ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

SYNAPSIS - MIDDLEWARE PER L'INTEGRAZIONE DI GAME ENGINE E SISTEMI MULTI-AGENTE

Tesi in:
Sistemi Autonomi

Tesi di Laurea di:
LUCA PASCUCCI

Relatore:
Prof. ANDREA OMICINI

Co-relatori:
Dott. STEFANO MARIANI

ANNO ACCADEMICO 2018–2019
SESSIONE II

PAROLE CHIAVE

Sistema Multi-Agente
Game Engine
Middleware
Play Framework
Ambiente virtuale

"Dedico questo mio lavoro che si pone a conclusione di un lungo e gioioso cammino alle persone che mi hanno aiutato ed insegnato"

Indice

Introduzione

L'ambiente, per definizione, è tutto ciò che circonda e con cui interagisce un organismo [36]. I Sistemi Multi-Agente (MAS) forniscono diverse astrazioni per la costruzione di un sistema software, ma le tecnologie disponibili risultano spesso carenti sotto il punto di vista della costruzione dell'ambiente (virtuale) in cui gli agenti operano, poiché si concentrano solamente sulla definizione di un ambiente computazionale esclusivamente logico, slegato dal mondo fisico (che può invece essere rappresentato sul piano virtuale).

Le Game Engine (GE), ovvero framework utilizzati per supportare la progettazione e lo sviluppo di videogiochi, viceversa, sono sempre più spesso utilizzate al di fuori dell'ambito video-ludico per rappresentare e gestire ambienti (virtuali) complessi, potenzialmente riflesso di un ambiente fisico, ad esempio in scenari di simulazione tattica (militare, di soccorso, etc.). In particolare, di recente le GE sono state utilizzate come mezzo abilitante per la coordinazione [19] all'interno di MAS.

L'ambiente, o scena, secondo la terminologia dei videogiochi, è una parte fondamentale che permette al giocatore di entrare in sinergia con il tipo, lo scopo e le modalità del gioco. Un esempio lampante è un FPS[1] dove la scena è solitamente vista in prima persona dal videogiocatore, e la presenza di elementi con i quali interagire, crea interesse nel giocatore ad esplorare l'ambiente attorno a lui.

In questa tesi si intende studiare lo stato di integrazione tra Game Engine (GE) e Sistemi Multi-Agente (MAS), per proporre un'infrastruttura generica utilizzabile per diversi scenari di associazione e comunicazione tra MAS e GE rispettandone il disaccoppiamento e l'integrità concettuale delle loro astrazioni.

[1] First-Person Shooter = sparatutto in prima persona

Il primo capitolo introduce il lettore ai Sistemi Multi-Agente (MAS), alle Game Engine (GE) ed alle integrazioni già realizzate in passato. Vengono dunque illustrate le astrazioni presenti nei MAS, le GE e si studia lo stato dell'arte delle integrazioni.

Il secondo capitolo, poi, analizza i differenti modelli computazionali delle tecnologie utilizzate, al fine di definire delle linee guida all'integrazione dei due sistemi, e descrive la struttura del sistema di integrazione.

Il terzo capitolo contiene una descrizione più approfondita degli elementi che compongono il middleware, la tecnologia di comunicazione utilizzata per collegare le diverse parti del sistema e le librerie sviluppate per mettere in comunicazione MAS e GE attraverso il middleware.

Il quarto capitolo contiene il caso di studio preso in esame per convalidare il sistema precedentemente delineato, mostrando struttura dei componenti realizzati e illustrando il flusso di interazione tra essi. Infine, nel Capitolo 5 si discuterà circa le conclusioni e verranno forniti spunti di riflessione per eventuali lavori futuri.

La principale motivazione che ha portato a questo studio risiede nel fatto che seppur la ricerca sui MAS ha prodotto modelli ricchi di astrazioni anche per la modellazione della dimensione ambientale, oltre a quella agentesca e sociale, le tecnologie che dovrebbero reificare tali modelli sono più rare e spesso limitate. Uno degli esempi più ricchi è costituito dal framework CArtAgO [27], non a caso sfruttato per il design e l'implementazione dell'infrastruttura proposta in questa tesi.

Esiste un divario nel progresso tecnologico che le GE hanno raggiunto rispetto al livello tecnologico delle infrastrutture orientate agli agenti nate all'interno della comunità accademica. Ciò non dovrebbe sorprendere nessuno: il settore video-ludico può contare su un maggiore supporto economico e su milioni di sviluppatori e tester (oltre ai giocatori), che sono ben pagati per spingere stabilità, prestazioni, usabilità dei loro prodotti a livelli di qualità senza precedenti e senza pari. Pertanto, vale la pena considerare la possibilità di trarre vantaggio da tali prodotti finemente ottimizzati per migliorare la qualità delle tecnologie mirate alla rappresentazione e gestione di ambienti virtuali nei MAS [10].

Una integrazione MAS – GE non darebbe benefici solo la "mondo MAS". C'è una lacuna nelle astrazioni concettuali e progettuali che la GE forni-

sce agli sviluppatori rispetto alle astrazioni molto più ricche che offrono soluzioni di ingegneria del software orientate agli agenti. La GE presa in considerazione in questo documento, Unity [34], ad esempio fornisce astrazioni di livello molto basso, specialmente lato "personaggi attivi", dove, ad esempio, programmare un comportamento ciclico e orientato a un goal equivale a scrivere coroutine che attraversano più fasi di rendering—goal e piani per raggiungerli non sono modellati da astrazioni di prima classe.

Inoltre, l'integrazione di MAS con GE può fornire nuove soluzioni per affrontare le problematiche tipiche degli scenari di realtà aumentata in cui si richiede che l'agente sia consapevole dello spazio in un ambiente fisico, sfruttando le tecnologie a disposizione nelle GE. D'altro canto tale integrazione mette a disposizione nuove funzionalità in grado di realizzare, all'interno dell'ecosistema GE, oggetti autonomi come ad esempio gli NPC[2] utilizzando le astrazioni presenti nei MAS.

[2]Non-Player Character: è un personaggio che non è sotto il controllo diretto del giocatore, ma viene invece gestito dal game master o dalla IA del software nel caso dei videogiochi

Capitolo 1

Background

1.1 Stato dell'arte

In questa sezione è presente una introduzione ai Sistemi Multi-Agente (MAS), alle Game Engine (GE) ed alle integrazioni già realizzate.

1.1.1 Sistema Multi-Agente

La crescente complessità nell'ingegnerizzazione dei sistemi software ha portato alla necessità di modelli e astrazioni in grado di rendere più facile la loro progettazione, lo sviluppo e il mantenimento. In questa direzione, la computazione orientata agli agenti viene in aiuto agli ingegneri ed informatici per costruire sistemi complessi, virtuali o artificiali permettendo una loro agevole e corretta gestione [22].

In particolare, la ricerca e le tecnologie per MAS hanno introdotto nuove astrazioni per affrontare la complessità durante la progettazione di sistemi o applicazioni composte da individui che non agiscono più da soli ma all'interno di una società. Le tecnologie e i modelli agent-oriented sono attualmente diventati una potente tecnica in grado di affrontare molti problemi che vengono alla luce durante la progettazione di sistemi computer-based in termini di entità che condividono caratteristiche quali l'autonomia, l'intelligenza, la distribuzione, l'interazione, la coordinazione, etc.

L'ingegnerizzazione dei MAS si occupa infatti di costruire sistemi complessi dove più entità autonome chiamate agenti cercano di raggiungere in

maniera proattiva i loro scopi sfruttando le interazioni tra di essi (come una società), e con l'ambiente circostante. Questo modello può essere visto come un paradigma general-purpose, il quale prevede l'utilizzo di tecnologie agent-oriented in diversi scenari applicativi [38].

Un MAS fornisce agli sviluppatori e ai designer tre astrazioni principali:

- Agenti: Le entità autonome che compongono il sistema. Sono in grado di comunicare e possono essere intelligenti, dinamici, e situati;
- Società: Rappresenta un gruppo di entità il cui comportamento emerge dall'interazione tra i singoli elementi;
- Ambiente: Il "contenitore" in cui gli agenti sono immersi e con il quale questi ultimi possono interagire, modificandolo. La caratteristica degli agenti di essere situati nell'ambiente in cui si trovano permette loro di percepire e produrre cambiamenti su di esso.

1.1.2 Game Engine

Le GE sono framework utilizzati per supportare la progettazione e lo sviluppo di giochi. Il termine "Game Engine" nacque a metà degli anni '90 in riferimento a giochi sparatutto in prima persona (FPS) come il popolare "Doom" progettato con una separazione ragionevolmente ben definita tra i suoi componenti software principali (come il sistema di rendering grafico tridimensionale, il sistema di rilevamento delle collisioni o il sistema audio) e le risorse artistiche, i mondi di gioco e le regole di gioco che comprendevano l'esperienza di gioco del giocatore.

La maggior parte delle GE sono realizzate con cura e messe a punto per eseguire un gioco particolare su una particolare piattaforma hardware. L'avvento di hardware per computer sempre più veloce e schede grafiche specializzate, insieme a algoritmi di rendering e strutture di dati sempre più efficienti, sta cominciando ad ammorbidire le differenze tra i motori grafici di diversi generi. È ora possibile utilizzare un motore sparatutto in prima persona per creare un gioco di strategia, ad esempio. Tuttavia, esiste ancora il compromesso tra generalità e ottimizzazione. Un gioco può sempre essere reso più impressionante perfezionando il motore in base ai requisiti e ai vincoli specifici di una determinata piattaforma di gioco e/o hardware [10].

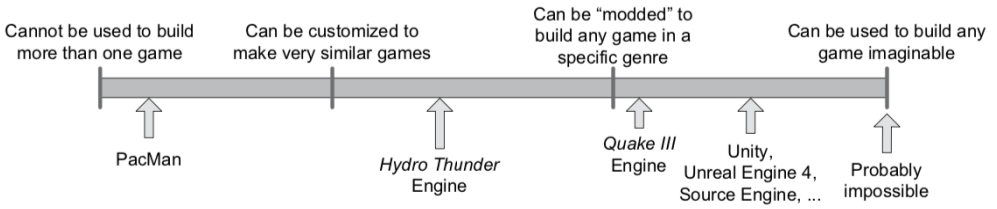

Figura 1.1: Game Engine reusability gamut [10]

Le GE moderne sono strutture general-purpose multipiattaforma orientate verso ogni aspetto della progettazione e dello sviluppo del gioco, come il rendering 2D/3D delle scene di gioco, i motori fisici per la dinamica ambientale (movimenti, dinamica delle particelle, rilevamento delle collisioni, prevenzione degli ostacoli, ecc.), suoni, script comportamentali, intelligenza artificiale dei personaggi e molto altro.

Come esempio significativo che rappresenta la gamma di piattaforme disponibili, nella sezione 1.3.4 verrà esaminata una delle più popolari GE - Unity [34] - con l'obiettivo di:

- rilevare quelle astrazioni e quei meccanismi che hanno più probabilità di avere una controparte nel MAS, o almeno quelli che sembrano fornire un supporto nel riformulare le astrazioni mancanti del MAS
- evidenziare le opportunità per colmare le lacune concettuali / tecniche che ostacolano l'integrazione dei due mondi

1.1.3 Integrazione

Sono già presenti esempi di integrazione tra GE e MAS che concentrano la propria attenzione su obiettivi specifici a livello tecnologico, piuttosto che sulla creazione di un'infrastruttura orientata agli agenti basata sul gioco per scopi generici. Per esempio:

- QuizMASter [2] concentrato sull'astrazione degli agenti collegando gli agenti MAS ai personaggi dei motori di gioco, nel contesto dell'apprendimento educativo

- CIGA [35] considera sia la modellazione degli agenti che quella dell'ambiente, per agenti virtuali generici in ambienti virtuali
- GameBots [15] concentrato sull'astrazione dell'agente, ma considera anche l'ambiente fornendo un framework di sviluppo e un runtime per i test di sistemi multi-agente in ambienti virtuali
- UTSAF [25] si concentra sulla modellistica ambientale nel contesto di simulazioni distribuite in ambito militare[1]

Sebbene rappresentino chiaramente esempi di integrazione (parzialmente) riuscita di MAS in GE, i lavori sopra elencati presentano alcune carenze rispetto all'obiettivo che perseguiamo in questo documento.

Solamente CIGA rappresenta un'eccezione che riconosce il divario concettuale tra MAS e GE, e propone soluzioni per affrontarlo (anche se a livello tecnologico). L'unico strato preso in considerazione nel perseguimento dell'integrazione è quello tecnologico - nessun modello, nessuna architettura, nessun linguaggio. All'interno di QuizMASter, UTSAF e GameBot (in una certa misura) l'integrazione è realizzata per specifico obiettivo, e la maggior parte degli approcci fornisce ai programmatori alcune astrazioni per trattare con agenti e ambiente, ma nessuna attenzione viene data alle astrazioni sociali [19].

