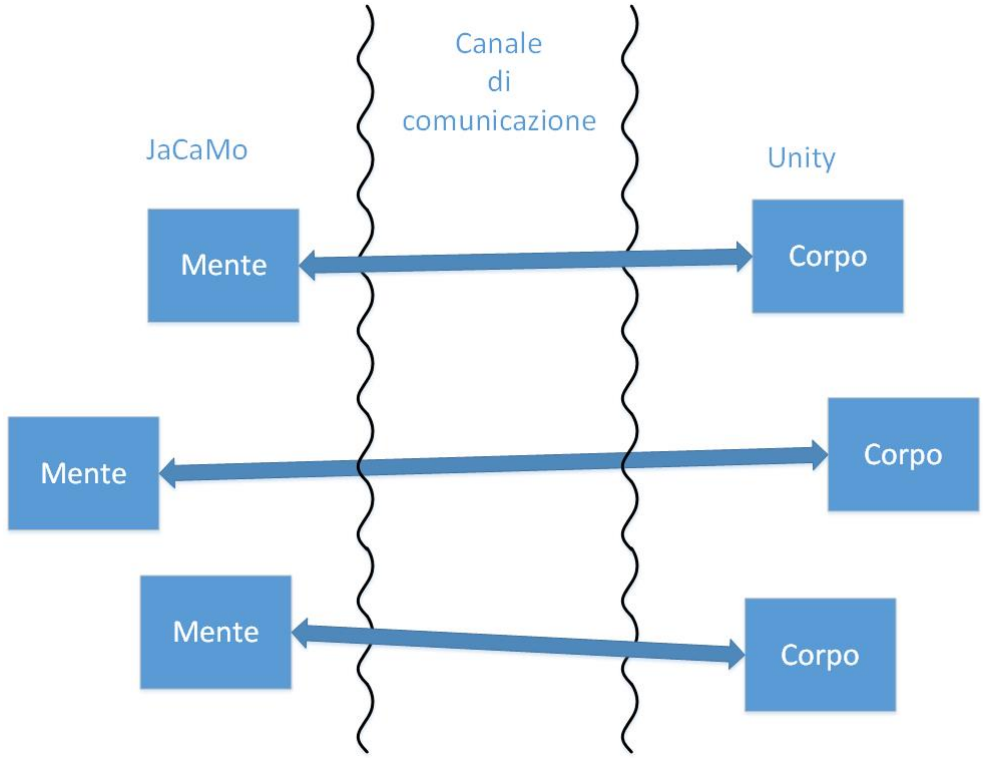
- **1:1** - È il caso in cui tutti gli altri casi possono essere ricondotti, è presente solo 1 MAS e può comunicare con 1 solo ambiente virtuale.

Fatte le precedenti osservazioni, si può concludere che qualunque sia la necessità di uno sviluppatore che utilizzi questo sistema, la progettazione della comunicazione di MAS con ambienti virtuali è sempre riconducibile alla comunicazione di 1 MAS con 1 ambiente virtuale. Però si darà la possibilità di eseguire i due sotto-sistemi sulla stessa macchina o da 2 macchine separate, connesse tra di loro.

3.2.2 Struttura generale

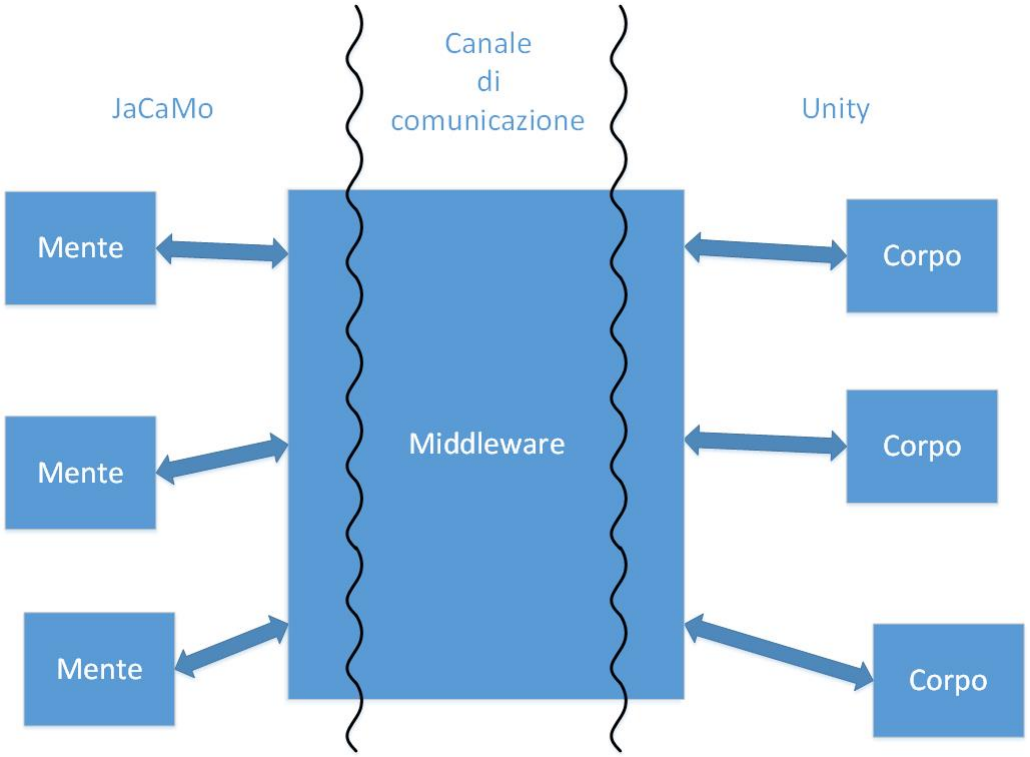
Per progettare l'intero sistema, si è adottato un approccio *top-down*, partendo dal sistema composto da macro elementi, scindendoli fino ad arrivare ai componenti veri e propri da implementare con le due tecnologie utilizzate.

L'idea di base, come già detto, è far comunicare diverse **menti virtuali** con altrettanti **corpi virtuali**.



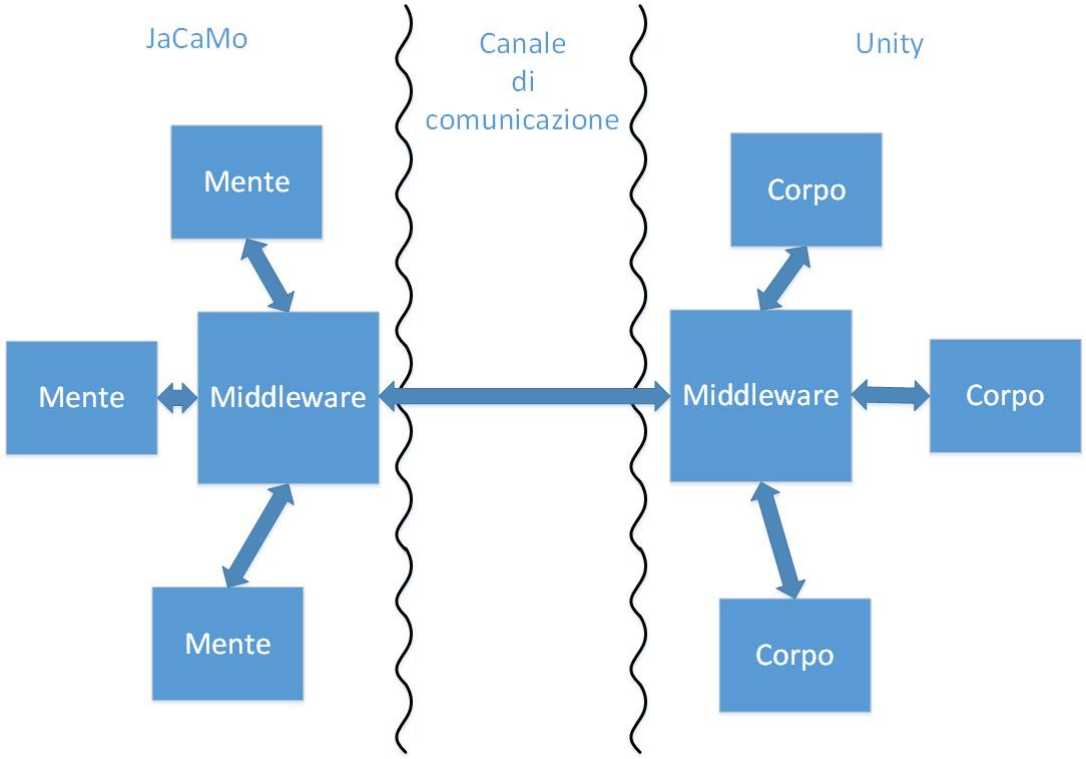
[Fig. 3.2] Idea di partenza

Per fare ciò è necessario progettare e implementare un middleware che permetta la comunicazione dei due "mondi".



[Fig. 3.3] Aggiunta di un middleware tra il lato JaCaMo e Unity

Un middleware, per definizione, è un programma che funge da intermediario tra diverse applicazioni e componenti software. Questi componenti che interagiscono con il middleware, sono spesso e volentieri implementati con tecnologie diverse (come in questo caso), perciò è necessario "spezzare" il middleware in 2 parti.



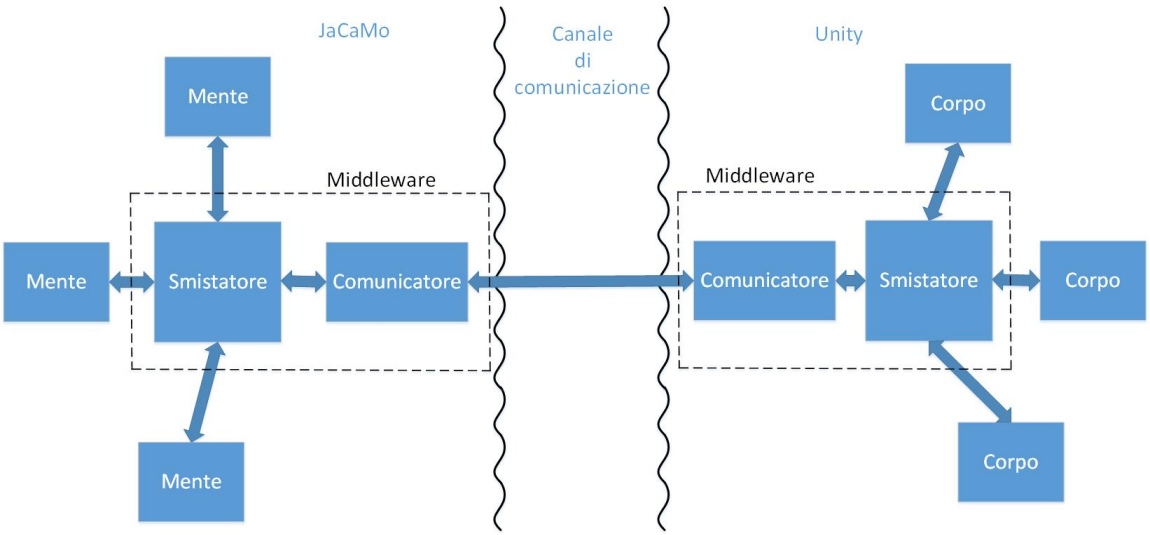
[Fig. 3.4] Il middleware viene diviso in 2 parti, poste sui due lati del canale di comunicazione

Salta subito all'occhio un problema, ora il middleware deve, oltre che gestire la comunicazione, assicurarsi che i comandi spediti da una determinata mente, arrivino solo al corpo relativo a quella mente. Parallelamente, le percezioni inviate da un determinato corpo, devono arrivare solo alla sua mente.

Quindi possiamo vedere i componenti di cui sarà composto ogni lato del middleware:

- **comunicator** (comunicatore) - sarà responsabile dello scambio di messaggi attraverso il canale di comunicazione.

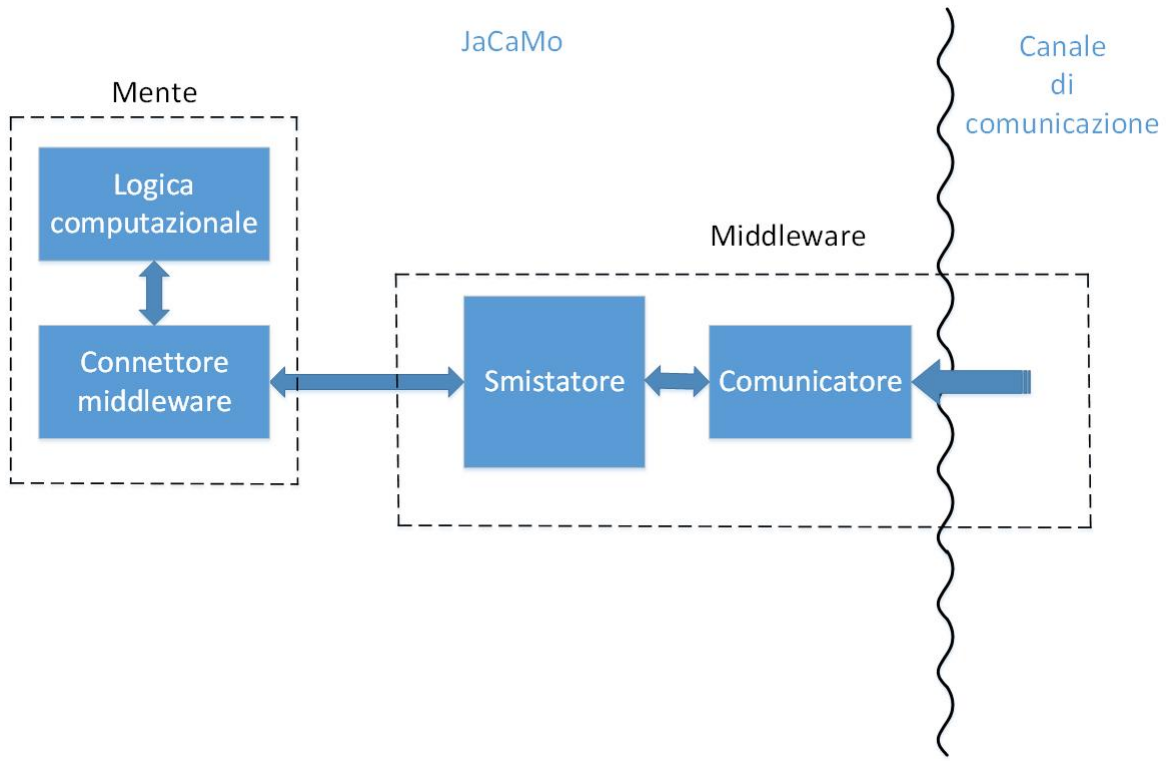
- **sorter** (smistatore) - si preoccuperà di far arrivare i messaggi ai giusti destinatari.



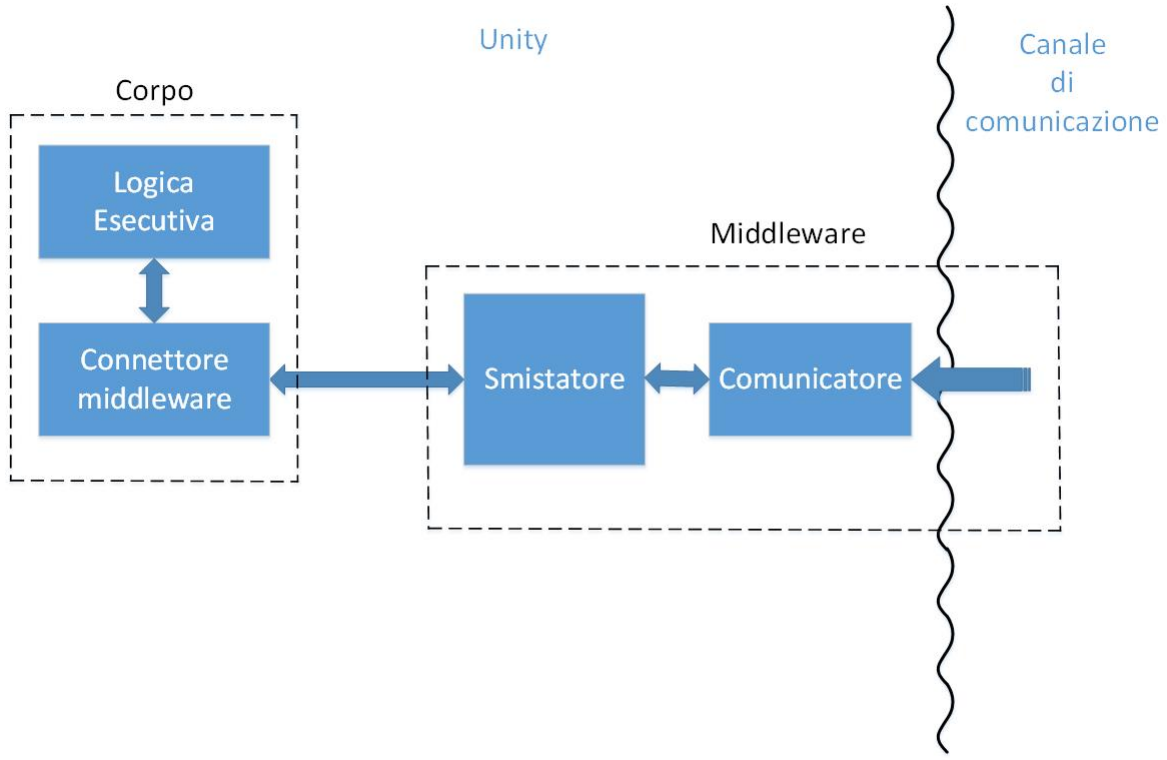
[Fig. 3.5] Si suddividono i middlewares con i loro componenti

Ottenuta questa struttura, bisogna notare una cosa: le menti devono avere un riferimento al relativo smistatore, il quale fa parte del middleware che si preoccuperà della comunicazione con la controparte dell'ambiente virtuale. Tale riferimento però genera una contraddizione con la volontà di creare un middleware che semplifichi il più possibile il lavoro agli sviluppatori di videogiochi e simulazioni, rendendo trasparente la comunicazione tra i due mondi. Discorso analogo si può fare per il lato dei corpi.

Per ovviare a questo problema, si è pensato di inserire un "pezzo" di middleware in ogni mente (e in ogni corpo), in modo tale da rendere trasparente la comunicazione. In questo modo si avranno delle entità (menti e corpi) con al loro interno una parte comunicativa, implementata in questa tesi, e una parte computazionale/esecutiva, lasciata agli sviluppatori che utilizzeranno questo progetto.



[Fig. 3.6] Suddivisione di una mente

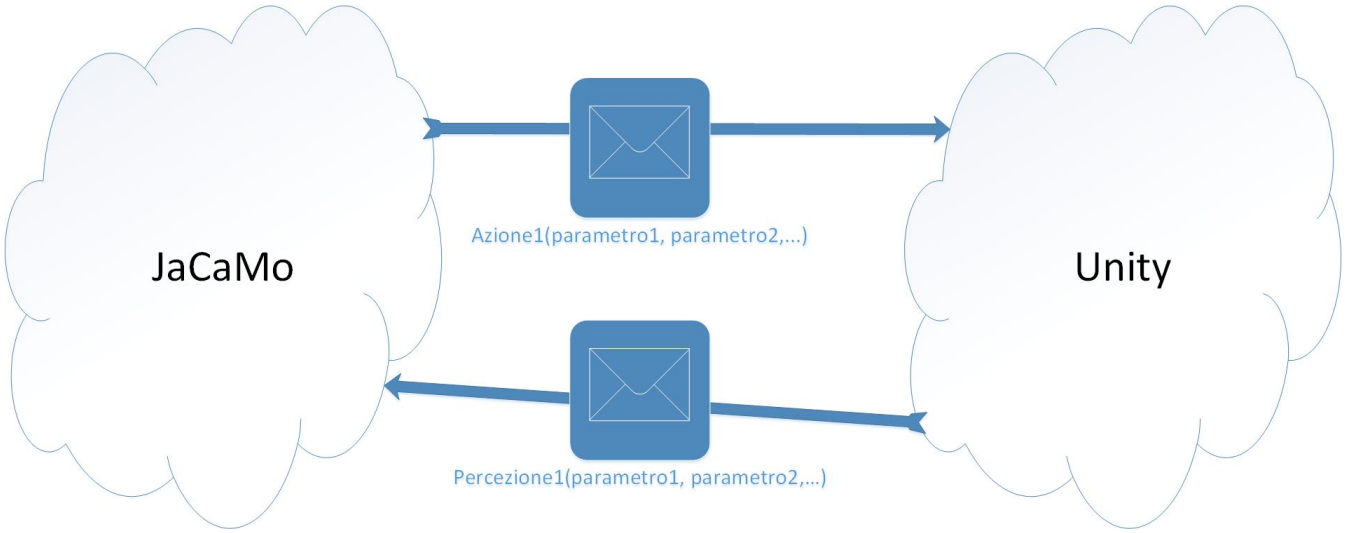


[Fig. 3.7] Suddivisione di un corpo

Finita la progettazione dell'architettura, si inizia a entrare nel dettaglio dei contenuti dei componenti.

Comunicazione dei componenti

Prossimo passo è definire come strutturare i messaggi scambiati tra le entità. Da una parte, le menti devono definire quale azione deve compiere il relativo corpo (es. "muoviti in avanti", "ruota", "spara", ecc.), dall'altro i corpi devono far sapere alle relative menti le proprie percezioni dell'ambiente circostante (es. "mi ha toccato un entità", "sono alle coordinate 23,12,-6", ecc.). Sia nel caso delle azioni da menti verso corpi, sia in quello delle percezioni da corpi verso menti, è utile definire anche dei parametri, per poter riutilizzare le stesse azioni/percezioni in circostanze diverse.

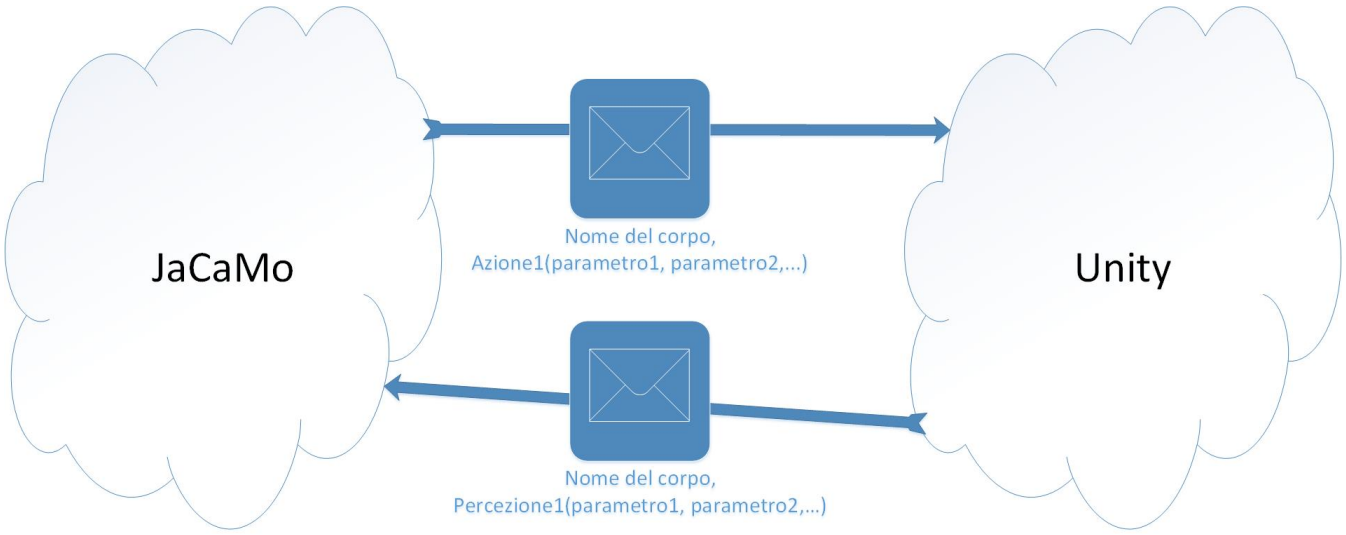


[Fig. 3.8] Composizione dei due tipi di messaggi

Si ricordi che è presente un componente che deve smistare i messaggi in ingresso, a tal proposito bisogna trovare un modo per identificare il destinatario di uno specifico messaggio.