1.2 Situazione inziale

I lavori precedentemente svolti, che hanno contribuito alla definizione di questo percorso, utilizzano due approcci nettamente separati per l'integrazione MAS e GE:

1. Implementazione delle caratteristiche dei MAS all'interno della GE, che funge dunque da "contenitore" del MAS stesso;
2. Realizzazione di un middleware, come layer software, per collegare l'ambiente MAS con GE, che restano quindi separate

[1] Gli agenti vengono considerati, ma solo come mezzo di integrazione tra diverse piattaforme di simulazione, non nel contesto del GE sfruttato per il rendering di simulazione

1.2.1 MAS all'interno di GE

Il primo punto è stato realizzato implementando due modelli tipici dei MAS:

- Il modello Beliefs, Desires, Intentions (BDI) per la programmazione degli agenti [24];
- Un modello di coordinazione degli agenti tramite spazio di tuple e primitive Linda [6][1];

Il cuore pulsante di entrambi i lavori risiede nell'uso intensivo di un interprete Prolog fatto ad hoc per Unity, UnityProlog [11]. Questo interprete dispone di molte funzionalità per estendere l'interoperabilità di Prolog con i GameObject. Dal momento che è stato progettato per essere usato in maniera specifica con Unity, nasce con delle primitive che permettono di accedere e manipolare GameObject e i relativi componenti direttamente da Prolog. UnityProlog introduce tuttavia alcune limitazioni da tenere bene in considerazione [24], anche se allo stato attuale è l'unica versione di Prolog del quale è stato dimostrato il corretto funzionamento:

- Un interprete per Prolog non sarà mai performante quanto lo può essere un compilatore e questo può rappresentare un problema per simulazioni di MAS piu grandi.
- Utilizza lo stack C# come stack di esecuzione, quindi la tail call optimization non è ancora supportata.
- Non supporta regole con più di 10 subgoal, quindi a fronte di una regola complessa con tanti goal da controllare, è necessario frammentare la regola in questione in sotto regole con non più di 10 subgoal per ognuna.

1.2.2 MAS e GE separati

Il secondo percorso si differenzia dal primo per la scelta di lasciare separati GE da MAS realizzando un canale di comunicazione tra i due ambienti. É stata introdotta una terminologia per contraddistinguere le entità realizzate sul GE (GameObject) e su MAS (agenti), rispettivamente definite "corpi" e "menti" virtuali [8].

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. Per rendere possibile questa comunicazione è stato progettato e implementato un sistema middleware definito secondo il seguente schema.

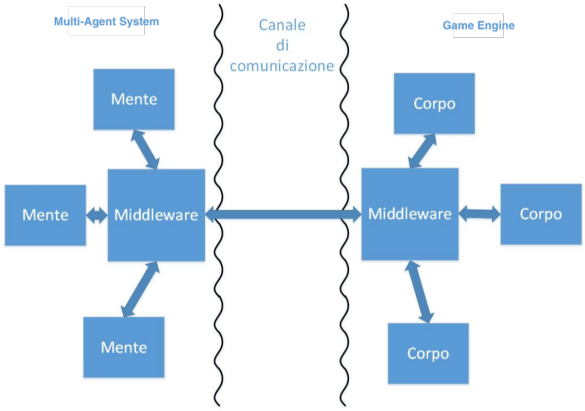

Figura 1.2: Il middleware viene suddiviso in due parti, poste sui due lati del canale di comunicazione [8]

Dallo schema (Figura 1.2) si può notare la separazione del middleware nei due sistemi, motivato dalle diverse tecnologie utilizzate dai due ambienti. Questa divisione vincola la realizzazione di una nuova parte di middleware in caso di utilizzo di un diversa tipologia di GE e/o MAS.

Il protocollo di comunicazione tra le entità è stato realizzato utilizzando messaggi strutturati. Da una parte, le menti devono definire quale azione deve compiere il relativo corpo (es. "muoviti in avanti", "ruota", "prendi", 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.) [8].

Nella sezione 2.3 è definita l'architettura del sistema realizzato evidenziando similitudini e differenze rispetto a questa soluzione.

1.3 Stack Tecnologico

Di seguito vengono illustrate le principali tecnologie utilizzate per l'effettiva realizzazione del middleware di collegamento dei due sistemi , stabilendo anche quale piattaforma per Sistema Multi-Agente (MAS) e Game Engine (GE) sono stati presi come principale riferimento.

1.3.1 JaCaMo

JaCaMo è un framework per la programmazione orientata agli agenti che combina tre tecnologie già affermate e sviluppate da diversi anni.

Un sistema multi-agente JaCaMo o, equivalentemente, un sistema software programmato con JaCaMo è definito da un'organizzazione Moise di agenti BDI autonomi basati su concetti come ruoli, gruppi, missione e schemi, implementati tramite Jason, che lavorano in ambienti condivisi distribuiti basati su artefatti, programmati in CArtAgO.

Ognuna delle tre tecnologie indipendenti che compongono il framework ha il proprio set di astrazioni, modelli di programmazione e meta-modelli di riferimento, per questo motivo in JaCaMo è stato realizzato un meta-modello globale, con l'obiettivo di definire le dipendenze, connessioni, mapping concettuali e le sinergie tra le differenti astrazioni rese disponibili da ogni livello [4].

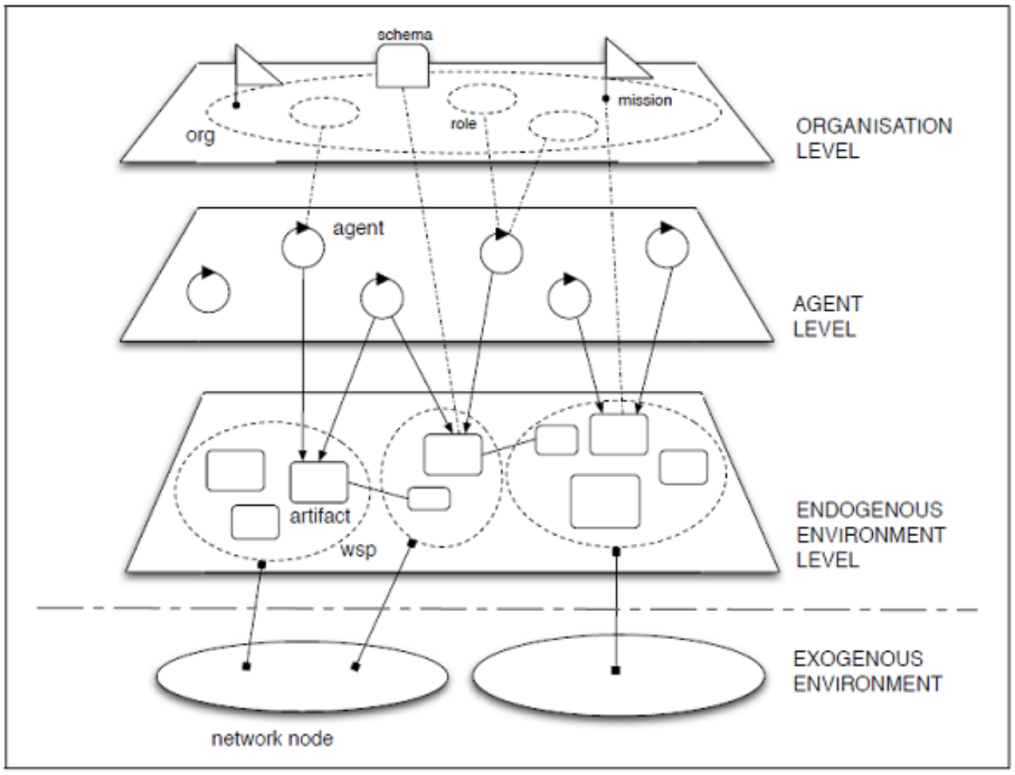

Figura 1.3: Livelli che compongono il framework JaCaMo [4]

1.3.1.1 Jason

Jason è un interprete di AgentSpeak che implementa la semantica operazionale del linguaggio e fornisce una piattaforma di sviluppo per Sistemi Multi-Agente, con molte funzionalità personalizzabili dall'utente.

Le astrazioni appartenenti alla dimensione degli agenti, correlate al metamodello Jason, sono principalmente ispirate all'architettura BDI sulla quale Jason è radicato. Quindi un agente è un'entità composta da un insieme di "beliefs", che rappresenta lo stato corrente e la conoscenza dell'agente sull'ambiente in cui si trova, una serie di "goals", che corrispondono a compiti che l'agente deve perseguire e una serie di "plans" ossia sequenze di azioni (external action or internal action), innescate da eventi, che gli agenti possono comporre, istanziare ed eseguire dinamicamente per compiere i "goals" [5].

1.3.1.2 CArtAgo

Per quanto riguarda l'ambiente, ciascuna istanza dell'ambiente CArtAgO[2] è composta da una o più entità workspace. Ogni workspace è formato da un insieme di artefatti, che forniscono un insieme di operazioni e proprietà osservabili, definendo anche l'interfaccia di utilizzo degli artefatti. L'esecuzione dell'operazione potrebbe generare aggiornamenti delle proprietà osservabili e degli eventi osservabili specifici. L'ultima entità relativa all'ambiente è il "manual", un'entità utilizzata per descrivere le funzionalità fornite da un artefatto. Cartago è basato sul meta-modello A&A (Agents & Artifacts), che definisce 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 [27].

Artefatto L'artefatto è un'entità reattiva, non autonoma, stateful, riutilizzabile, controllabile ed osservabile. Modellano strumenti, risorse e porzioni di ambiente agendo da strumenti mediatori di azioni e interazioni sociali tra partecipanti individuali e lo stesso ambiente [21].

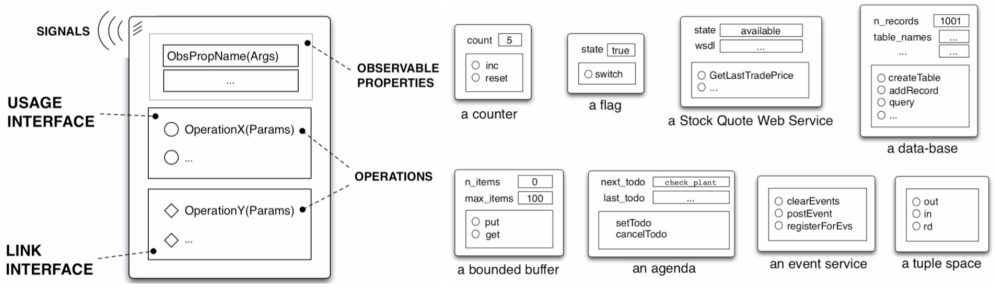

Figura 1.4: Struttura artefatto con relativi esempi

La modalità di interazione tra artefatto ed agente viene riepilogato nell'immagine sottostante.

[2]Common ARTifact infrastructure for AGents Open environments

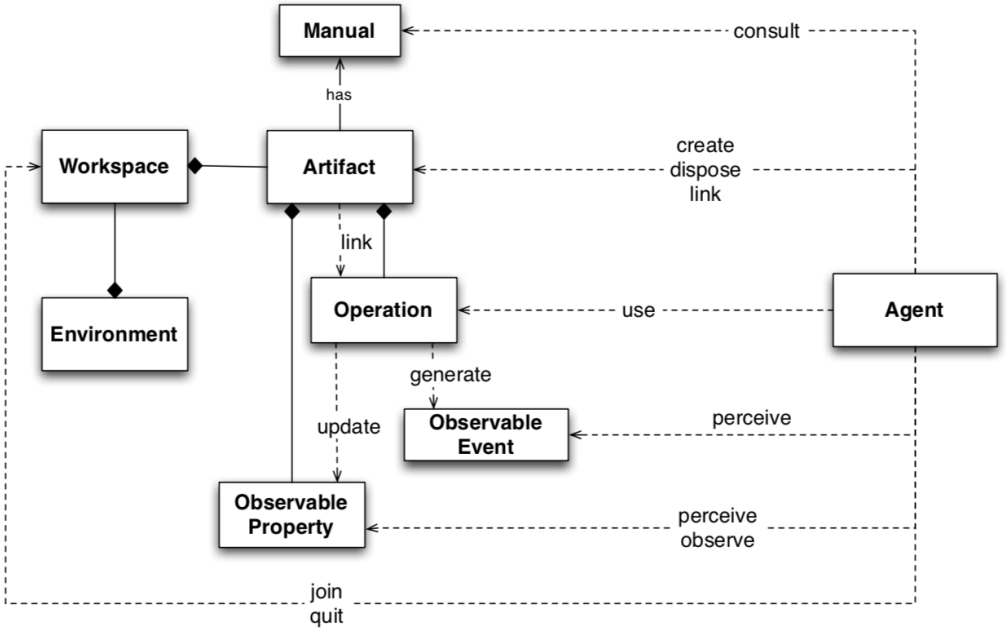

Figura 1.5: Interazione tra agente ed artefatto

1.3.1.3 Moise

Moise è un meta-modello organizzativo per MAS 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.

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 (essenzialmente, la proprietà di lavorare con un numero e una diversità di componenti che non è imposta una volta per tutte) e adattamento.

1.3.2 Play

Play è un framework lightweight, stateless e asincrono per la creazione di applicazioni e servizi Web. È stato costruito utilizzando Scala e Akka e

mira a fornire gli strumenti per la realizzazione di applicazioni altamente scalabili con consumo minimo di risorse, ad esempio CPU, memoria, thread [13].