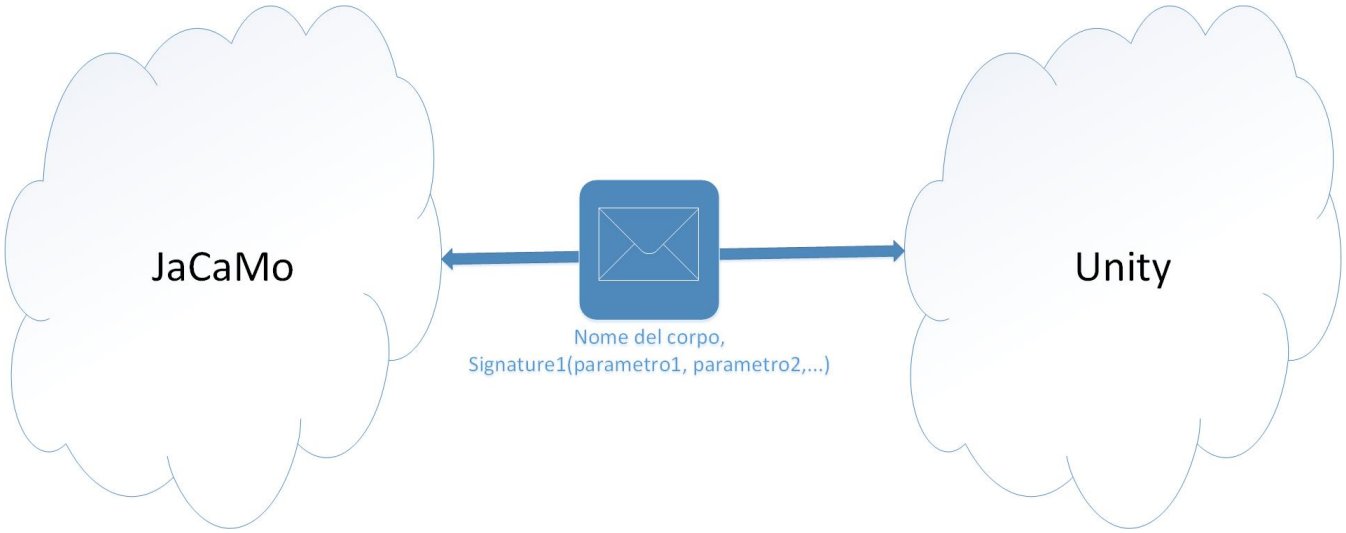
Per risolvere tale problema, si è deciso di assegnare un nome univoco per ogni mente e ogni corpo. Avendo ipotizzato di poter collegare una sola mente a un solo corpo (e viceversa), è possibile mettere nel messaggio solo uno dei due nomi (della mente o del corpo).

Supponendo di scegliere il nome del corpo da includere nel messaggio, la composizione finale dei messaggi di azione e di percezione è la seguente.



[Fig. 3.9] Aggiunta del campo "nome del corpo"

Si noti come la struttura dei messaggi di azione e dei messaggi di percezione sia pressoché identica. Si è deciso allora di unificare le due strutture, fondendo i campi "azione" e "percezione" sotto il nome di "signature" ("firma"), tale campo identificherà una specifica azione nei corpi e una specifica percezione nelle menti.

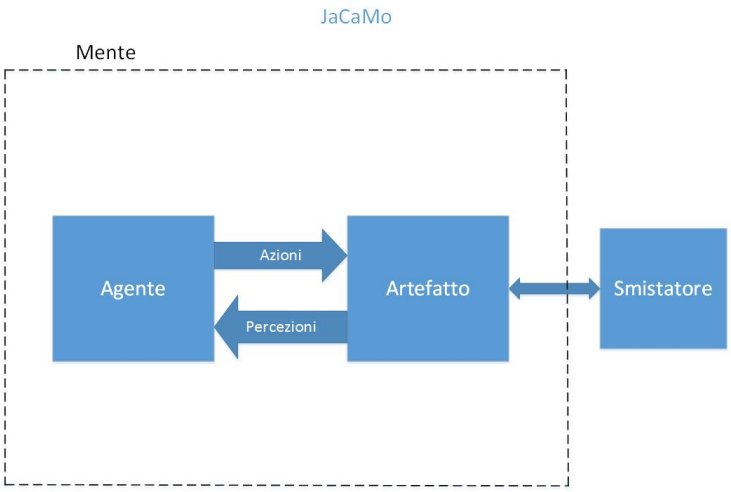


[Fig. 3.10] Unificazione dei messaggi

La mente

Per sviluppare le menti, si può far uso di due strutture già spiegate nel capitolo 2: agenti e artefatti.

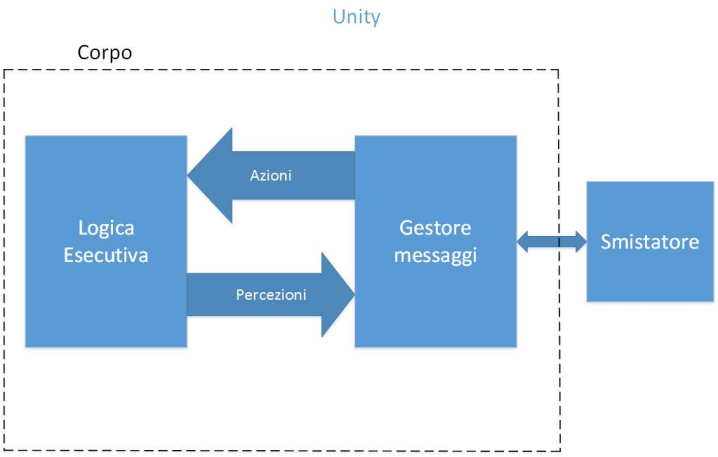
Gli agenti conterranno la logica computazionale della mente (cioè l'intelligenza artificiale), mentre gli artefatti verranno considerati come i corpi di quelle menti (come si vedrà nella sezione implementativa, saranno gli artefatti a essere responsabili dell'invio dei messaggi ai corpi su Unity). Lo sviluppatore "finale" (colui che utilizzerà il middleware), sarà incaricato di creare i messaggi contenenti le azioni e inviarli con una funzione specifica del middleware. Analogalmente, riceverà in maniera trasparente il messaggio contenente le percezioni, ma avrà la responsabilità di leggerlo per poi creare un I.A. che si comporti di conseguenza.



[Fig. 3.11] Utilizzo delle strutture di JaCaMo per le menti

Il corpo

Per sviluppare i corpi, si farà in modo che lo sviluppatore finale riceva il messaggio contenente la signature e i parametri dell'azione da compiere, sarà poi sua la responsabilità di leggere il messaggio e scrivere il codice che permetterà al corpo virtuale di compiere l'azione richiesta. Sempre lo sviluppatore, potrà decidere di ottenere le percezioni come meglio crede, avendo l'onere di creare il messaggio contenente la signature e i parametri della percezione, per poi spedirlo con un metodo specifico che renda l'invio trasparente ai suoi occhi.



[Fig. 3.12] Divisione interna dei corpi

3.3 Implementazione

Comunicazione dei componenti: messaggi TCP

Da una parte è presente un sistema basato sul linguaggio *C#* (Unity), dall'altro un sistema basato sul linguaggio *Java* (JaCaMo). Tenendo presente che bisogna dare la possibilità di creare sia un sistema concentrato (MAS e ambiente sulla stessa macchina), sia distribuito (MAS e ambiente su 2 macchine diverse, collegate tra di loro), si è scelto di utilizzare la comunicazione di rete come canale di comunicazione dei due sotto-sistemi. Nel caso di un sistema concentrato, basterà effettuare una connessione del client all'indirizzo di loopback (127.0.0.1) per collegare i due sotto-sistemi presenti sulla stessa macchina. Nel caso di un sistema distribuito, non si dovrà usare l'indirizzo di loopback, ma l'indirizzo della macchina contenente il server.

Si è scelto di utilizzare il protocollo TCP per il collegamento dei sotto-sistemi, piuttosto che UDP, vista la necessità di avere una connessione stabile e soprattutto affidabile.

Per quanto riguarda quale dei due sotto-sistemi dovesse agire come server e quale come client, vista la natura a doppio senso della comunicazione, si è deciso di creare il server lato JaCaMo, mentre il lato Unity dovrà sapere a priori l'indirizzo e la porta del MAS a cui dovrà riferirsi.

Protocollo

Il protocollo di comunicazione sarà molto semplice, prevederà un solo messaggio per ogni interazione. Bisogna tenere presente che, se un messaggio viene inviato **dal lato JaCa-**

Mo, sicuramente sarà un **azione**; se un messaggio viene inviato **dal lato Unity**, sarà invece una **percezione**.

Questo messaggio sarà così strutturato:

- **Nome del corpo** - indicherà quale corpo dovrà ricevere il messaggio (per le azioni) o quale corpo ha inviato il messaggio (per le percezioni);
- **Signature** - indicherà l'identificativo dell'azione o della percezione inviata;
- **Numero di parametri** - indicherà quanti parametri sono stati inviati insieme al messaggio, può essere anche 0 se l'azione/percezione non necessita di parametri;
- **Parametro** - indica il valore di 1 singolo parametro, questo campo è ripetuto tante volte quanti parametri sono stati inviati col messaggio (non è presente se "Numero di parametri" è pari a 0).

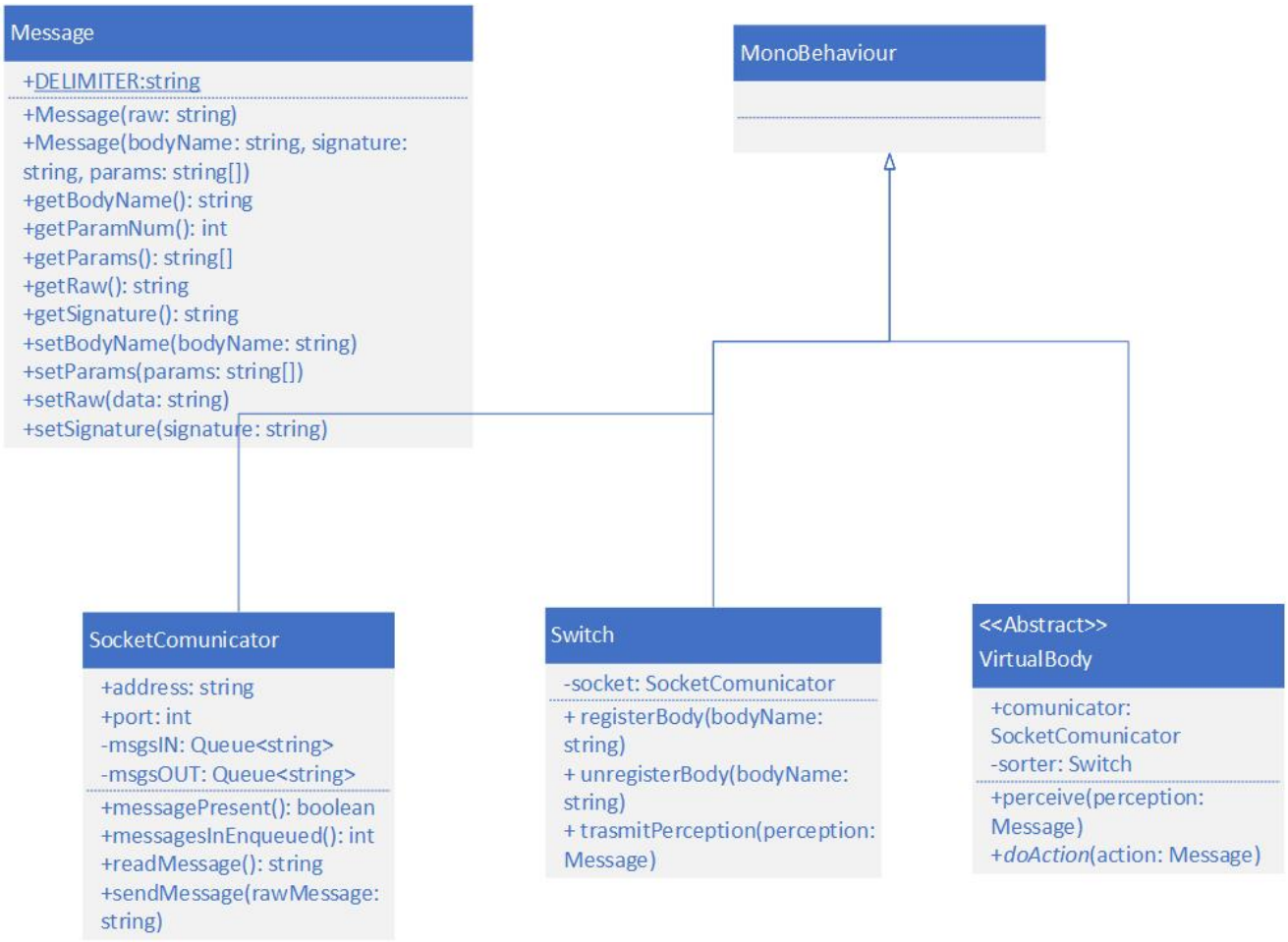
Ogni messaggio che percorrerà la rete, sarà rappresentato da una stringa con il seguente formato:

```
NomeDelCorpo|Signature|NumeroDeiParametri|Parametro1|Parametro2|Parametro3|...|ParametroN
```

Seguono alcuni esempi a scopo esplicativo:

- Il corpo "sfera1" deve compiere l'azione "spostatiIn(23,-5,0)"
  `sfera1|spostatiIn|3|23|-5|0`
- Il corpo "armaDestra" deve compiere l'azione "spara()"
  `armaDestra|spara|0`
- Il corpo "giocatore" deve inviare la percezione "toccatoDa(proiettile)"
  `giocatore|toccatoDa|1|proiettile`

Il corpo: Unity3D



[Fig. 3.13]

Per implementare l'intero ambiente virtuale, sono necessarie 4 classi (vedi schema):

- *SocketComunicator* - rappresenta il comunicatore del middleware, è responsabile della comunicazione con la controparte in JaCaMo. All'avvio dell'ambiente virtuale, si occupa di instaurare la connessione con il lato JaCaMo, dopodiché fa essenzialmente 2 cose:
  - **ascolta la socket** per messaggi in ingresso e li mette in una coda (*msgsIN*), pronti per essere distribuiti dallo smistatore;
  - **scrive i messaggi** presenti in una coda (*msgsOUT*) nella socket, togliendoli da tale coda.
- *Switch* - rappresenta lo smistatore del middleware, si occupa della distribuzione dei messaggi ai giusti destinatari. Quando i corpi dovrebbero compiere le azioni, quindi

durante la fase *Update* del loop di Unity (vedi capitolo 1), svuota la coda *msgsOut* del *socketComunicator*, distribuendo le azioni ai corpi indicati nei messaggi.