Play incorpora un HTTP Server integrato (quindi non è necessario un server di separato come in molti Web Framework Java), un modello per la realizzazione di applicazioni baste su servizi RESTful e mette a disposizioni strumenti per la gestione di Form, protezione CSRF[3] e meccanismi di instradamento. Per semplificare il suo utilizzo fa largo uso del pattern Model-View-Controller, comune e facilmente utilizzabile, fornendo paradigmi di programmazione concisi e funzionali.

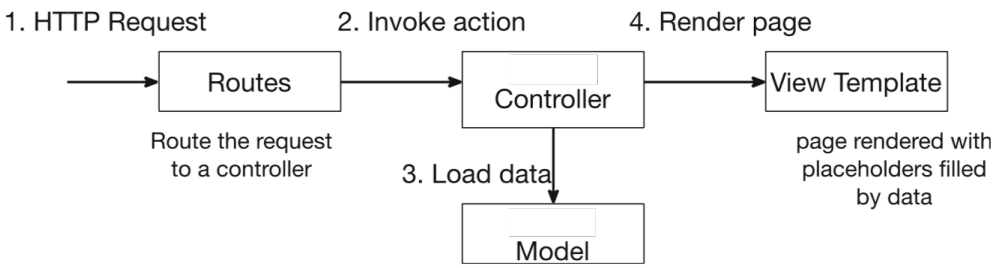

Figura 1.6: Struttura MVC di un'applicazione realizzata con Play [12]

Lo stack nelle Web Application nel mondo Java Enterprise è basato su una tecnologia che si è evoluta nel corso degli anni e richiede diversi elementi (strati) per funzionare. E' molto probabile che le molteplici tecnologie che comprendono questo stack rendano anche l'implementazione di semplici applicazioni problematica e soggetta a errori poiché ogni tecnologia deve essere integrata con successo con la successiva, spesso basandosi su file di configurazione o convenzioni standard [12].

[3]Cross-Site Request Forgery

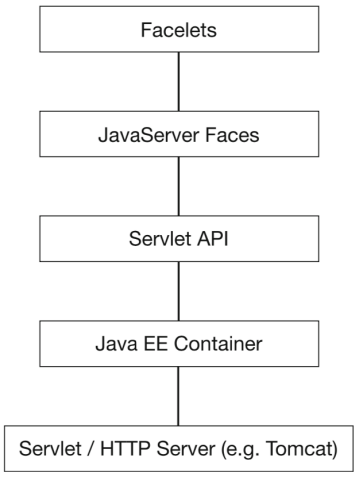

Figura 1.7: Architettura a strati JavaEE [12]

Il framework Play è stato progettato per diminuire lo stack (Figura 1.7) richiedendo l'utilizzo di un solo server HTTP per funzionare.

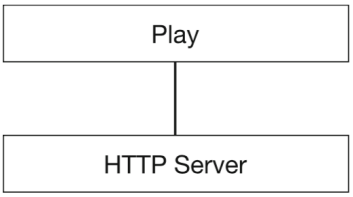

Figura 1.8: Architettura a strati Play framework [12]

Lo strato Play (Figura 1.8) è formato da una serie di componenti che includono:

- HTTP Server: componente che riceve la richiesta HTTP da un client e restituisce un risultato basato sulle informazioni fornite nella richiesta;
- Router: determina dove instradare la richiesta, pertanto fornisce un file di configurazione dei percorsi disponibili nell'applicativo;
- Sistema di templating HTML dinamico: Utilizza pagine standard in HTML e le popola con dati generati dinamicamente dall'applicazione;

- Console integrata: Per semplificare l'utilizzo di Play, viene fornita una suite di strumenti che possono essere utilizzati per creare, aggiornare e distribuire l'applicazione Play. Questi strumenti sono accessibili e gestiti dalla console;

- Persistent framework: Funzionalità utili per accesso a database.

1.3.2.1 Akka - Modello ad attori

Akka è un toolkit per la creazione di applicazioni altamente distribuite, concorrenti, event-driven, tolleranti ai guasti. Play framework utilizza il modello ad attori presente in Akka, dove l'attore è l'entità principale, per aumentare il livello di astrazione e fornire una piattaforma per la realizzazioni di applicazioni concorrenti e scalabili [16].

Il modello ad attori (che risale al 1973) si basa sull'idea di avere attori simultanei indipendenti che ricevono e inviano messaggi asincroni e che svolgono un comportamento basato su questi messaggi. Gli attori possono mantenere il proprio stato e comportamento. Tuttavia, idealmente solo i dati immutabili vengono scambiati tra di loro, pertanto ogni attore è indipendente da tutti gli altri ed esegue solo alcuni calcoli o elaborazioni basati su un messaggio ricevuto da esso.

L'idea chiave alla base del modello ad attori è che la maggior parte dei problemi come concorrenza, deadlock, corruzione dei dati, derivino dalla condivisione dello stato. Pertanto, nel mondo degli attori non esiste uno stato condiviso (come una coda concorrente produttore-consumatore). Al contrario, i messaggi vengono inviati tra attori e questi messaggi vengono messi in coda in una casella di posta in modo simile ai messaggi di posta elettronica [12].

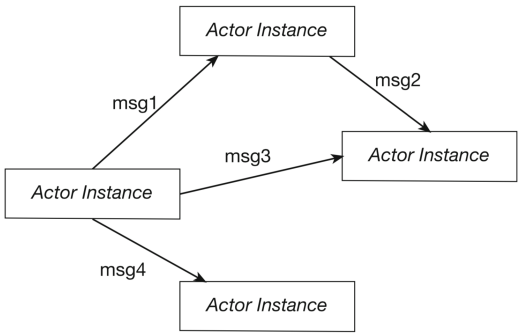

Figura 1.9: Comunicazione tra attori attraverso messaggi asincroni [16]

1.3.3 WebSocket

Il protocollo WebSocket (WS) consente la comunicazione bidirezionale tra un client a un host remoto instaurando un canale di comunicazione utilizzabile da entrambi sia in scrittura che in lettura. Il modello di sicurezza utilizzato è l'origin-based security model comunemente usato dai browser web. Il protocollo consiste in una fase di hand-shake di apertura seguita dal successivo invio/ricezione di una serie di messaggi strutturati, stratificata su una connessione TCP[4] persistente. L'obiettivo di questa tecnologia è fornire un meccanismo per le applicazioni basate su browser che necessitano di comunicazione bidirezionale in tempo reale con server che non si basano sull'apertura di più connessioni HTTP [7].

1.3.3.1 Introduzione delle WebSocket

Storicamente, la creazione di applicazioni Web che richiedono la comunicazione bidirezionale tra client e server (ad esempio applicazioni di messaggistica istantanea e giochi) ha portato ad un abuso di HTTP utilizzato per operazioni di polling (verifica ciclica) verso il server con lo scopo di controllare gli aggiornamenti, inviando notifiche upstream come chiamate HTTP distinte [18].

Il protocollo HTTP è stato inteso fin dalla prima versione ideata da Tim Berners-Lee come metodo per recuperare risorse remote in maniera semplice: una richiesta per ogni pagina Web, ogni immagine o l'invio di dati

[4]Transmission Control Protocol

da rendere persistente. Con il passare degli anni però, all'incirca intorno al 2004, lo sviluppo di applicazioni Web subì una forte accelerazione dovuta all'introduzione di una nuova tecnologia, Ajax, che grazie all'utilizzo di Javascript fu in grado di creare e gestire richieste HTTP asincrone tramite funzioni di callback dedicate.

Seguendo l'evoluzione delle applicazioni Web molte applicazioni prevedevano una user experience orientata al real-time. Esempi possono essere applicazioni di chat, videogiochi multiplayer o sistemi di notifiche, tutte applicazioni che la sola tecnologia Ajax (o simili, come connessioni HTTP persistenti COMET) poteva simulare solo in parte con sistemi di polling poco performanti e complessi da implementare.

La soluzione arrivò quando ci si rese conto che la risposta a questi problemi risiedeva effettivamente nel protocollo stesso: HTTP sfrutta a livello di rete il protocollo TCP/IP, connection-oriented, usata in altri contesti singolarmente per connessioni full-duplex. Nacque così il protocollo WebSocket con un ottimo tempismo considerando l'avvento contemporaneo di HTML5 e altre tecnologie che contribuiranno successivamente alla diffusione del protocollo [3].

1.3.3.2 Principali caratteristiche

Ecco le caratteristiche principali del protocollo WebSocket:

- **Bidirezionali**: quando il canale di comunicazione è attivo, sia il client che il server sono connessi ed entrambi possono inviare e ricevere messaggi;
- **Full-duplex**: i dati inviati contemporaneamente dai due attori (client e server) non generano collisioni e vengono ricevuti correttamente;
- **Basati su TCP**: il protocollo usato a livello di rete per la comunicazione è il TCP, che garantisce un meccanismo affidabile (controllo degli errori, re-invio di pacchetti persi, ecc) per il trasporto di byte da una sorgente a una destinazione;
- **Client-key handshake**: All'apertura di una connessione, il client invia al server una chiave segreta di 16 byte codificata con base64.

Il server aggiunge a questa un'altra stringa, detta GUID[5] specificata nel protocollo e codifica con SHA1 e invia il risultato al client. Cosi facendo, il client può verificare che l'identità del server che ha risposto corrisponda a quella desiderata;

- **Sicurezza origin-based**: Alla richiesta di una nuova connessione, il server può identificare l'origine della richiesta come non autorizzata o non attendibile e rifiutarla.
- **Maschera dei dati**: Nella trama iniziale di ogni messaggio, il client invia una maschera di 4 byte per l'offuscamento. Effettuando uno XOR bit a bit tra i dati trasmessi e la chiave è possibile ottenere il messaggio originale. Ciò è utile per evitare lo sniffing, cioè l'intercettazione di informazioni da parte di terze parti.

1.3.3.3 WebSocket URIs

La specifica RFC [7] stabilisce due differenti tipologie di URI[6] per rappresentare una risorsa remota di tipo WebSocket:

1. ws-URI = ws://HOST[:PORT]/PATH[?QUERY];
2. wss-URI = wss://HOST[:PORT]/PATH[?QUERY].

In entrambi i casi i seguenti parametri stanno a significare:

- HOST: l'host dove risiede la risorsa;
- PORT: la porta sul quale l'host rende disponibile la risorsa. Questo pramentro è opzionale e se non specificato viene inteso:
  - Porta 80 per connessioni WS
  - Porta 443 per connessioni WSS
- PATH: il percorso all'interno dell'host dove trovare la risorsa;
- QUERY: la query da compiere sulla risorsa.

[5] Globally Unique Identifier
[6] Uniform Resource Identifier

L'utilizzo di questo protocollo per lo svolgimento della tesi è stato favorito dalle caratteristiche coerenti al tipo di sistema che si intende creare, nonché dal supporto nativo presente nel framework Play e dalla facile reperibilità di librerie per JaCaMo e Unity:

- **websocket-sharp**: Implementazione C# del protocollo WebSocket client/server [28]
- **project tyrus**: Implementazione Java dello standard JSR 356[7] [23]

1.3.4 Unity

Unity è una Game Engine (GE) cross-platform sviluppata da Unity Technologies, utilizzata per la creazione di videogiochi (sia 2D che 3D) e simulazioni, che supporta la distribuzione su una larga varietà di piattaforme (PC, console, dispositivi mobili, etc.). Fornisce astrazioni che contribuiscono ad estendere il suo utilizzo tra gli sviluppatori e programmatori, rendendola una delle GE più utilizzate per produrre in maniera veloce ed efficace applicazioni e giochi [34].

Inoltre, questa GE supporta molte funzionalità facili da utilizzare e sfruttabili per creare giochi realistici e simulazioni immersive, come un intuitivo editor real-time, un sistema di fisica integrato, luci dinamiche, la possibilità di creare oggetti 2D e 3D direttamente dall'IDE o di importarli esternamente, gli shader, un supporto per l'intelligenza artificiale (capacità di evitare gli ostacoli, ricerca del percorso, etc.), e cosi via.

Le funzionalità principali messe a disposizione del designer sono:

- **GameObject**: La classe base per tutte le entità presenti su una scena di Unity: un personaggio controllabile dall'utente, un personaggio non giocabile, un oggetto (2D/3D). Tutto ciò che è presente sulla scena è un GameObject.
- **Script**: Codice sorgente applicato a un GameObject, grazie al quale è possibile assegnare a quest'ultimo comportamenti e proprietà dinamiche. Gli script vengono eseguiti dal game loop di Unity, che in maniera sequenziale esegue una volta ogni script, durante ogni frame del gioco.

[7] Java API per WebSocket, conforme al protocollo RFC6455

Non esiste concorrenza. Il comportamento è il risultato della logica definita nello script attraverso funzioni e routine. Le proprietà equivalgono a variabili che possono essere manipolate nello script oppure definite dall'IDE grafico.

- **Component**: Elemento, proprietà speciale assegnabile ai GameObject. A seconda del tipo di GameObject che si desidera creare è necessario aggiungere diverse combinazioni di Components. I Components basilari riguardano la fisica (Transform, Collider,..), l'illuminazione (Light) e la renderizzazione del GameObject (Render). É possibile istanziare runtime Components attraverso gli script.

- **Coroutine**: Una soluzione alla sequenzialità imposta agli script, grazie al quale è possibile partizionare una computazione e distribuirla su più frame, sospendendo e riprendendo l'esecuzione in precisi punti del codice.

- **Prefab**: Rappresentazione di un GameObject complesso, completo di Script e Component, istanziabile più volte a run-time. Le modifiche della struttura, proprietà e componenti del Prefab si propagheranno a tutti i GameObject collegati allo stesso presenti nella scena di gioco.

- **Event e Messaging System**: sistema ad eventi utile per far comunicare tra loro diversi GameObject. Questi sistemi sono formati tipicamente da eventi e listener. I listener si sottoscrivono ad eventi di un certo tipo; quando l'evento si verifica, viene notificato a tutti i listener in ascolto dello stesso tipo attraverso l'invio di un messaggio.

1.4 Realizzare un oggetto in Unity

Gli strumenti a disposizione permettono agli sviluppatori di realizzare qualunque tipo di oggetto: dai più semplici, come un cubo, ai più articolati, ad esempio un robot.

Per realizzare oggetti complessi è possibile definire una gerarchia di componenti e dotare ognuno di loro delle stesse funzionalità dell'oggetto padre. Ripensando all'esempio del robot, lo sviluppatore può suddividere lo stesso in sotto-componenti più articolate, quali testa, braccia, addome, gambe, fino ad arrivare a realizzare parti basilari quali dita, occhi e così via.

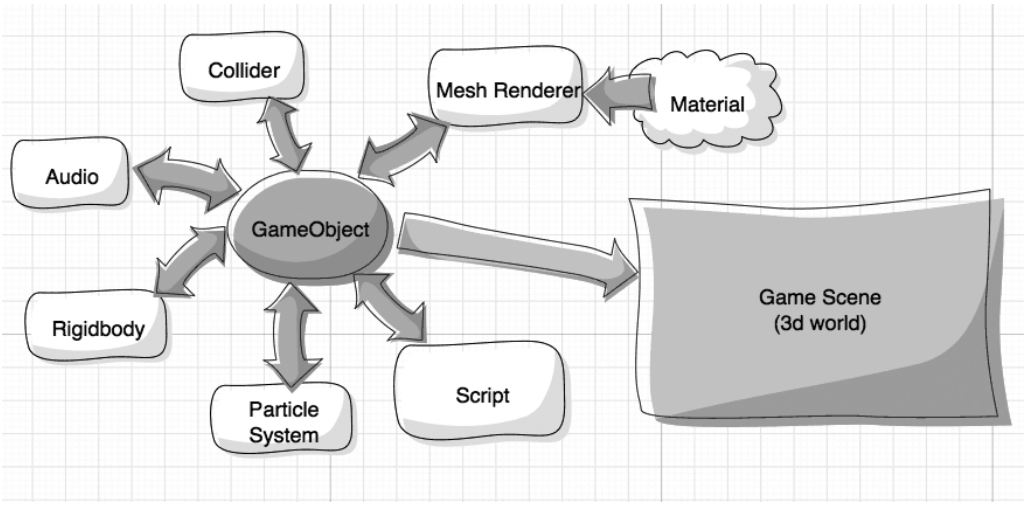

Figura 1.10: GameObject e Components

1.4.1 Muovere il GameObject

Come spiegato precedentemente, ogni oggetto presente su Unity è un GameObject e, per definizione, contiene le seguenti proprietà fondamentali:

- Posizione
- Scala
- Rotazione

Le proprietà appena elencate sono elementi fondamentali del "component" Transform [33], automaticamente realizzato per ogni oggetto in scena.

Attraverso l'associazione di uno script al GameObject, lo sviluppatore può accedere ad ogni proprietà di quest'ultimo. In tal modo è possibile modificare runtime la sua posizione, la scala e la rotazione e, applicando diverse tipologie di trasformazioni, lo si anima. Questa modalità di animazione è basilare, ma sufficiente per l'obiettivo finale posto. Sono poi presenti meccanismi complessi in caso di elaborazioni più articolate e specifiche come, ad esempio, la simulazione della corsa umana.

1.4.2 Fisicità del GameObject

La semplicità nella realizzazione di oggetti all'interno delle Game Engine è affiancata alla presenza di un motore fisico: attraverso quest'ultimo, Unity elabora e modifica dinamicamente ogni oggetto in scena in base alle specifiche fisiche ad esso attribuite. Il motore fisico rende possibile la simulazione di forza di gravità, la fisica dei movimenti e le occlusioni della scena.

In Unity, per definire la fisicità di un GameObject, è necessario attribuirgli uno specifico "component" chiamato RigidBody [32]. Aggiungendo questo componente, il movimento del GameObject nella scena è controllato dal motore fisico di Unity. Di questo componente è possibile specificare:

- Massa
- Resistenza
- Velocità di movimento
- Soggezione alla gravità

1.4.3 Percepire l'ambiente

La percezione generalmente è associata all'acquisizione di una realtà interna o esterna attraverso l'elaborazione organica e psichica di stimoli sensoriali [36]. Nelle Game Engine, rendere un oggetto capace di percepire la realtà è spesso collegato a renderlo fisicamente consapevole della propria superficie.

Su Unity è presente il "Collision Detection System", il quale controlla ogni evento di interazione fisica tra due o più GameObject nella scena e, attraverso l'aggiunta del "component" Collider al GameObject, viene specificato che l'oggetto deve essere preso in considerazione dal sistema durante la generazione di eventi di collisione.

Il Collider è associabile ad un GameObject, più o meno complesso, ed è capace di creare un'area generica, come ad esempio un cubo/sfera, che circonda l'oggetto, oppure mappare alla perfezione la sua superficie. Gli eventi di interazione creati dal "Collision Detection System" sono utilizzabili, da parte dello sviluppatore, nello script collegato al GameObject, difatti durante una collisione vengono invocati degli specifici metodi all'interno dello script con tutte le informazioni sull'evento emesso [29].

L'ultimo passaggio è fondamentale, dato che permette di completare il processo di percezione dell'ambiente da parte di un generico GameObject e, quindi, lo rende consapevole della propria presenza nella scena.

Per aumentare la capacità di percezione del GameObject, all'interno di Unity ogni oggetto può essere visto e utilizzato da ogni altro oggetto in scena. In questa maniera oltre alla percezione del proprio corpo fisico, il GameObject è in grado di conoscere anche quali altri elementi compongono la scena, ottenendo quindi la percezione totale dell'ambiente.

Tutte le procedure sopra illustrate sono replicabili per ogni GameObject presente in scena. In questo modo è possibile realizzare scene più o meno complesse.

1.4.4 Modificare l'ambiente

Il GameObject, come illustrato in precedenza, è la classe base di tutti gli oggetti presenti su una scena Unity, di conseguenza, l'ambiente stesso è un GameObject (più o meno complesso). Questo concetto, unito alla possibilità di ogni GameObject di interagire sia fisicamente che logicamente (script) con ogni altro oggetto in scena, dà luogo ad infinite possibilità di modifica della scena. Ad esempio, in caso di collisione tra due oggetti, il motore fisico, unito al motore grafico, calcola il possibile spostamento degli stessi che quindi porta ad un'effettiva modifica dell'ambiente.

Capitolo 2

Synapsis: Modello e architettura

In questo capitolo viene definita la terminologia usata nella restante parte della trattazione, vengono analizzati i differenti modelli computazionali delle tecnologie utilizzate per definire delle linee guida di integrazione dei sistemi e infine viene spiegata la struttura del sistema di integrazione.

2.1 Terminologia

La sinapsi (o giunzione sinaptica) (dal greco synàptein, vale a dire "connettere") è una struttura altamente specializzata che consente la comunicazione delle cellule del tessuto nervoso tra loro (neuroni) o con altre cellule (cellule muscolari, sensoriali). Nello specifico la sinapsi neuromuscolare rappresenta la giunzione tra neurone motore e muscolo a livello della placca motrice, ove ha luogo la trasmissione dell'impulso con le modalità delle sinapsi chimiche: lo spazio extracellulare della sinapsi neuromuscolare è detto chiave sinaptica [36]. La semplice associazione tra l'obiettivo di questo percorso e la parola sopra definita ha portato a denominare il middleware "Synapsis"[1].

2.1.1 Entità

Successivamente nella trattazione verrà fatto uso del termine "entità" che generalmente viene intesa come insieme di elementi dotati di proprietà comuni dal punto di vista dell'applicazione considerata [36]. Concettualmente,

[1] Traduzione in inglese del termine italiano sinapsi.

in questo dominio, l'entità viene intesa come oggetto divisibile in due parti, mente e corpo, che collegate riescono a trasmettersi informazioni, utilizzate dalla mente per raggiungere i propri obiettivi e dal corpo per diventare "attivo" nell'ambiente in cui si trova.

2.1.2 Mente

La nozione di mente può essere caratterizzata da alcuni punti chiave fondamentali:

- autonomia;
- interazione;
- obiettivi.

In altre parole, una mente può essere pensata come un componente software autonomo che interagisce con l'ambiente per svolgere i propri compiti. I punti sopra elencati rendono facile l'associazione della mente al concetto di Agente, spiegato nella sezione 1.1.1, poiché questa entità del Sistema Multi-Agente (MAS) ingloba astrazioni simili a quelle illustrate nella sezione 1.3.1.1.

2.1.3 Corpo

Corpo è un termine generico che indica qualsiasi porzione limitata di materia, cui si attribuiscono, in fisica, le proprietà di estensione, divisibilità, impenetrabilità [36]. In questa trattazione è associabile alla nozione di GameObject di Unity, spiegata nella sezione 1.3.4, utilizzata per avere una rappresentazione fisica dell'entità da realizzare.

2.1.4 Azione

Nel suo significato più generale è intesa come attività od operazione posta in essere da un determinato soggetto [36]. In questo studio, si considera come "azione" un certo gesto richiesto dalla mente che può essere associato ad una operazione eseguita dal corpo, ad esempio, nel caso di un'azione del tipo *"vai a (posizione)"*, richiesta dalla mente, corrisponde il movimento del corpo nell'ambiente verso la posizione indicata.

2.1.5 Percezione

La percezione è un atto cognitivo mediato dai sensi con cui si avverte la realtà di un determinato oggetto e che implica un processo di organizzazione e interpretazione [36].

In questo lavoro, la percezione si collega ad una certa sensazione rilevata dal corpo ed inviata alla mente per portarla a conoscenza di questa nuova informazione, ad esempio, nel caso del raggiungimento della posizione richiesta in precedenza, il corpo trasmette la percezione *"arrivato (posizione)"* che informa la mente del completamento dell'operazione.

Esiste inoltre, da parte del corpo, la possibilità di inviare percezioni "libere" ossia non associate a risposta di un'azione inviata dalla mente. Un semplice esempio è il contatto del corpo con una qualsiasi altra entità nell'ambiente che corrisponde all'invio di una percezione del tipo *"toccato(nome_entità)"*.

2.1.6 Struttura di un'entità

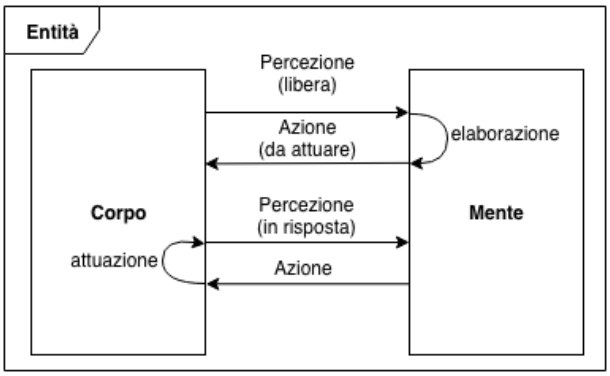

Figura 2.1: Struttura di una generica entità

La figura 2.1 rappresentata la struttura di una generica entità, dove:

- Il corpo esegue azioni e, in risposta a queste ultime, oppure, a seguito di determinati eventi esterni, trasmette le proprie percezioni alla mente.

- La mente elabora le percezioni per decidere quali azioni far svolgere al proprio corpo.

2.2 Modelli computazionali

Prima di definire l'architettura del sistema, è stato necessario analizzare i diversi modelli computazionali delle tecnologie utilizzate al fine di effettuare una coerente integrazione tra i sistemi utilizzati.

2.2.1 JaCaMo

2.2.1.1 Jason - BDI Agent Model

Gli agenti rappresentano l'astrazione principale dei MAS: sono entità proattive che incapsulano il controllo, governandolo attraverso azioni che consentono all'agente stesso di perseguire i propri obiettivi (ovvero, ciò che vuole ottenere) usando e cambiando qualcosa nel mondo in cui sono immersi (percependo lo stato dell'ambiente e adattando le proprie azioni e il proprio comportamento in base ad esso). In questo senso, gli agenti sono situati, strettamente uniti con il contesto e l'ambiente circostante e, cosa più importante, sono sociali: esprimono autonomia nelle interazioni tra agenti, come avviene in una società.