- *VirtualBody* - rappresenta il componente comunicativo presente nelle classi che svilupperanno gli utilizzatori del middleware. Gli oggetti che devono venir considerati come corpi virtuali (cioè che debbano avere un accoppiamento con una mente lato JaCaMo), devono estendere questa classe. La proprietà pubblica *comunicator* dovrà riferirsi al comunicatore istanziato.

  Il metodo *perceive* sostanzialmente permette ai messaggi passati come parametro, di arrivare alla mente collegata al corpo.

  Il metodo *doAction* è un metodo astratto, quindi dovrà essere implementato nelle classi derivate. Questo metodo è chiamato dallo smistatore quando riceve un messaggio rivolto a quella particolare istanza. Lo sviluppatore che implementerà questo metodo, potrà leggere dal parametro *action* la signature del messaggio ricevuto, implementando algoritmi specifici per ogni azione che il corpo potrà compiere.

- *Message* - è la classe che rappresenta i messaggi ricevuti e inviati dal middleware, implementa getter e setter per tutti i campi discussi nella sezione **Protocollo**. Inoltre è responsabile di creare e decodificare la stringa che deve passare attraverso la rete.

La mente: JaCaMo

Premessa: ancora non esiste uno standard per la rappresentazione di schemi per modelli ad agenti. Quindi nel seguente schema ho utilizzato la notazione UML per modelli ad oggetti, con le seguenti modifiche:

- Gli agenti sono caratterizzati dallo stereotipo `<<Agent>>` (è stato cambiato anche il colore per facilità di lettura);

- Per gli agenti, la sezione delle proprietà è dedicata alle conoscenze iniziali e agli obbiettivi iniziali(se preceduti da *!*);

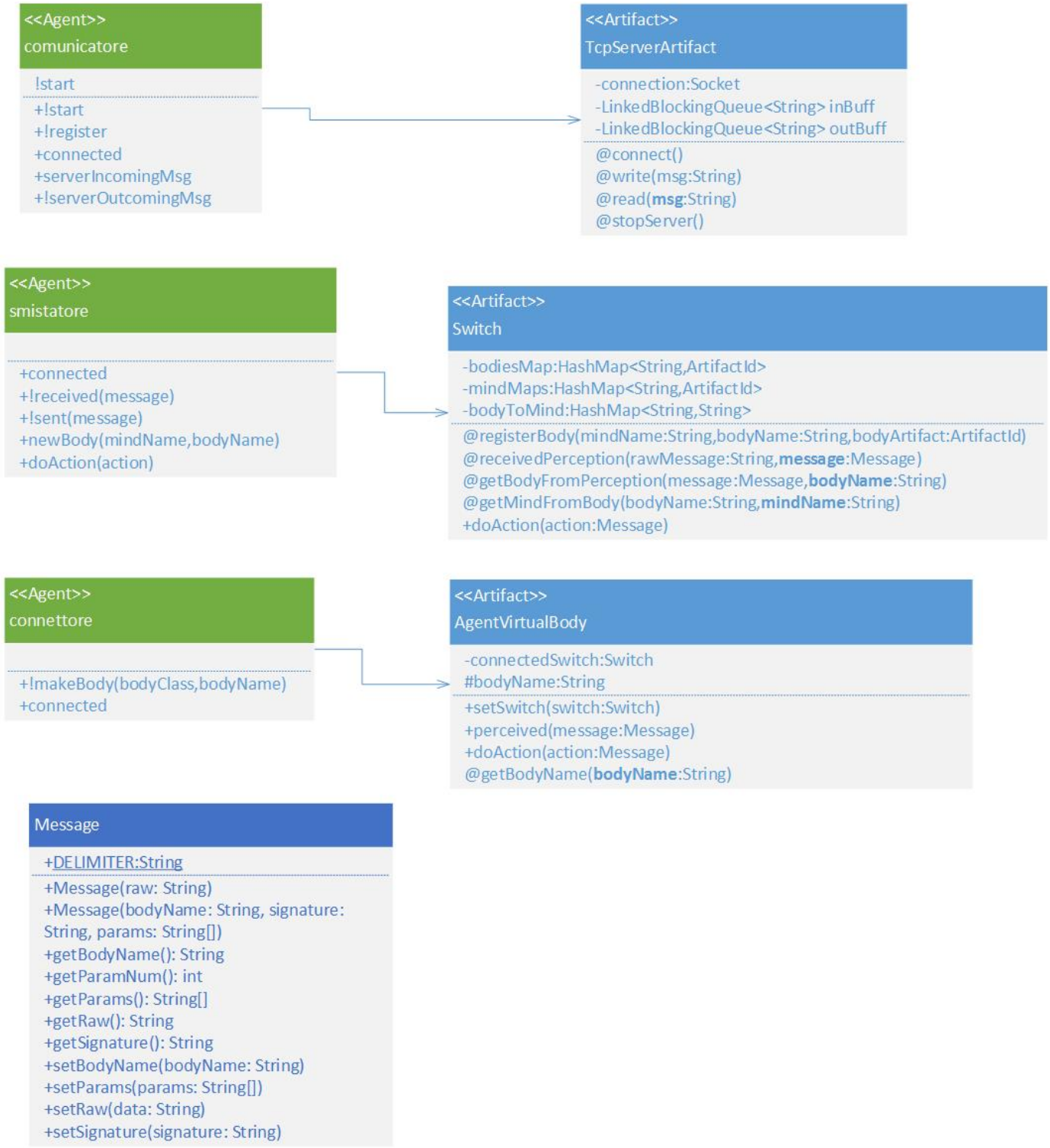
- Per gli agenti, la sezione dei metodi è dedicata ai *trigger* dei loro *piani* (si rimanda al capitolo 2 per delucidazioni);

- Gli artefatti sono caratterizzati dallo stereotipo `<<Artifact>>` (è stato cambiato anche il colore per facilità di lettura), inoltre ogni artefatto utilizzato, estende la classe `Artifact` di Cartago, questa estensione è stata omessa nello schema;

- Le frecce che collegano artefatti e agenti indicato che un determinato agente ha accesso alle operazioni di un dato artefatto;

- Il simbolo @ che precede un metodo negli artefatti, indica un'*operazione*; inoltre i parametri in uscita sono indicati in grassetto ed è stato omesso il wrapper *OpFeedbackParam*

- Gli oggetti Java non hanno stereotipo (è stato cambiato anche il colore per facilità di lettura).



[Fig. 3.14]

Come spiegato nel capitolo 2, ogni agente necessita degli artefatti per estendere le proprie funzionalità e interagire con il mondo "esterno".

Quindi i componenti del middleware lato JaCaMo spiegati nella sezione 3.2.2, saranno divisi in 1 agente e 1 artefatto. Tenere sempre presente che per il modello ad agenti, gli

agenti sono entità autonome, mentre gli artefatti sono reattive, utilizzati dagli agenti.

- Il comunicatore sarà composto dall'agente `comunicatore` e dall'artefatto `TcpServerArtifact`. L'agente *comunicatore*, oltre che a chiamare i comandi del proprio artefatto per inizializzare il *Server TCP*, farà da mediatore tra i messaggi della rete e gli altri agenti del MAS, in particolare con lo *smistatore*. L'artefatto *TcpServerArtifact*, si occuperà fisicamente dell'inizializzazione del server. Inoltre sarà responsabile della scrittura in rete dei messaggi e della notifica al *comunicatore* dell'avvenuta ricezione di un nuovo messaggio. La notifica dei nuovi messaggi in ingresso, in realtà invalida la regola per la quale un artefatto è solo reattivo, dato che praticamente questo artefatto conterrà un *Thread* autonomo che controlli la porta di ingresso dalla rete. Questa eccezione è necessaria per evitare di tenere l'agente *comunicatore* in un loop continuo di controllo dei messaggi in ingresso dalla rete.

- Lo smistatore sarà composto dall'agente `smistatore` e dall'artefatto `Switch`. Questi due componenti collaboreranno per la consegna delle percezioni alle menti giuste. Oltre a smistare appunto, le menti che verranno istanziate, dovranno registrarsi allo *Switch* fornendo il nome dell'agente (*mindName*), l'id dell'artefatto (*bodyArtifact*) lato JaCaMo e il nome del corpo lato Unity (*bodyName*) per poter permettere lo smistamento.

- Il connettore sarà composto dall'agente `connettore` e dall'artefatto `AgentVirtualBody`. Il *connettore* è responsabile di instaurare le connessioni necessarie con il resto del middleware in modo trasparente allo sviluppatore della I.A. e della creazione dell'istanza dell'artefatto esteso da *AgentVirtualBody*. L'artefatto che "simulerà" il corpo lato JaCaMo dovrà estendere `AgentVirtualBody`. I metodi che interessano gli sviluppatori finali, sono `doAction` e `perceived`.

  - `doAction(action:Message)` - deve essere chiamata da dentro la classe derivata, è il metodo che permette alle azioni di arrivare dal lato Unity;
  - `perceive(perception:Message)` - di base, è un metodo che chiama il trigger `+perception(Percezione)` sul proprio agente, se presente, dove `Percezione` contiene il messaggio arrivato dal lato Unity. è possibile sovrascrivere questa funzione per gestire le percezioni in maniera diversa (magari gestendole direttamente dal codice dentro l'artefatto, senza passare dall'agente).

- La classe `Message` rappresenterà, come per la parte Unity, i messaggi ricevuti e inviati dal middleware. Sarà responsabile di creare e decodificare la stringa che deve passare attraverso la rete.

Il comunicatore e lo smistatore entreranno in contatto sfruttando una funzionalità di JaCaMo chiamata *blackboard* (lavagna), in pratica è uno spazio tuple condiviso, dove una volta inizializzati, si "segneranno" con l'istruzione `.out` definendo il proprio nome. Leggendo poi con l'istruzione `.in` verranno a conoscenza del nome della loro controparte per poi potersi "parlare" direttamente con l'istruzione `.send` di JaCaMo.

Le varie menti istanziate entreranno a conoscenza del nome dello smistatore con la medesima funzione.

3.4 Esempio di implementazione

Segue il codice di un piccolo esempio che è stato utilizzato per i test. Tenere presente che questo è il codice che dovrebbe scrivere lo sviluppatore finale che utilizzi il middleware.

Su Unity sono presenti due cubi e una sfera, il cubo di destra `CubeRight` è comandato da un agente in JaCaMo, mentre il cubo di sinistra e la sfera sono programmati nel modo classico con uno script C#. Successivamente verrà mostrato come cambia il codice dal metodo "classico" a quello mediante middleware per la sfera. L'agente comanda al cubo di ruotare su se stesso, e quando percepisce una collisione, gli fa cambiare colore.

L'altro cubo è utilizzato come riferimento per le prestazioni mentre la sfera si muove attraverso i due cubi in continuazione per generare le collisioni periodicamente.

Lato JaCaMo

Artefatto: `TestBody`

```
public class TestBody extends AgentVirtualBody {


        @OPERATION
        void ruota(){
                Message msg=new Message(
                        bodyName,"rotate",new String[]{"0","5","0"});
                doAction(msg);
```

```
        }

        @OPERATION
        void transla(){
                Message msg=new Message(
                        bodyName,"translate",new String[]{"1","0","0"});
                doAction(msg);
        }

        @OPERATION
        void cambiaColore(){
                Message msg=new Message(
                        bodyName,"changeColor",new String[]{});
                doAction(msg);
        }
}
```

Agente: testAgent

```
!start.

{ include("connettore.asl")}

+!start <- !makeBody("TestBody","CubeRight").
+!corpoPronto <- !lavora.
+!lavora <- ruota;!!lavora.
+perception(Msg) <- -perception(Msg); cambiaColore.
```

Lato Unity

In fase di avvio, viene istanziato un *GameObject* chiamato `CubeRight` al quale viene attaccato come componente lo script che segue. Inoltre è presente un *GameObject* "invisibile" contenente gli script `SocketComunicator` e `Switch`

Notare l'estensione da `VirtualBody` e l'utilizzo del metodo `perceive` dentro al `OnTriggerEnter`.