Un particolare tipo di agente è quello presente in JaCaMo, sviluppato secondo l'architettura BDI. É possibile, e abbastanza comune, pensare ad un agente BDI come un sistema razionale con atteggiamenti mentali [26], vale a dire credenze (Beliefs), desideri (Desires), e intenzioni (Intentions), le quali rappresentano rispettivamente ciò che l'agente conosce del mondo, cosa lo motiva e cosa sta facendo per raggiungere i propri obiettivi. Come proposto in [26], il ciclo di reasoning di un agente è composto da quattro fasi principali:

1. Generazione di opzioni: l'agente restituisce un insieme di opzioni in base a cosa conosce riguardo al mondo e quali sono i suoi desideri;

2. Deliberazione: l'agente seleziona un sottoinsieme delle opzioni precedentemente selezionate;

3. Esecuzione: l'agente esegue, se è presente la relativa intenzione, un'azione tra le opzioni precedentemente selezionate;

4. Percezione: l'agente infine aggiorna la sua conoscenza del mondo (ed eventualmente sè stesso).

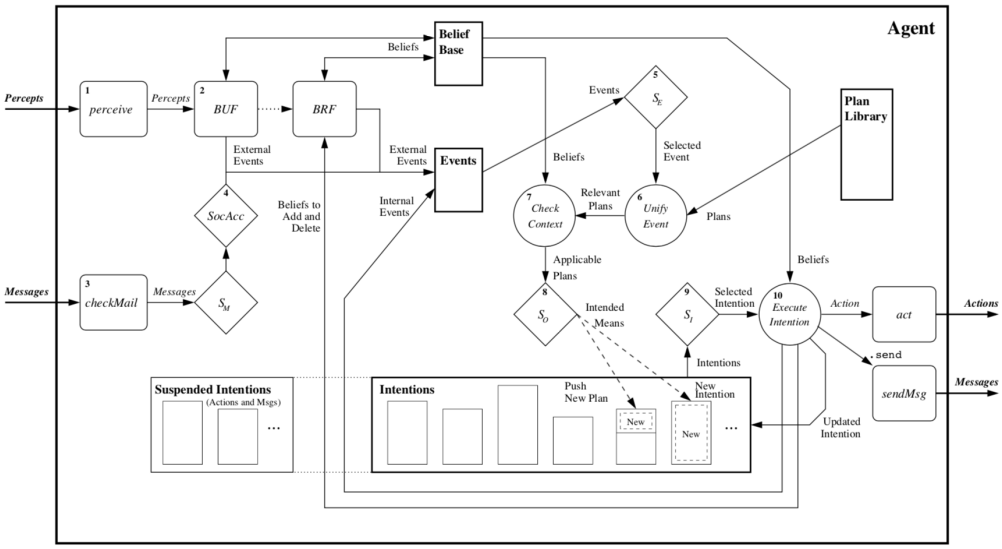

Figura 2.2: Architettura BDI di un agente [5]

Durante l'analisi delle astrazioni disponibili su Jason è risultato particolarmente utile il concetto di "belief". La Belief Base, raffigurata nell'immagine 2.2, è il contenitore di conoscenza dell'agente che viene in parte modificata dalle percezioni esterne ricevute dall'ambiente.

Tali percezioni sono facilmente riconducibili agli eventi che un GameObject può ricevere durante la sua permanenza nell'ambiente (sezione 1.4), come, ad esempio la notifica di una collisione con un secondo GameObject in scena. Questo concetto ha portato alla decisione di utilizzare il belief come strumento per "notificare" all'agente le percezioni inviate dal GameObject.

2.2.1.2 CArtAgO

L'architettura astratta di CArtAgO (e degli ambienti di lavoro di CArtAgO) è composta da tre elementi costitutivi principali (vedi Fig. 2.3): (i) corpi

degli agenti - in quanto entità che rendono possibile situare agenti all'interno dell'ambiente di lavoro; (ii) artefatti: come elementi di base per strutturare l'ambiente di lavoro; e (iii) aree di lavoro, come contenitori logici di artefatti, utili per definire la topologia dell'ambiente di lavoro [27].

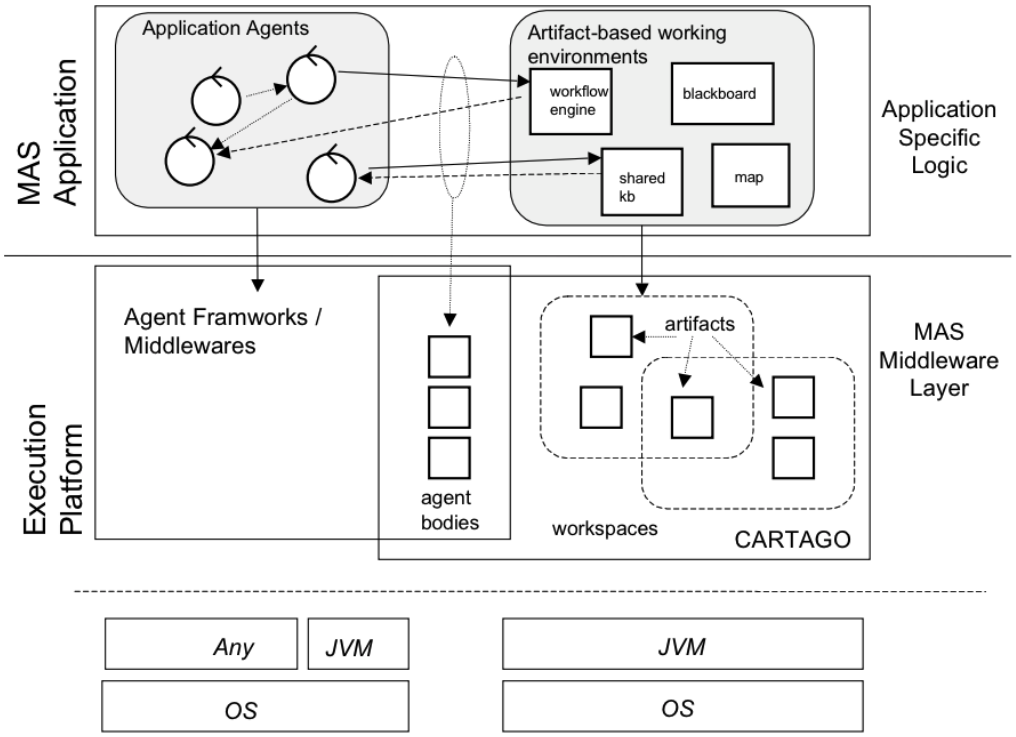

Figura 2.3: Livelli del MAS che adottano il supporto CArtAgO. Gli ambienti applicativi sono modellati in termini di ambienti di lavoro basati su artefatti. Il middleware CArtAgO gestisce il ciclo di vita degli ambienti di lavoro, composto da artefatti raggruppati in aree di lavoro. I corpi degli agenti vengono utilizzati per collocarli all'interno degli ambienti di lavoro, eseguendo azioni su artefatto e percependo artefatti osservabili stato ed eventi [27].

Analogamente ai manufatti nella nostra società, il modello di base che caratterizza l'interazione tra agenti e manufatti si basa su una nozione di uso e osservazione. Gli agenti possono utilizzare un artefatto innescando l'esecuzione delle operazioni elencate nell'interfaccia di utilizzo dell'artefat-

to. Un'operazione è caratterizzata da un nome e un insieme di parametri digitati. L'esecuzione di un'operazione provoca, in genere, l'aggiornamento dello stato interno di un artefatto e potenzialmente la generazione di uno o più eventi osservabili - comprese le condizioni di errore - che possono essere eventualmente raccolti dal sensore degli agenti quando vengono generati e percepiti attraverso azioni di rilevamento esplicite.

L'artefatto è computazionalmente associabile al concetto di monitor. Un monitor è un costrutto di sincronizzazione di un linguaggio di alto livello. Un'istanza di un tipo monitor può essere utilizzata da due o più processi o thread per rendere mutuamente esclusivo l'accesso a risorse condivise. Il vantaggio nell'utilizzo del monitor deriva dal fatto che non si deve codificare esplicitamente alcun meccanismo per realizzare la mutua esclusione, giacché il monitor permette che un solo processo sia attivo al suo interno, imponendo la sequenzalizzazione delle azioni, che può essere limitante, ma serve a garantire consistenza dello stato interno del monitor.

Analizzando le astrazioni disponibili su CArtAgO, i concetti di artefatto, "Observable Properties" e di "Operations" sono stati utili per l'integrazione con la Game Engine. L'immagine 2.4 contiene un esempio di interazione tra agente ed artefatto dove viene fatto uso dei concetti precedentemente elencati.

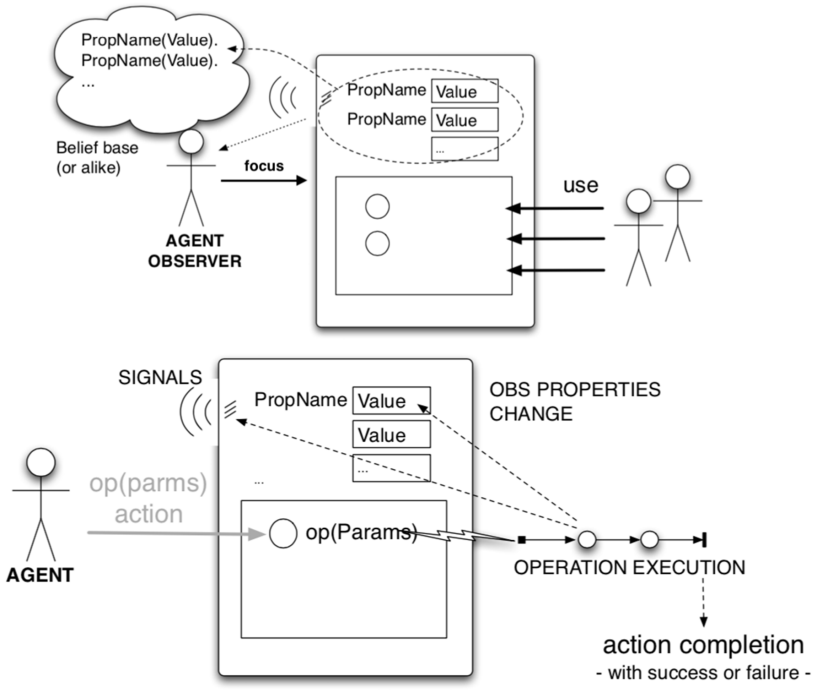

Figura 2.4: Interazione tra agente ed artefatto

L'artefatto ha contribuito a definire la principale modalità di integrazione tra JaCamo e Unity, dato che le sue finalità sono collegabili alle finalità di utilizzo del GameObject. Entrambi devono rappresentare una porzione di ambiente, risultare uno strumento a disposizione dell'agente per effettuare operazioni/azioni sull'ambiente e "notificare" l'agente in caso di modifiche sulle informazioni su essi contenute.

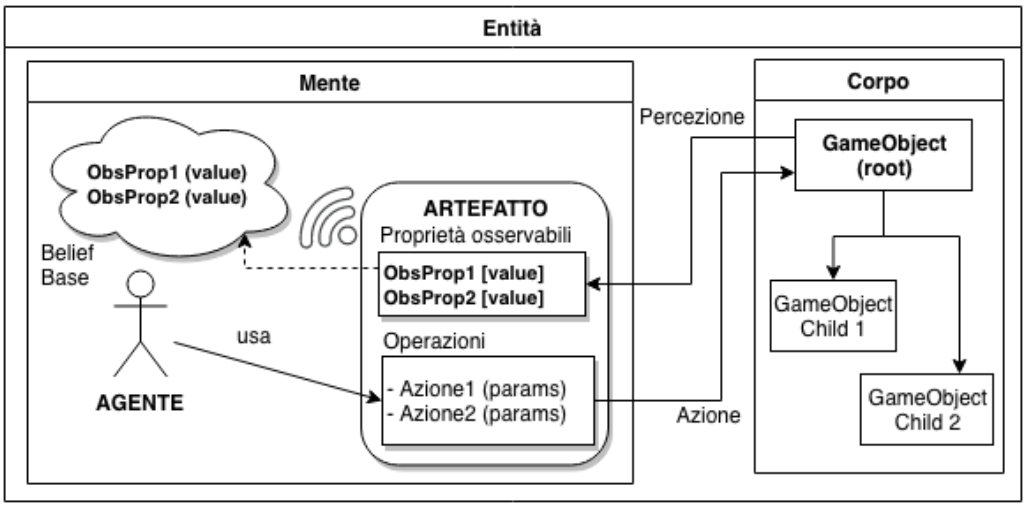

Figura 2.5: Nuova struttura entità

La definizione di queste associazioni tra Jason, CArtAgO e Unity ha portato ad una prima struttura delle componenti interne ad una generica entità, rappresentata nella figura 2.5. É da notare la composizione del corpo: difatti, è stata lasciata la possibilità di strutturare il corpo utilizzando più GameObject con l'unico vincolo che solo la radice sia in grado di comunicare con la mente.

Per questo motivo è stato deciso di effettuare un'associazione 1:1 tra GameObject ed artefatto permettendo quindi al GameObject di inviare le proprie percezioni, ricevute dell'ambiente Unity sotto forma di eventi. Ciò è stato possibile utilizzando le proprietà osservabili dell'artefatto ed all'agente di effettuare azioni sul corpo attraverso le operazioni disponibili nell'artefatto.

2.3 Architettura di sistema

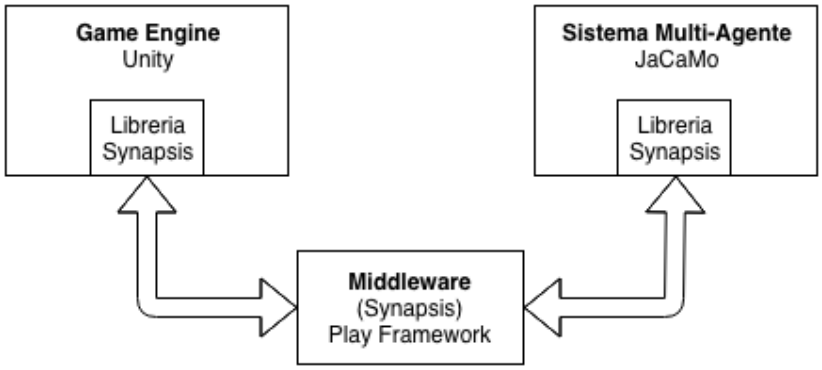

Figura 2.6: Architettura ad alto livello

L'architettura di sistema (Figura 2.6) introduce un nuovo componente, definito Synapsis e realizzato con il framework Play, separato e autonomo rispetto alle differenti tecnologie utilizzate su MAS e GE. L'obiettivo di questo middleware è mettere in collegamento le parti di entità presenti nei due sistemi e, quindi di trasportare le azioni da mente a corpo e le percezione da corpo a mente.

Per collegare MAS e GE al middleware, sono state realizzate due librerie che contengono funzionalità di collegamento e comunicazione con Synapsis. Le librerie rispettano astrazioni e modelli computazionali di entrambi i sistemi (MAS e GE) ed utilizzano la terminologia precedentemente definita.

Aggiungendo la struttura di una generica entità, definita nella sezione 2.1.6, all'architettura precedente è possibile notare come l'entità risulti suddivisa tra i due sistemi (MAS e GE) ed attraverso il collegamento al middleware, venga reso possibile lo scambio di informazioni (percezioni, azioni) pur avendo mente e corpo computazionalmente separati.

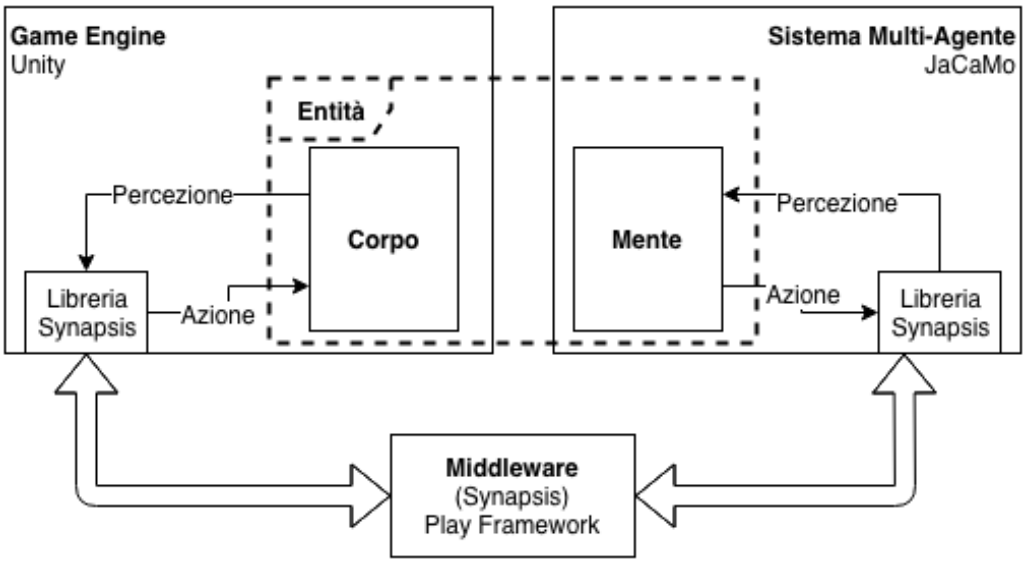

Figura 2.7: Divisione di un'entità nel sistema

L'aggiunta del middleware come layer di collegamento, rispetto alla soluzione proposta nei lavori visti nella sezione 1.2.1, favorisce il completo utilizzo, da parte degli sviluppatori, delle astrazioni e delle funzionalità presenti nelle Game Engine e nei MAS.

2.3.1 Confronto con la precedente soluzione

Nella sezione 1.2.2 è stata descritta l'architettura del precedente lavoro [8] con lo stesso obiettivo del sistema realizzato in questo elaborato. Le due architetture presentano delle similitudini:

- Utilizzo del concetto di "mente" e "corpo" per identificare le parti dell'entità nei rispettivi sistemi (MAS e GE);
- Utilizzo dell'artefatto come strumento di comunicazione dell'agente verso il proprio GameObject (corpo);
- Definizione dell'entità "mente" attraverso il collegamento tra agente ed artefatto specifico;
- Protocollo di comunicazione tra mente e corpo basato su messaggi strutturalmente simili (sezione 3.4.1).

Le principali differenze riguardano i seguenti aspetti:

- Struttura del middleware: Nel precedente lavoro, il middleware è stato separato in due parti inglobate nella Game Engine e nel Sistema Multi-Agente. Nell'architettura proposta il middleware diventa un unico componente separato che mette in comunicazione i due sistemi;

- Struttura della comunicazione: Nella soluzione proposta ogni parte di entità (mente e corpo) possiede il proprio collegamento al middleware al posto di utilizzare un unico componente (smistatore), presente in entrambi i sistemi (MAS e GE), per interfacciarsi alla controparte. Difatto nella precedente soluzione esiste un solo canale di comunicazione invece di diversi canali nella architettura proposta (sezione 3.9).

2.4 Struttura middleware

Synapsis è stato realizzato utilizzando il framework Play, illustrato nella sezione 1.3.2, le cui peculiarità sono la modularità e la distribuzione, raggiunte grazie all'utilizzo del sistema ad attori ed il modello computazionale associato: Event-driven/Message-driven.

Un sistema asincrono basato sui messaggi può utilizzare in modo più efficiente le risorse di un sistema poiché consuma risorse, come i thread, solo quando è effettivamente necessario. I messaggi possono essere recapitati anche a macchine remote (trasparenza della posizione), man mano che i messaggi vengono messi in coda e recapitati all'attore [9].

Il sistema ad attori ha portato alla definizione di un'entità "copia" all'interno del middleware, strutturata allo stesso modo dell'entità "esterna" presente in parte su GE ed in parte su MAS.

Questa soluzione ha semplificato concettualmente la gestione delle entità esterne da parte del middleware. Difatti, realizzando rispettivamente un attore che identifica il corpo ed un attore che rappresenta la mente è venuta meno la realizzazione di una componente logica per lo smistamento dei messaggi ricevuti dall'esterno. Ad esempio, quando un attore "mente" riceve un messaggio dall'esterno è consapevole che tale messaggio è stato generato ed inviato dall'entità "mente" e, di conseguenza, è chiaro che il

destinatario è l'attore "corpo", che a sua volta invia il messaggio all'entità "corpo" esterna.

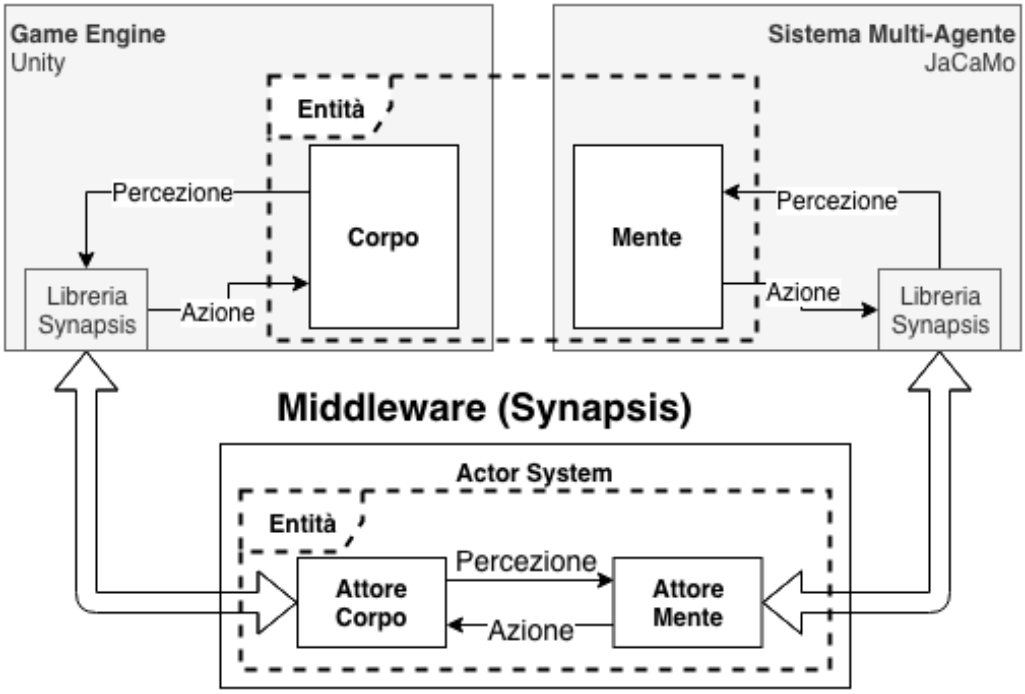

Figura 2.8: Associazione tra entità "esterna" ed "interna"

Lo schema (figura 2.8) illustra il concetto appena definito unito alla precedente architettura di sistema, dove ad ogni entità esistente nei sistemi MAS e GE viene associata una coppia di attori "mente" e "corpo" all'interno del middleware.

Capitolo 3

Synapsis: Middleware

In questo capitolo vengono descritti, in maniera più approfondita, gli elementi che compongono il sistema, quali il middleware, le librerie sviluppate per mettere collegare MAS e GE con il middleware, la tecnologia utilizzata per scambiare informazioni tra le diverse parti del sistema e la struttura dei messaggi inviati.

3.1 Middleware

In questa sezione viene definita la struttura degli attori presenti nel middleware, la modalità di associazione con l'entità esterna (mente e corpo) ed il collegamento interno tra attori.

3.1.1 Attori

Come illustrato nella sezione 2.4 la presenza di Akka all'interno del framework Play ha portato all'utilizzo degli attori come rappresentazione di una parte dell'entità (mente/corpo) all'interno del middleware.

In Synapsis è definito un attore *"BaseActor"* come attore generico in grado di ricevere messaggio dall'esterno, comunicare con la controparte, sempre attore, e di inviare messaggi alla propria parte di entità collegata.

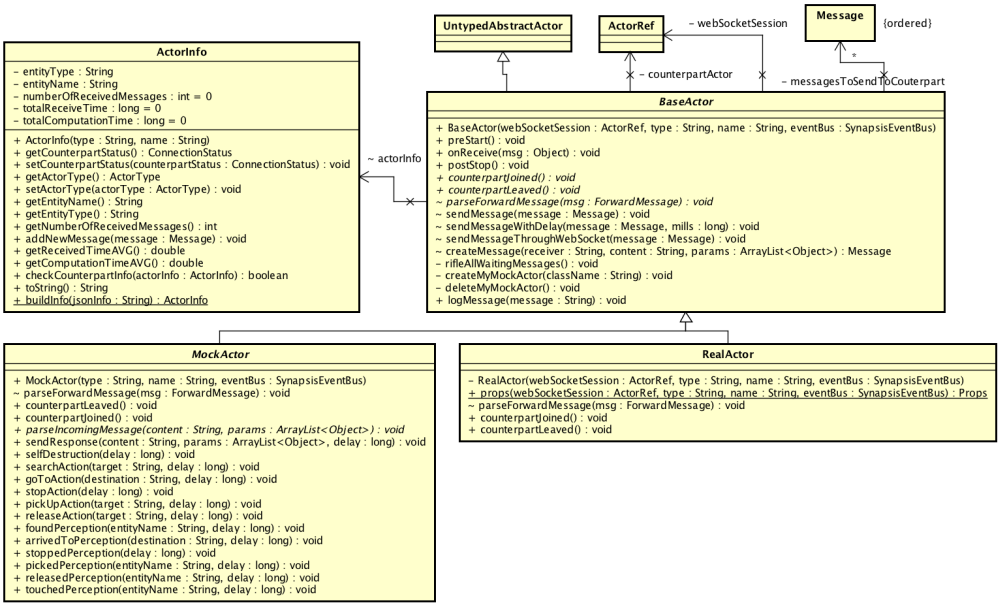

Figura 3.1: Diagramma della classi degli attori presenti in Synapsis

Nella figura 3.1 è presente il diagramma delle classi focalizzato sulla struttura degli attori all'interno di Synapsis:

- **BaseActor**: classe che estende *UntypedAbstractActor* e contiene i riferimenti all'attore controparte, all'entità collegata e alle funzionalità base necessarie ad un attore per svolgere il proprio compito;

- **ActorInfo**: classe che contiene le informazioni necessarie ad identificare un attore come nome e tipologia (mente o corpo);

- **RealActor**: classe che estende *BaseActor* e che viene istanziata quando un'entità esterna si collega al middleware;

- **MockActor**: classe che estende *BaseActor* utile per il rapid prototyping dato che permette di simulare un attore, e quindi una parte dell'entità senza che quest'ultima sia realmente collegata al middleware. Per approfondimenti consultare l'appendice A.2.