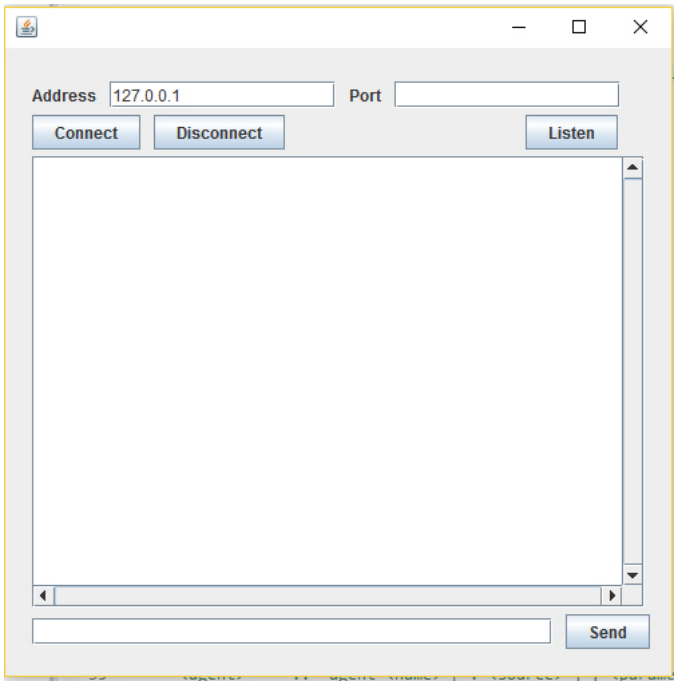
cubeBodyScript

```
public class cubeBodyScript : VirtualBody {

        void OnTriggerEnter(Collider c)
        {
        perceive(new Message(this.name,"collision",new string[] { }));
        }

        public override void doAction(Message msg)
        {
                if ("rotate".Equals(msg.getSignature()))
                {
                        this.transform.Rotate(
                                new Vector3(System.Int32.Parse(msg.getParams()[0]),
                                System.Int32.Parse(msg.getParams()[1]),
                                System.Int32.Parse(msg.getParams()[2])));
                }

                if ("translate".Equals(msg.getSignature()))
                {
                        this.transform.Translate(
                        new Vector3(System.Int32.Parse(msg.getParams()[0]),
                        System.Int32.Parse(msg.getParams()[1]),
                        System.Int32.Parse(msg.getParams()[2])));
                }

                if ("changeColor".Equals(msg.getSignature()))
                {
                        System.Random rand = new System.Random();
                        float r = 0, g=0, b=0;
```

```
                    r=(float)rand.NextDouble();
                    g = (float)rand.NextDouble();
                    b = (float)rand.NextDouble();

                gameObject.GetComponent<Renderer>().material.color =
                new Color(r, g, b);
                }
        }
}
```

3.5 Test

Per effettuare i primi test, si è sviluppato un programma, che chiameremo *comunicatore* capace di collegarsi a server un TCP o aprire un server TCP, che mostri i pacchetti ricevuti in una console e sia in grado di inviare pacchetti creati "on-the-fly". Questo software ha permesso di testare i sistemi singolarmente, senza dover avviare la controparte.



[Fig. 3.15]

- *Address* - specifica a quale indirizzo collegarsi (solo nel caso di connessione come client), i test sono stati fatti tutti in locale, perciò sarà sempre `127.0.0.1`;

- *Port* - specifica a quale porta di comunicazione verrà stabilita la connessione, o verrà aperto il server;

- *Connect* - premendolo, si apre una connessione Client verso un server specificato da address e port;

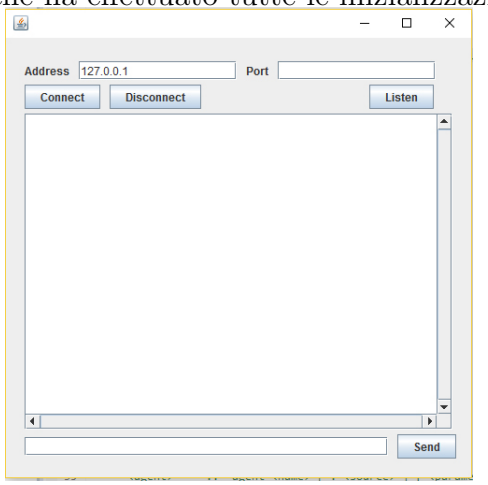
- *Listen* - apre una connessione Server sulla porta specificata da port; address viene ignorato;

- sono presenti anche due caselle di testo: quella più grande permetterà di leggere i messaggi in entrata, mentre quella più piccola in basso servirà a scrivere dei messaggi (inviandoli poi col pulsante *Send*).

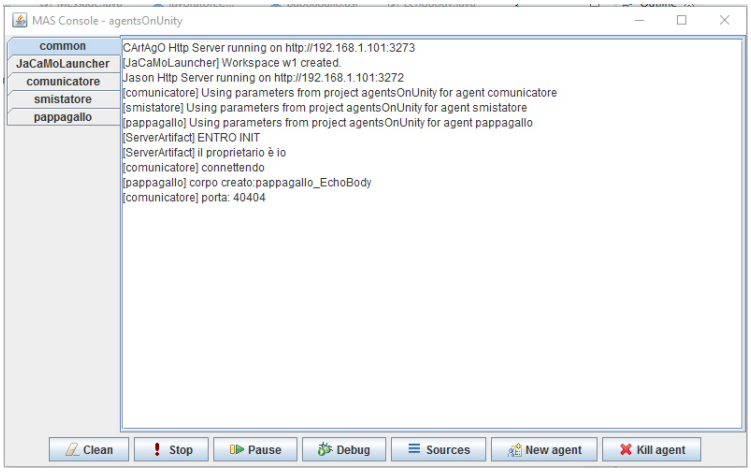
3.5.1 Test lato JaCaMo

È stato creato un agente chiamato "pappagallo" che rispedirà indietro i messaggi ricevuti

1. A sinistra troviamo il comunicatore, ancora non collegato, a destra il sistema MAS che ha effettuato tutte le inizializzazioni.

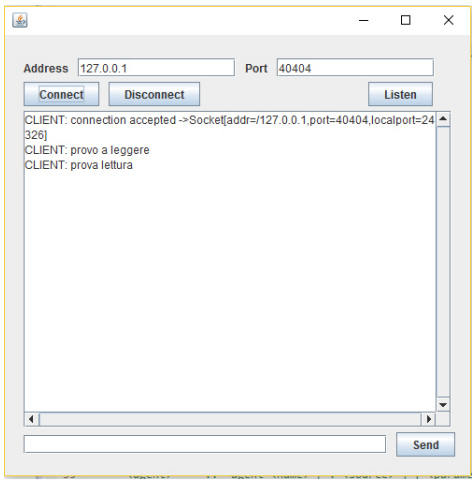


[Fig. 3.16]

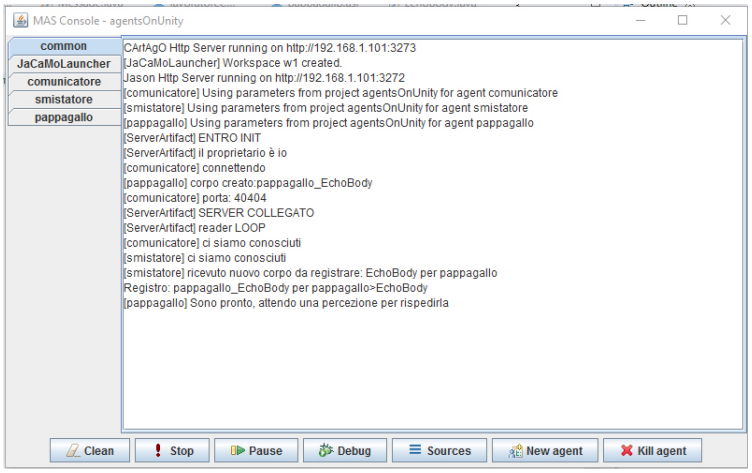


[Fig. 3.17]

2. I due programmi si sono collegati tra loro (ricordando che il MAS è il Server e il comunicatore funge da Client)

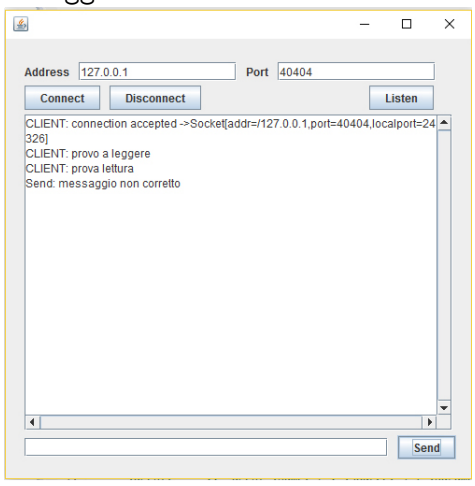


[Fig. 3.18]

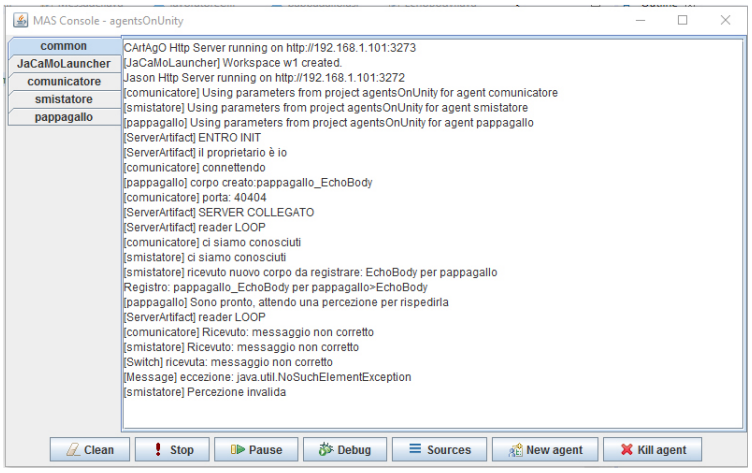


[Fig. 3.19]

3. È stato inviato un messaggio che il MAS non ha riconosciuto come percezione: `messaggio non corretto`

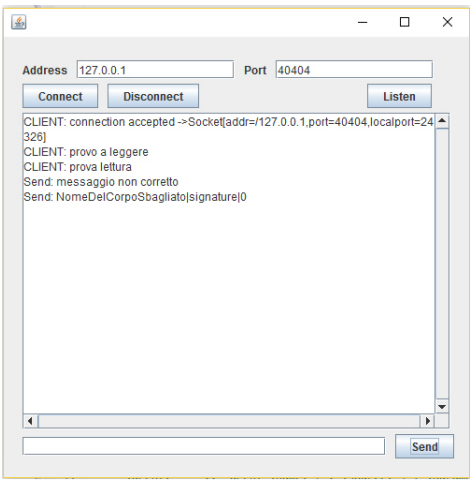


[Fig. 3.20]

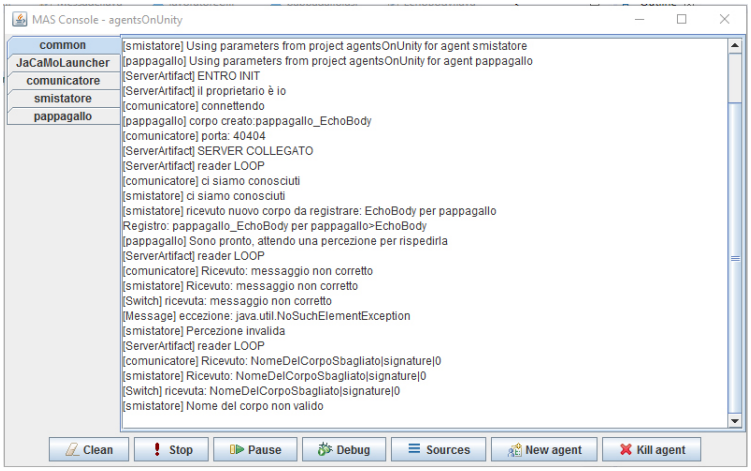


[Fig. 3.21]

4. È stato inviato un messaggio con il giusto formato, ma con un `nomeDelCorpo` sbagliato: `NomeDelCorpoSbagliato|signature|0`

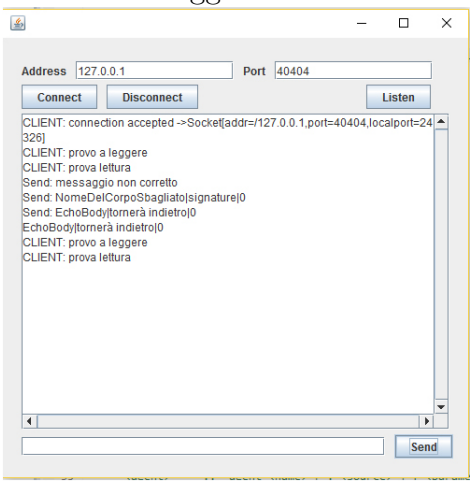


[Fig. 3.22]

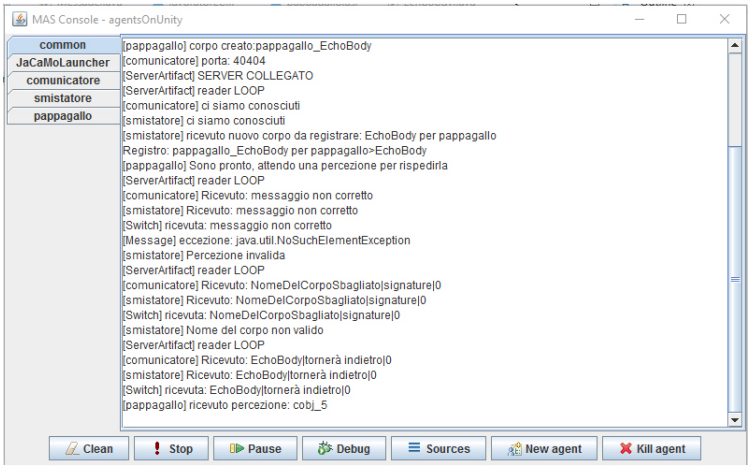


[Fig. 3.23]

5. È stato inviato un messaggio valido all'artefatto dell'agente "pappagallo" che si chiama "EchoBody". Notare che subito dopo la riga di invio del messaggio, è presente la medesima riga senza `Send:` prima; sta a significare che è stato rispedito indietro lo stesso messaggio



[Fig. 3.24]

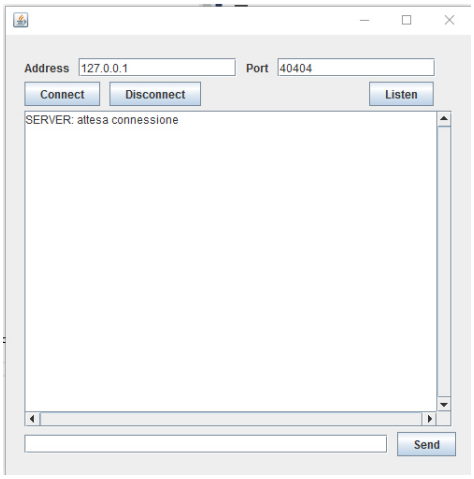


[Fig. 3.25]

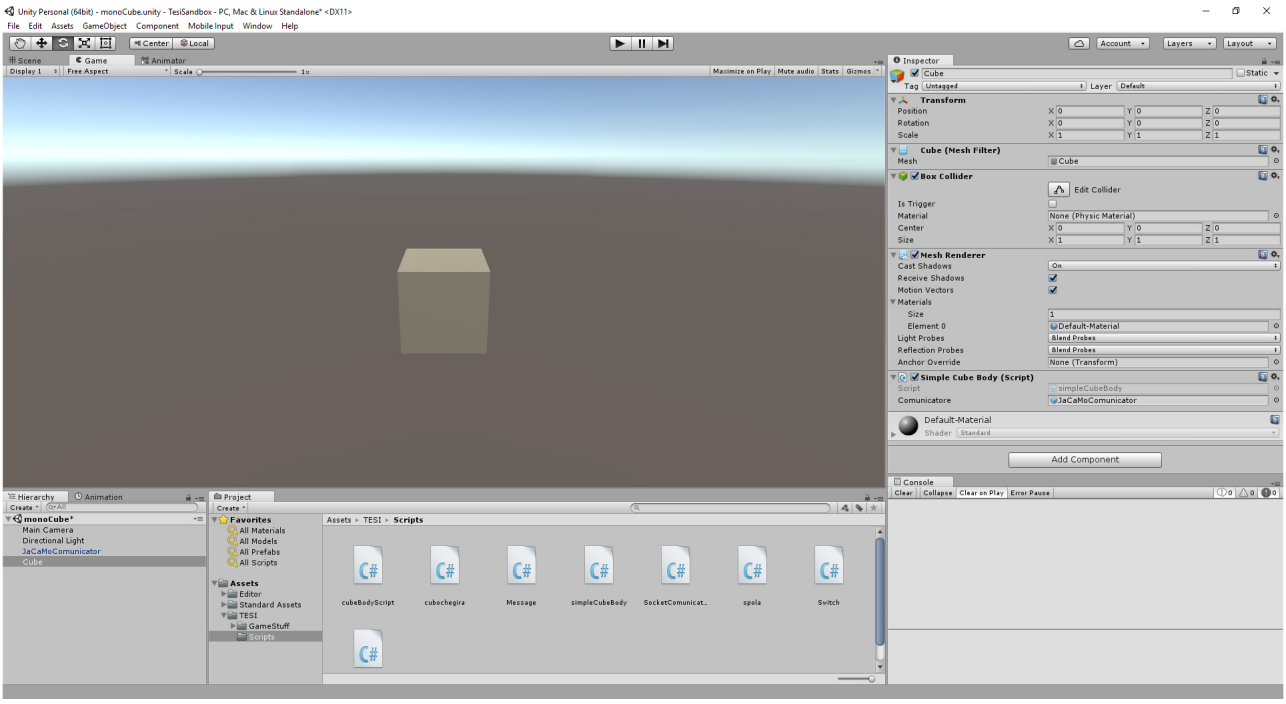
3.5.2 Test lato Unity

È stato creato un cubo che avrà a disposizione le azioni `ruota`(con 3 parametri) e `cambiacolore` (senza parametri). Il primo farà ruotare il corpo su se stesso in base ai parametri inviati, il secondo modificherà casualmente il colore del cubo.

1. Il comunicatore viene avviato in modalità Server e rimane in ascolto sulla porta 40404 (l'importante è che il *socketComunicator* di Unity si colleghi alla stessa porta in ascolto del Server)

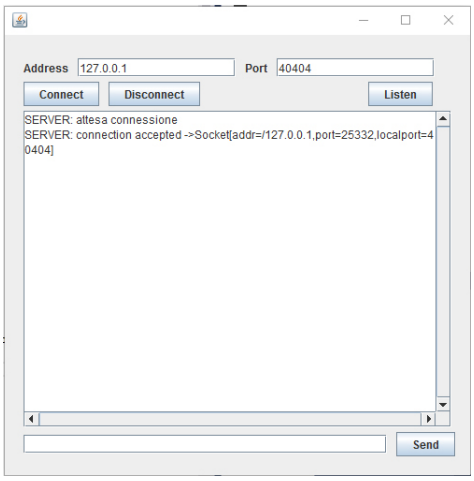


[Fig. 3.26]

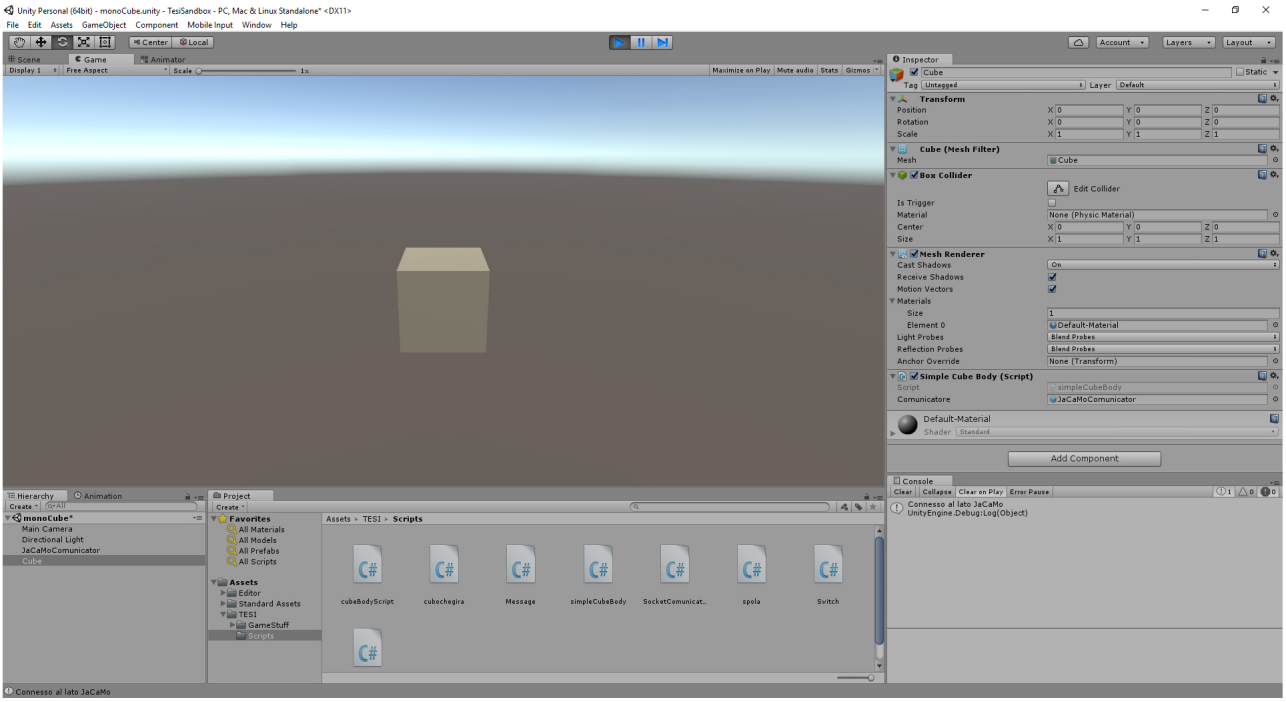


[Fig. 3.27]

2. Viene avviato Unity e viene effettuata la connessione (come si nota nella console in boasso a destra dentro l'interfaccia di Unity)

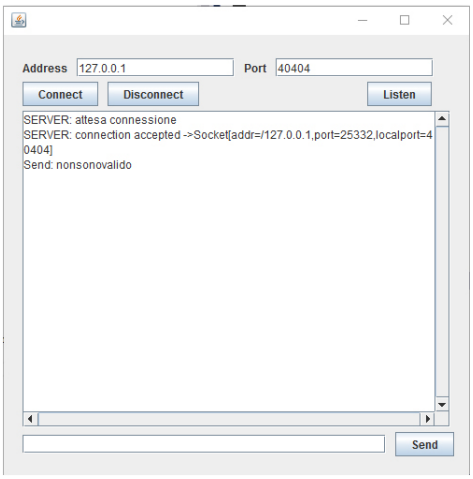


[Fig. 3.28]

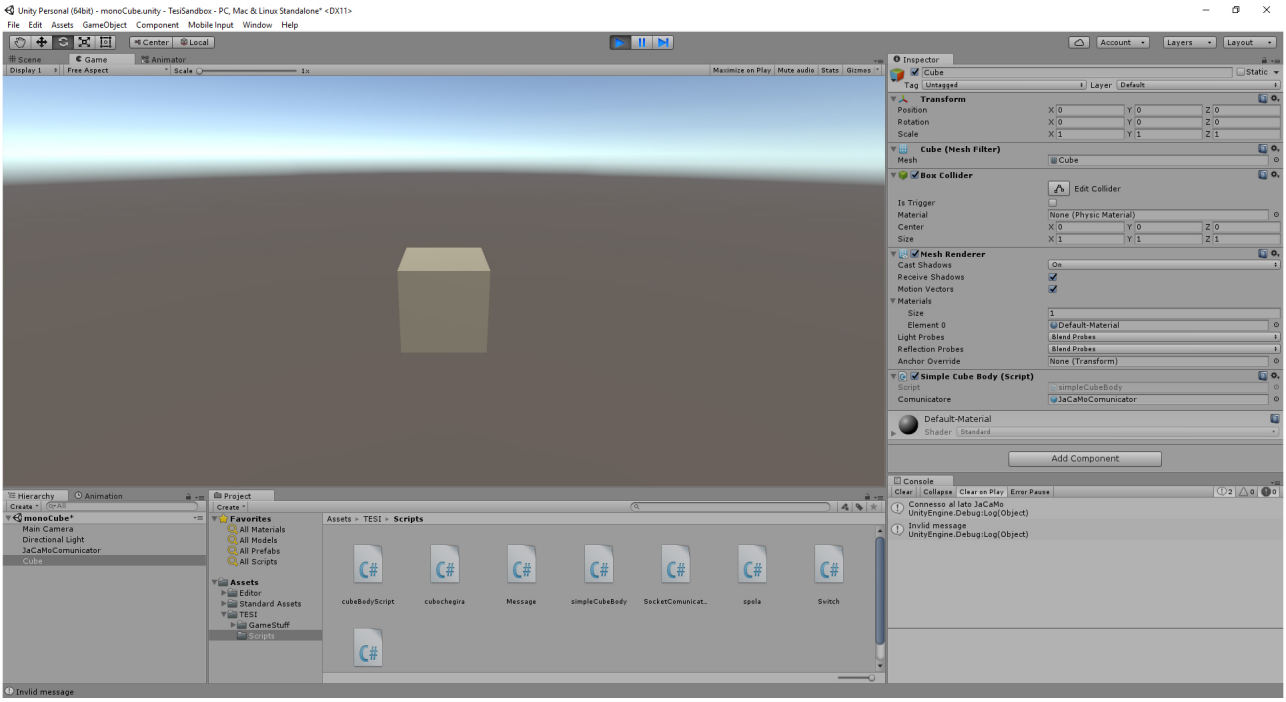


[Fig. 3.29]