3.1.2 Collegamento tra attore e entità

Il framework Play offre diverse modalità per rendere raggiungibile l'applicazione dall'esterno come, ad esempio, richieste HTTP sincrone, asincrone e WebSocket (WS). Queste ultime sono risultate le più adatte per effettuare il collegamento tra attore ed entità, data la possibilità di instaurare una connessione duratura e bidirezionale.

Stabilendo un collegamento WS, Play si occupa di inglobare il canale in un attore, rendendo possibile il suo utilizzo all'interno del sistema ad attori e mettendo a disposizione dello sviluppatore il riferimento all'attore appena creato.

In Synapsis, questo riferimento viene passato all'effettivo attore che si occuperà di gestire i messaggi inviati dall'entità. Lo stesso procedimento viene eseguito per ogni connessione aperta, quindi, il rapporto tra numero di connessioni WS e attori "mente/corpo" è sempre di 1:1. In caso di chiusura del canale WS, Play rimuove automaticamente l'attore creato in precedenza dal sistema.

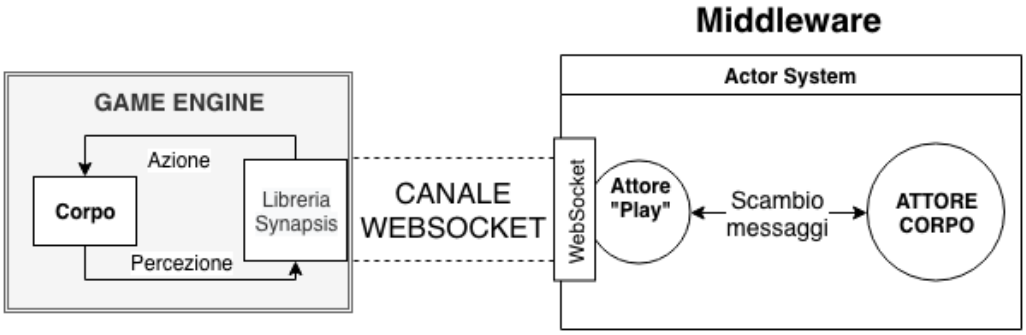

Figura 3.2: Esempio di collegamento tra attore e entità "corpo"

3.1.3 Indirizzo di collegamento al middleware

L'utilizzo di WebSocket, come protocollo di comunicazione tra i componenti del sistema, ha necessariamente portato a definire quale tra questi svolgerà la funzione di WebSocket Server. Il framework Play supportando nativamente questo protocollo è anche configurabile come server.

Lo standard RFC6455 stabilisce la struttura del URI che il WebSocket client utilizzerà per collegarsi alla risorsa/servizio messo a disposizione dal server. Per Synapsis è stato definito questo URI:

- ws://localhost:9000/**synapsiservice**/*type*/*name*

L'indirizzo specifica l'utilizzo di una connessione non criptata *"ws"* verso l'istanza locale dell'applicativo Play *"localhost:9000"* e l'utilizzo del servizio *"synapsiservice"* con l'aggiunta di parametri utili ad identificare l'entità che si collegherà a Synapsis.

In Play è presente un router HTTP, componente configurabile con il compito di tradurre ogni richiesta HTTP in un'azione dell'applicativo, dove configurare le rotte che l'applicazione rende utilizzabili dai client. Per il pattern MVC ogni richiesta HTTP viene tramutata in evento che invoca un determinato metodo. Ogni rotta è formata dalle seguenti componenti:

- Il metodo HTTP (e.g. GET, POST, ...).
- l'indirizzo della richiesta (e.g. /clienti/1542, /foto/lista), incluse eventuali Query

```
# Routes
# This file defines all application routes (Higher priority routes first)

# Controller utilizzato per la home page
GET   /   controllers.Application.index
# Controller utilizzato per il collegamento WebSocket di entità
GET   /synapsiservice/:type/:name
    controllers.Application.synapsisService(type: String, name: String)

# Map static resources from the /public folder to the /assets URL path
GET   /assets/*file controllers.Assets.versioned(path="/public", file: Asset)
```

Listing 3.1: File *routes*, configurazione rotte su Play

Il listato 3.1 mostra il file utilizzato per definire la struttura dell'indirizzo per collegarsi al middleware e come vengono utilizzati i parametri passati in fase di instaurazione del collegamento WebSocket. Ogni entità corpo o mente per creare il collegamento WebSocket deve specificare la propria tipologia e il nome che la identifica, ad esempio se la mente dell'entità "robot" vuole collegarsi al middleware l'indirizzo che utilizzerà è il seguente:

- ws://localhost:9000/synapsiservice/**mind**/**robot**

Il passaggio di questi parametri viene utilizzato dal middleware per istanziare l'attore, che rappresenta la parte di entità all'interno del sistema ad attori.

3.1.4 Collegamento tra attori "mente" e "corpo"

All'intero del sistema ad attori non è possibile inviare messaggi senza avere il riferimento al destinatario. Questo aspetto ha rappresentato un problema per il collegamento interno tra attore "mente" e attore "corpo", dato che entrambi vengono creati nel sistema ad attori successivamente all'instaurazione di un canale WS. Infatti, non è sicuro che l'entità "corpo" e l'entità "mente" si colleghino simultaneamente a Synapsis.

L'interfaccia EventBus[1], a disposizione degli sviluppatori, permette di creare un sistema di notifiche broadcast identificate da uno specifico argomento (topic), a tutti gli attori registrati allo stesso tipo di argomento. All'interno del middleware è stata realizzata la classe *SynapsisEventBus*, implementando l'interfaccia EventBus, che viene utilizzata per gestire il primo collegamento e la disconnessione tra attori della stessa entità.

[1]presente nella libreria Akka

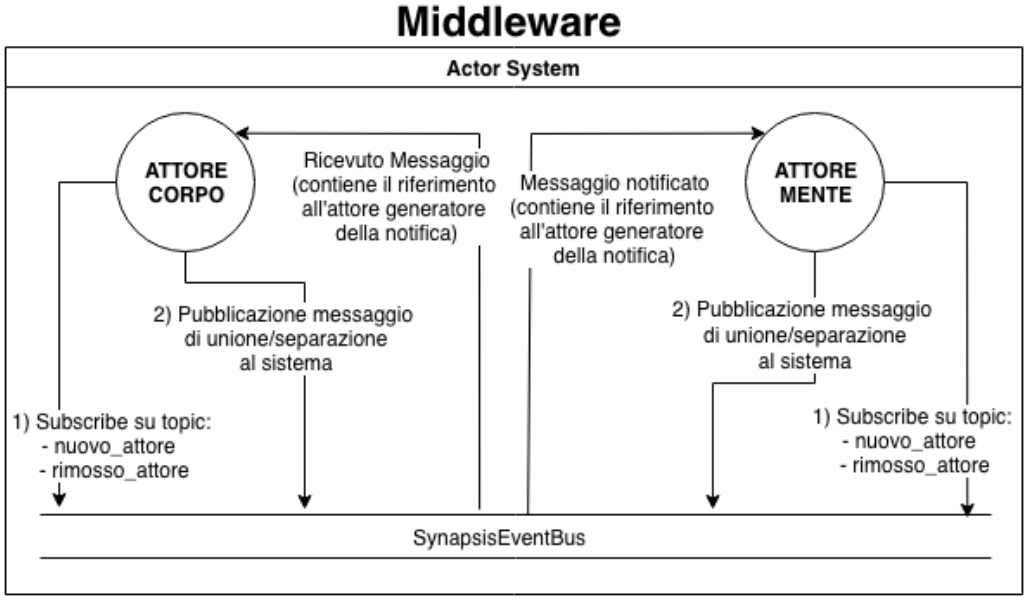

Figura 3.3: SynapsisEventBus - collegamento e disconnessione attori

Di seguito viene illustrato un esempio di funzionamento in caso di creazione dell'attore "corpo", ipotizzando che l'attore "mente" sia già presente nel sistema:

1. Il nuovo attore "corpo" si sottoscrive sul SynapsisEventBus sugli argomenti *"nuovo_attore"* e *"rimosso_attore"*,

2. L'attore "corpo" pubblica su SynapsisEventBus un messaggio con argomento *"nuovo_attore"*,

3. SynapsisEventBus notifica tutti i sottoscritti a quel determinato evento con un messaggio che conterrà anche il riferimento dell'attore che ha pubblicato il messaggio,

4. L'attore "mente" riceve il messaggio con il riferimento all'attore "corpo" e notifica nuovamente (aveva già comunicato all'EventBus la propria presenza), così da far ricevere il proprio riferimento all'attore "corpo" appena unito al sistema.

3.2 Libreria JaCaMo

In questa sezione viene descritta la liberia realizzata per JaCaMo mostrando la struttura e le componenti.

3.2.1 Struttura

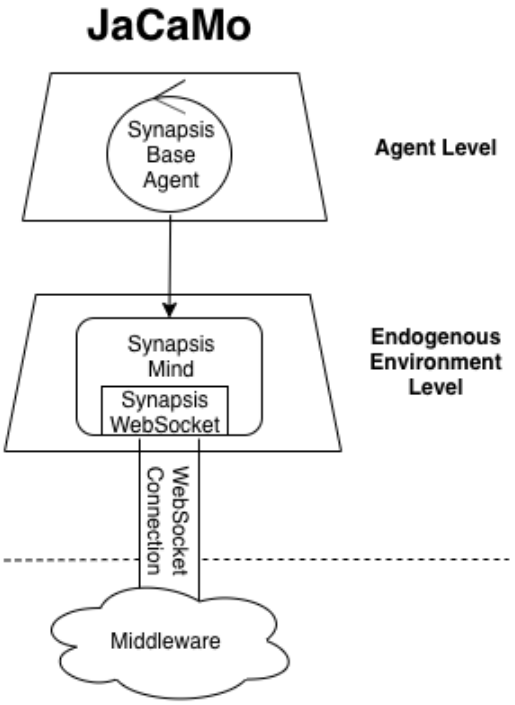

Figura 3.4: Struttura libreria per JaCaMo

L'immagine 3.4, simile alla figura 1.3 che illustrava i livelli del framework JaCaMo, mostra la composizione della libreria realizzata per mettere in comunicazione il Sistema Multi-Agente con Synapsis.

3.2.2 Artefatto Synapsis

L'artefatto è risultato il componente più adatto (sezione 2.5), lato MAS, per realizzare l'effettivo collegamento, tramite WebSocket, al middleware.

All'interno dell'artefatto *Synapsis Mind* sono presenti le funzionalità per gestire la WebSocket, per inviare azioni (in forma di messaggi strutturati) e ricevere le percezioni inviate da Synapsis.

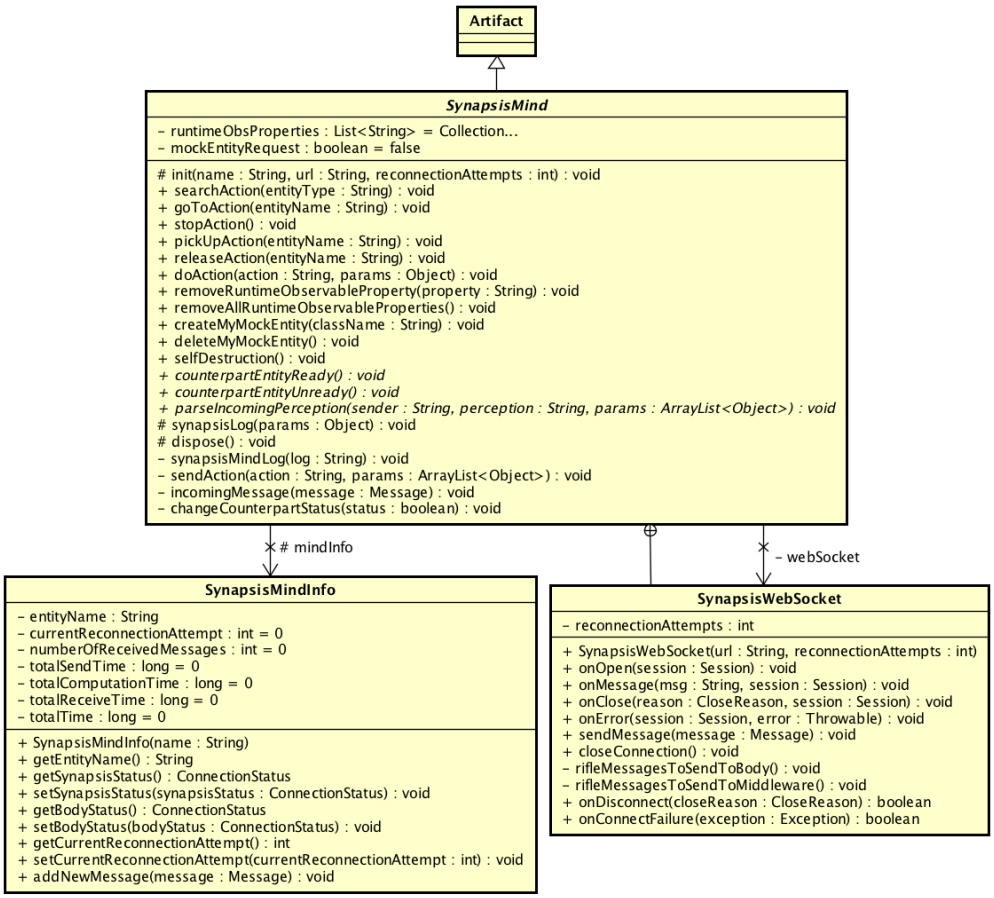

Figura 3.5: Diagramma delle classi per l'artefatto synapsis

Nella figura 3.5 è presente la struttura della classe *SynapsisMind* che estende *Artifact*, rendendola a tutti gli effetti un artefatto per il MAS. Il metodo *incomingMessage(message)* viene invocato ad ogni messaggio ricevuto dal middleware e si occupa di aggiungere il contenuto del messaggio alle proprietà osservabili dell'artefatto, notificando l'agente collegato dell'arrivo di una nuova percezione.