3. Viene inviato un messaggio non valido: `nonsonovalido`, Unity lo riconosce e lo scarta (notare la Console di Unity)

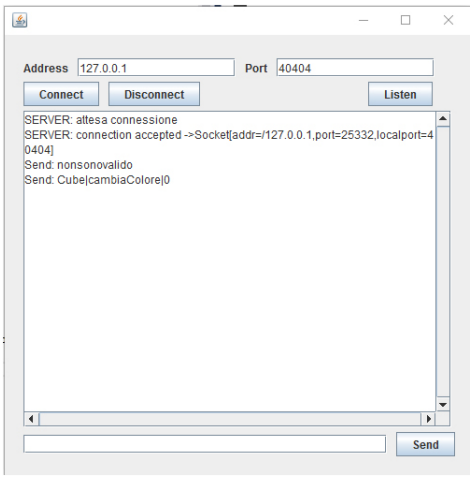


[Fig. 3.30]

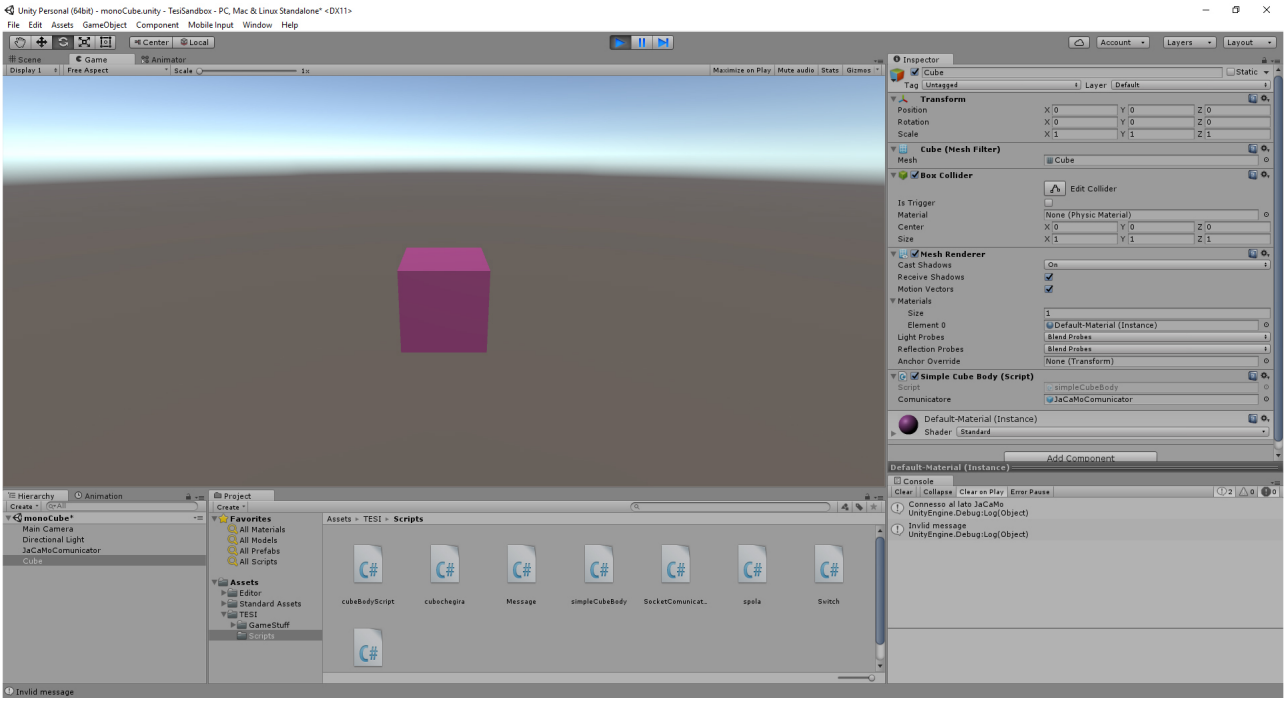


[Fig. 3.31]

4. Viene inviato un messaggio contenente l'azione `cambiaColore` e nessun parametro, il cubo cambia colore come previsto

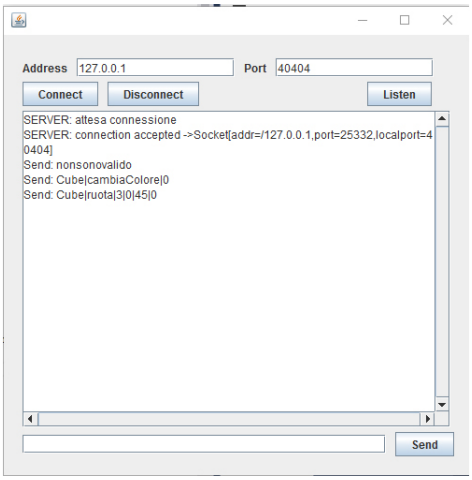


[Fig. 3.32]

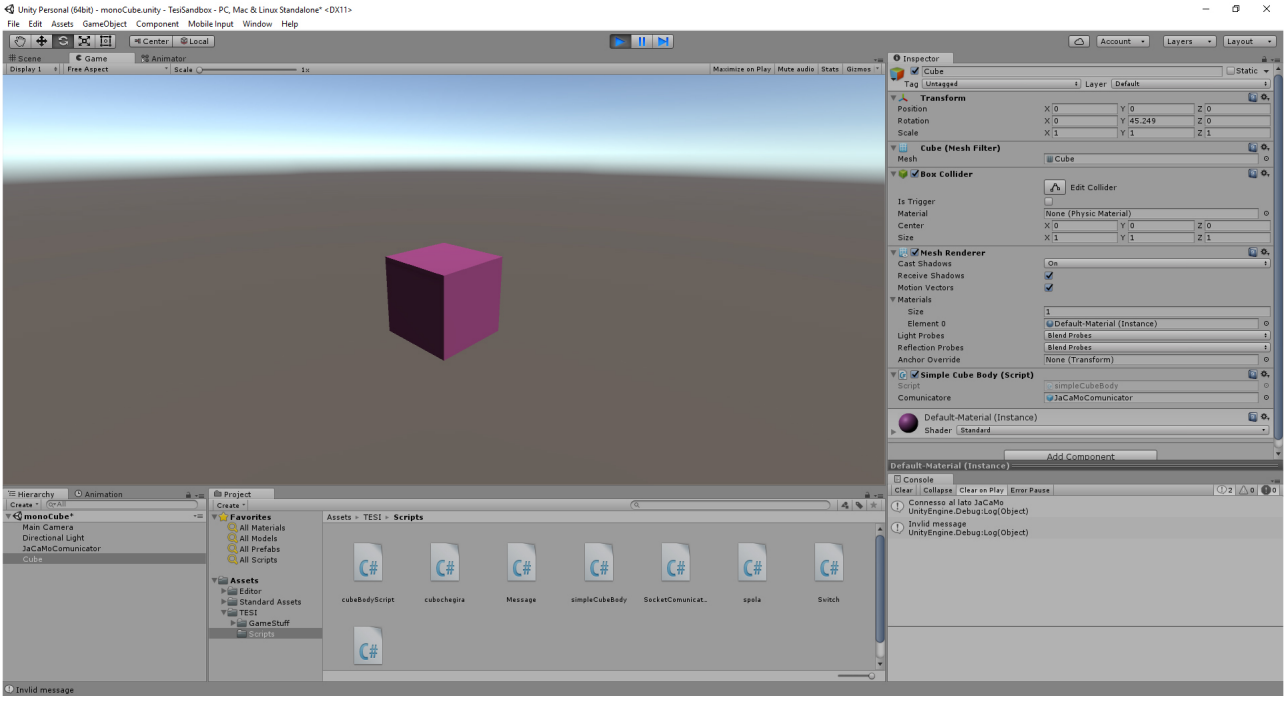


[Fig. 3.33]

5. Viene inviato un messaggio contente l'azione `ruota` con i parametri `0,50,0`, facendo ruotare di 50 unità il cubo secondo l'asse Y (i parametri si riferiscono agli assi X, Y, Z in questo ordine)

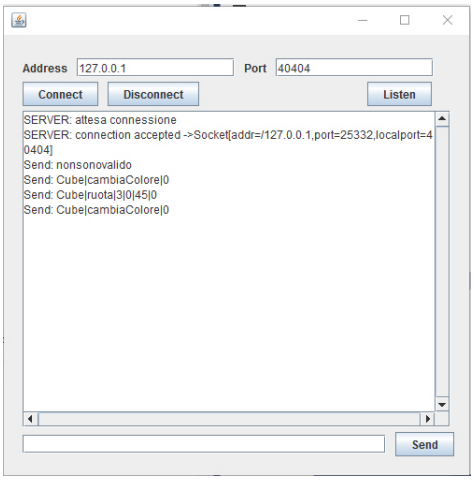


[Fig. 3.34]

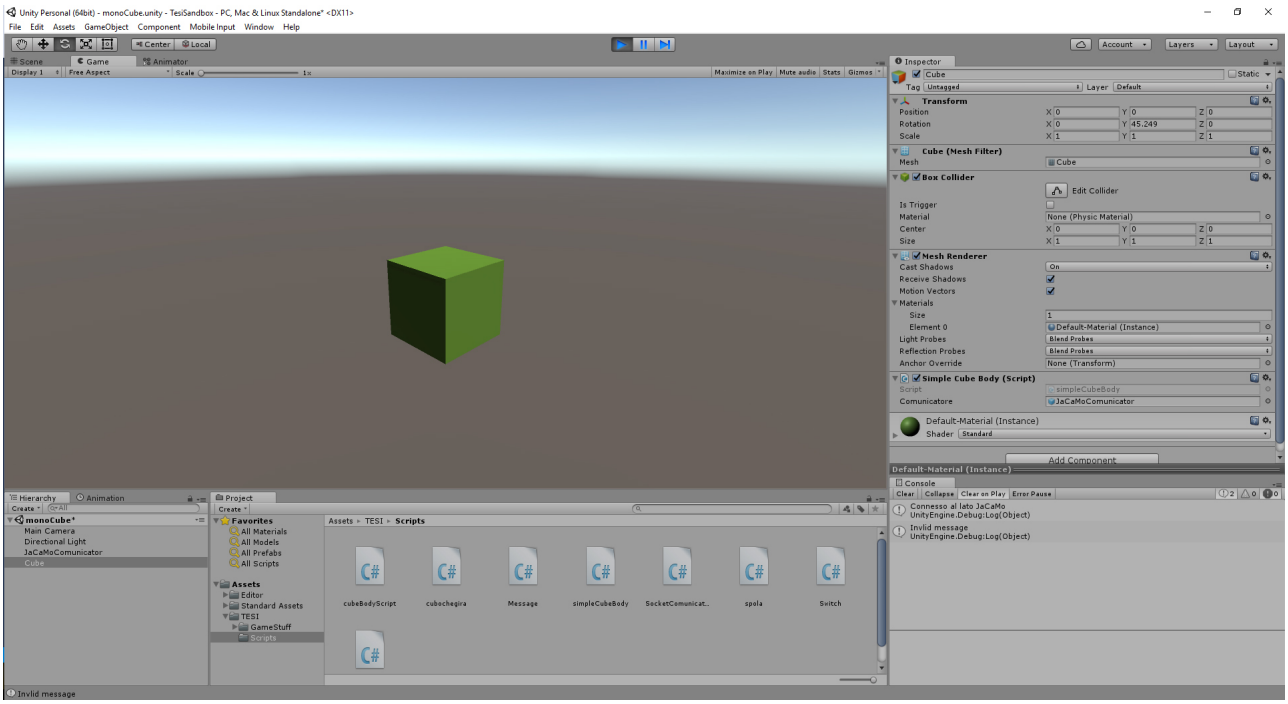


[Fig. 3.35]

6. Viene mandato un messaggio identico al primo valido, cambiando ulteriormente il colore del cubo



[Fig. 3.36]

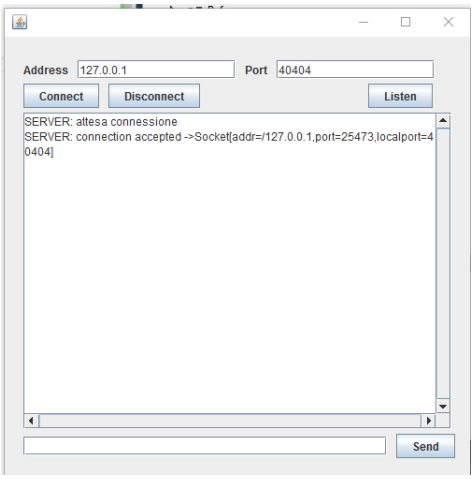


[Fig. 3.37]

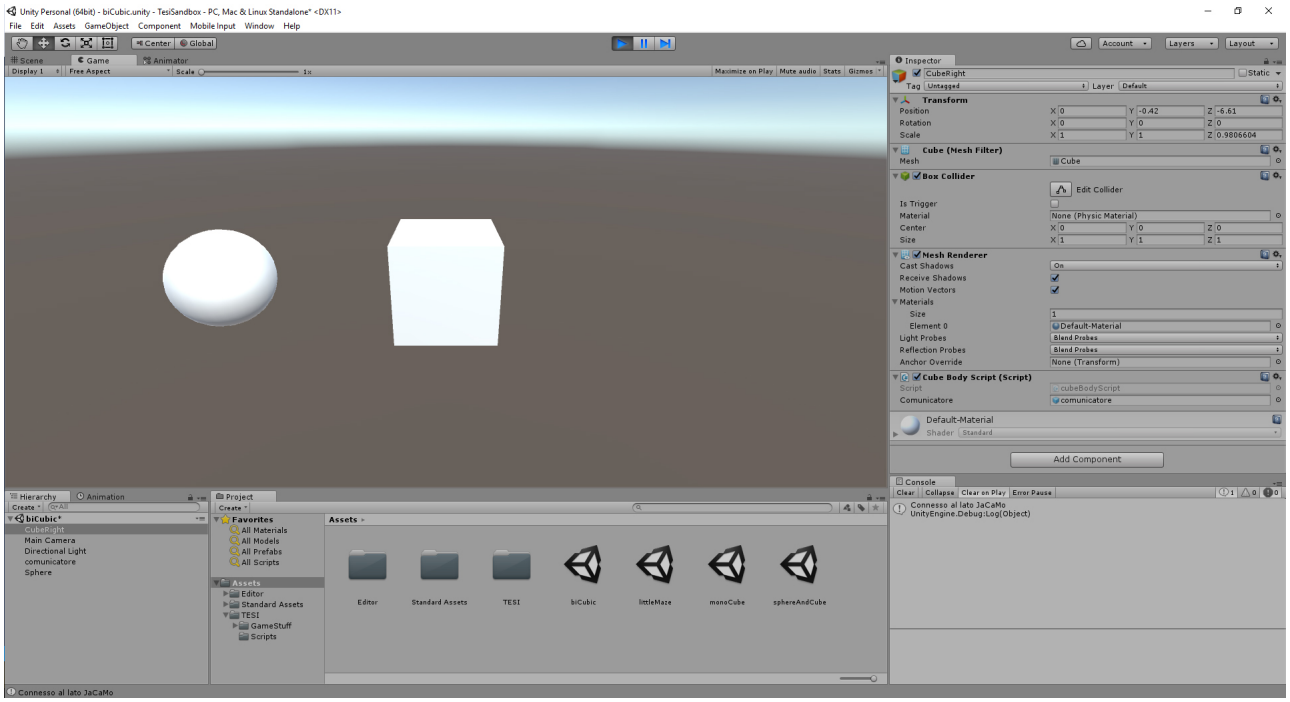
3.5.3 Test di percezione di una collisione

Oltre a un cubo, è presente una sfera che si muove. nel momento della collisione, il corpo del cubo (lato Unity) invia una percezione di collisione.

1. Il sistema viene avviato come il caso precedente

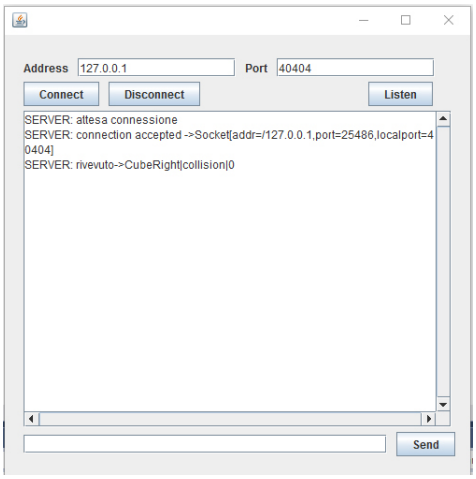


[Fig. 3.38]

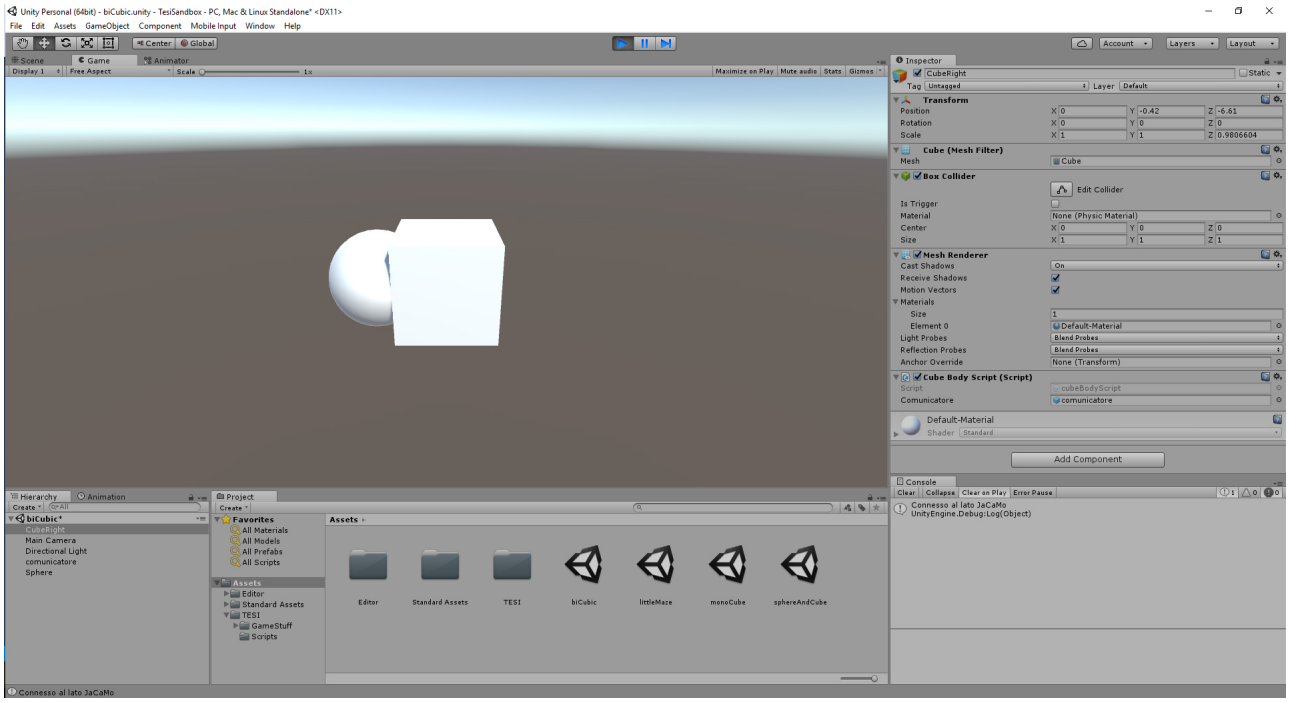


[Fig. 3.39]

2. Nel momento della collisione, il corpo invia un messaggio con la percezione della collisione



[Fig. 3.40]



[Fig. 3.41]

Capitolo 4

Conclusioni

Al momento però è solo possibile creare i corpi in Unity e le menti in JaCaMo.

Cosa succederebbe se fra qualche anno, una di queste due piattaforme, venisse rimpiazzata da una più performante e/o di più largo uso? E se invece di utilizzare questo middleware per collegare le menti di JaCaMo a corpi virtuali usati in simulazioni e videogiochi, si volessero applicare gli algoritmi di I.A. a corpi fisici come per esempio i droni?

In realtà, per come è stato progettato il middleware, è possibile implementare entrambe le parti (lato menti e lato corpi) su qualsiasi tecnologia a piacere, con il semplice requisito che sia possibile instaurare una connessione TCP.

Così facendo, le applicazioni sono pressoché infinite, per esempio:

- Si può creare un ambiente di simulazione su Unity e progettare una I.A. per il controllo di un robot, vedendo su Unity quale comportamento potrà avere. Dopodiché è possibile, usando gli stessi algoritmi ad agenti che si sono utilizzati su Unity, applicarli a un vero robot, dovendo aver cura solo di scrivere un interfaccia lato corpo, uguale a quella utilizzata nella simulazione su Unity.

- Si può creare una simulazione/videogioco, applicando delle I.A. create su piattaforme diverse da JaCaMo senza dover reimplementare il meddleware lato Unity.

- Al contrario, è possibile applicare I.A. scritte in JaCaMo ad altri ambienti di sviluppo per videogiochi, reimplementando il middleware solo lato corpi.

Grazie a questo middleware, sarà possibile sfruttare tecnologie ad agenti per la programmazione dei BOT (entità attive non controllate dal giocatore) e quindi di avere

un'architettura di riferimento per l'introduzione di tecniche di I.A. usufruendo di una piattaforma dedicata e non più cercando di modellare delle I.A. utilizzando la "classica" programmazione ad oggetti.

Bibliografia

[1] T.S. Kuhn, *La struttura delle rivoluzioni scientifiche*, Einaudi, 1969 (traduzione dall'orig. del 1962)

[2] liberamente tradotto da: Rafael H. Bordini, Jomi Fred Hubner, Michael Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason*, Wiley Series in Agent Technology, John Wiley & Sons, 2007

[3] liberamente tradotto da: Bratman ME 1990 *What is intention?* in *Intentions in Communication* (ed. Cohen PR, Morgan JL e Pollack ME), The MIT Press, Cambridge, MA.

[4] liberamente tradotta da: Ambra Molesini, Andrea Omicini, Alessandro Ricci e Enrico Denti. *Zooming multi-agent systems*. In Jorg P. Muller e Franco Zambonelli, editori, *Agent-Oriented Software Engineering VI*, volume 3950 di LNCS, pagine 81-93. Springer, 2006. 6th International Workshop (AOSE 2005), Utrecht, Olanda,25-26 Luglio 2005.

[5] liberamente tradotto da: Andrea Omicini, *Introduction to Distributed System*, Dipartimento di Informatica - Scienza e Ingeggneria (DISI), Alma Mater Studiorum - Università di Bologna a Cesena, Anno accademico 2015/2016

[6] liberamente tradotto da: *Jason, a Java-based interpreter for an extended version of AgentSpeak*. Tratto il 25 novembre da //jason.sourceforge.net/wp/

[7] liberamente tradotto da: *CArtAgO - Common ARTifacti infrastructure for AGents Open environments*. Tratto il 26 novembre da //cartago.sourceforge.net/

[8] liberamente tradotto da: *The Moise Organisation Oriented Programming Framework*. Tratto il 26 Novembre da //moise.sourceforge.net/

[9] liberamente tradotto da: *SpringerLink*. Tratto il 26 Novembre da //link.springer.com/article/10.1007%2Fs10458-009-9084-y

[10] Agente. *Treccani, la cultura italiana*. Tratto il 25 novembre 2016 da //www.treccani.it/vocabolario/agente/

[11] Serious game. (15 luglio 2016). *Wikipedia, L'enciclopedia libera*. Tratto il 25 novembre 2016 da //it.wikipedia.org/w/index.php?title=Serious_game&oldid=82043422.

[12] Intelligenza artificiale. (4 novembre 2016). *Wikipedia, L'enciclopedia libera*. Tratto il 25 novembre 2016 da //it.wikipedia.org/w/index.php?title=Intelligenza_artificiale&oldid=84134049.

[13] Programmazione imperativa. (18 ottobre 2016). *Wikipedia, L'enciclopedia libera*. Tratto il 25 novembre 2016 da //it.wikipedia.org/w/index.php?title=Programmazione_imperativa&oldid=83840632.

[14] Ragionamento. (12 dicembre 2015). *Wikipedia, L'enciclopedia libera*. Tratto il 25 novembre 2016 da //it.wikipedia.org/w/index.php?title=Ragionamento&oldid=77167737.

[15] Middleware. (21 ottobre 2016). *Wikipedia, L'enciclopedia libera*. Tratto il 28 novembre 2016 da //it.wikipedia.org/w/index.php?title=Middleware&oldid=83888333.

[16] Ingegno. *Wordreference.com — dizionari di lingua online*. Tratto il 25 novembre 2016 da //www.wordreference.com/definizione/ingegno

[17] Agire. *Wordreference.com — dizionari di lingua online*. Tratto il 25 novembre 2016 da //www.wordreference.com/definizione/agire

[18] Pensiero. *Wordreference.com — dizionari di lingua online*. Tratto il 25 novembre 2016 da //www.wordreference.com/definizione/pensiero

Elenco delle figure citate