Sono stati realizzati diversi metodi, che abilitano la mente a comunicare con il corpo attraverso l'utilizzo della notazione *@OPERATION*, ad esempio *doAction(action,params)* che permette l'invio di una generica azione. La notazione *@OPERATION* è la modalità definita su CArtAgO per definire le operazioni (metodi) utilizzabili dall'agente. I restanti metodi, presenti nel listato 3.2, sono stati realizzati perché presentano azioni comuni a molti scenari, oltre a quello scelto come esempio (sezione 4) e delineano azioni predefinite già collegate ad effettive attività che il corpo è stato reso in grado di svolgere (vedere la sezione 3.3).

```
package synapsis;

public abstract class SynapsisMind extends Artifact {
    /**
     * Operazione per avviare la ricerca di un'entità
     * @param entityType nome anche parziale dell'entità da cercare
     */
    @OPERATION
    public void searchAction(final String entityType) {...}

    /**
     * Operazione per inviare un'azione di movimento al corpo
     * @param entityName nome entità da raggiungere
     */
    @OPERATION
    public void goToAction(final String entityName) {...}

    /**
     * Operazione per fermare il corpo
     */
    @OPERATION
    public void stopAction() {...}

    /**
     * Operazione per prelevare un'entità
     * @param entityName nome dell'entita da prelevare
     */
    @OPERATION
    public void pickUpAction(final String entityName) {...}

    /**
     * Operazione per rilasciare un'entità
     * @param entityName nome dell'entita da rilasciare
     */
    @OPERATION
    public void releaseAction(final String entityName) {...}

    /**
     * Operazione per una generica azione
     * @param action azione
```

```
41      * @param params parametri
        */
       @OPERATION
44     public void doAction(final String action, final Object... params) {...}

}
```

Listing 3.2: Operazioni (azioni) disponibili all'agente

All'interno di *SynapsisMind* è presente la classe *SynapsisWebSocket* con l'obiettivo di gestire tutte le funzionalità associate al protocollo WebSocket, messe a disposizione dalla liberia Tyrus [23]. Computazionalmente l'utilizzo di una WebSocket genera un processo separato rispetto al flusso computazionale di CArtAgO, per questo motivo è stato fatto uso delle API, *beginExternalSession* e *endExternalSession*, presenti nell'artefatto che permettono di realizzare metodi utilizzabili da processi esterni.

Per collegarsi al middleware è necessario definire a quale indirizzo è attivo Synapsis. É l'agente che riceve questa informazione durante il processo di inizializzazione dell'artefatto. Per approfondimenti consultare l'appendice.

La classe *SynapsisMindInfo* contiene le informazioni necessarie ad identificare l'entità rappresentata, lo stato del collegamento al middleware e lo stato di collegamento con l'entità "corpo" sulla Game Engine.

3.2.3 Agente Synapsis

L'agente *Synapsis Base Agent* contiene le funzionalità (beliefs e plans) basilari per la realizzazione di un agente predisposto a Synapsis.

```
// Synapsis base agent --> Agente che deve essere incluso per creare ed
    utilizzare l'artefatto SynapsisMind
2
// Initial beliefs and rules
synapsis_base_name("synapsis_"). //Nome base utilizzato per identificare
    tutti gli artefatti SynapsisMind
5
// Plans

8   // Plan da sovrascrivere in ogni agente per sapere lo stato di collegamento
    del proprio corpo
+synapsis_counterpart_status(Name,C): .my_name(Me) & .substring(Me,Name) <-
   ?my_synapsis_mind_ID(ArtId);
11    synapsisLog("Sovrascrivere belief -->
          +synapsis_counterpart_status(Name,C): .my_name(Me) &
          .substring(Me,Name)") [artifact_id(ArtId)];
```

```
    if (C == true){
       synapsisLog("Controparte collegata") [artifact_id(ArtId)];
    } else {
       synapsisLog("Controparte non collegata") [artifact_id(ArtId)];
    }.

// Plan per istanziare il proprio artefatto SinapsisMind con parametri
    custom
+!createSynapsisMind(Params): synapsis_url(Url) &
    synapsis_mind_class(Class) & reconnection_attempts(Attempts) <-
    ?synapsis_base_name(BaseName);
    .my_name(Me);
    .concat(BaseName,Me,ArtifactName);
    makeArtifact(ArtifactName,Class,[Me,Url,Attempts,Params],ArtId);
    +my_synapsis_mind_ID(ArtId);
    focus(ArtId).

-!createSynapsisMind(Params) <-
    ?synapsis_base_name(BaseName);
    .my_name(Me);
    .concat(BaseName,Me,ArtifactName);
    .print("Creazione dell'artefatto ", ArtifactName, " fallita!!", Message).

// Plan per istanziare il proprio artefatto SinapsisMind
+!createSynapsisMind: synapsis_url(Url) & synapsis_mind_class(Class) &
    reconnection_attempts(Attempts) <-
    ?synapsis_base_name(BaseName);
    .my_name(Me);
    .concat(BaseName,Me,ArtifactName);
    makeArtifact(ArtifactName,Class,[Me,Url,Attempts],ArtId);
    +my_synapsis_body_ID(ArtId);
    focus(ArtId).

-!createSynapsisMind <-
    ?synapsis_base_name(BaseName);
    .my_name(Me);
    .concat(BaseName,Me,ArtifactName);
    .print("Creazione dell'artefatto ", ArtifactName, " fallita!!", Message).
```

Listing 3.3: Agente Base Synapsis

Il listato 3.3 mostra beliefs e plans necessari all'agente per istanziare il proprio artefatto SynapsisMind. Il belief *synapsis_base_name("synapsis_"* è utilizzato per identificare, all'interno del MAS, gli artefatti istanziati utilizzando questa libreria. La parte restante del nome dell'artefatto è direttamente collegato al nome dell'agente, quindi, nel caso l'agente si chiami *robot*, il proprio artefatto sarà nominato *synapsis_robot*.

Il plan *+synapsis_counterpart_status(Name,C)* ha lo scopo di "notificare" l'agente dello stato di collegamento alla controparte "corpo", sia

in caso di connessione che di disconnessione. Attraverso *+!createSynapsisMind(Params)* e *+!createSynapsisMind* è possibile istanziare il proprio artefatto. Il primo plan permette la realizzazione di un artefatto con parametri aggiuntivi rispetto a quelli predefiniti.

Per realizzare agenti specifici è quindi necessario utilizzare *Synapsis Base Agent* come agente base dal quale prendere beliefs e plans. Nell'appendice B è presente una spiegazione più dettagliata del procedimento da seguire.

3.3 Libreria Unity

In questa sezione viene descritta la libreria realizzata per Unity mostrando la sua struttura e le sue componenti.

3.3.1 Struttura

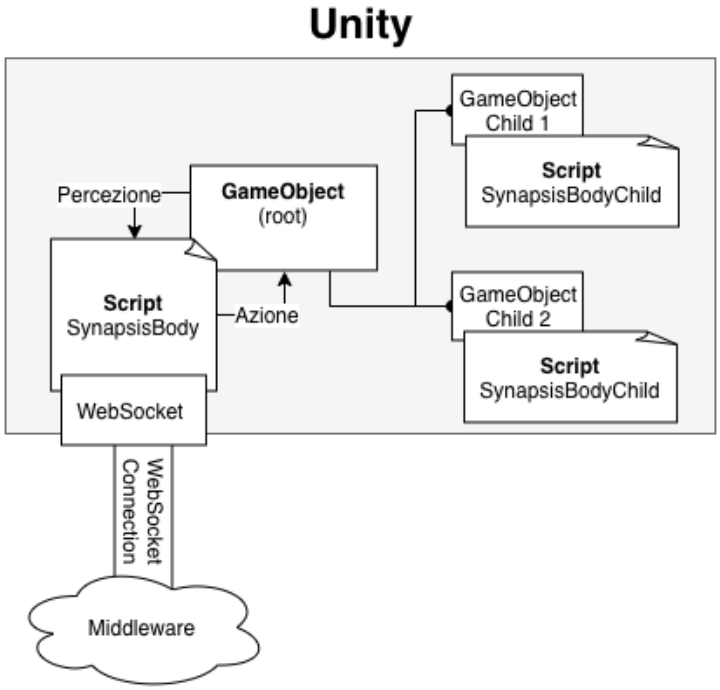

Figura 3.6: Struttura libreria per Unity

L'immagine 3.6 mostra gli elementi che compongono la libreria realizzata per Unity.

3.3.2 Script Synapsis

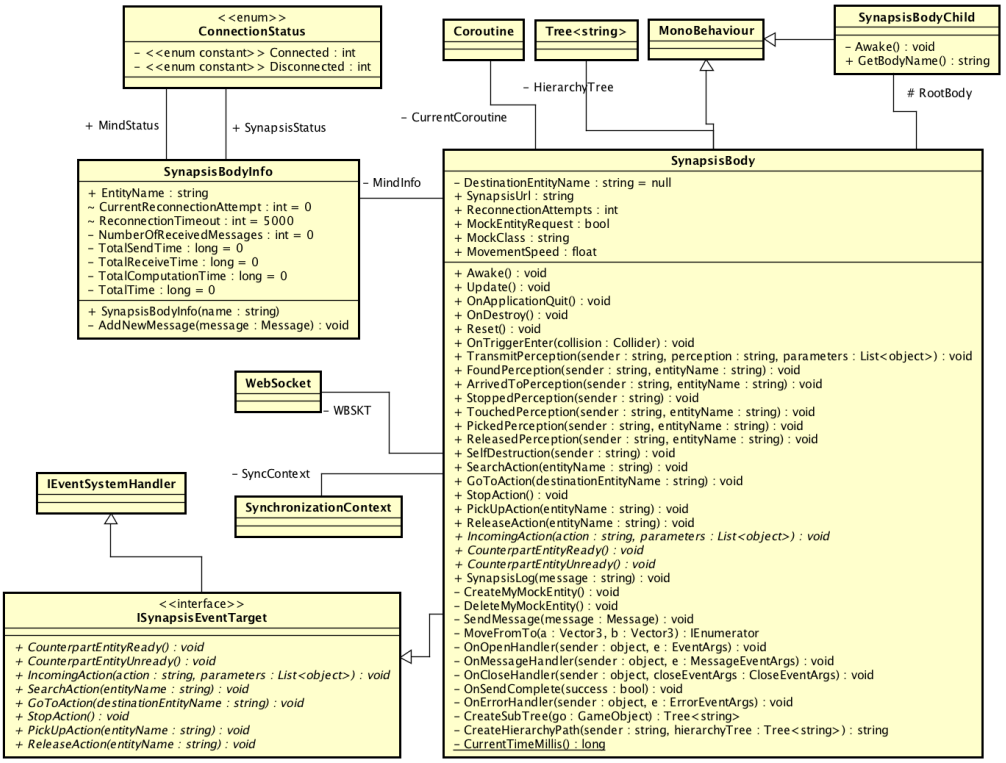

Figura 3.7: Diagramma delle classi della libreria per Unity

Il componente centrale della libreria per Unity è lo script *SynapsisBody* che estende *MonoBehaviour*, classe principale di Unity per definire comportamenti e caratteristiche dinamiche dei GameObject. Un qualsiasi GameObject che utilizza questo script è definibile come entità di tipo "corpo".

Ad ogni GameObject a cui viene collegato uno script che estende SynapsisBody vengono richieste le componenti Collider [29] e Rigibody [32]

utilizzate per rendere il GameObject sensibile alle collisioni, informando automaticamente la mente tramite percezione.

3.3.3 Funzionalità disponibili

I seguenti metodi, presenti nello script *SynapsisBody* sono a disposizione dello sviluppatore per inviare percezioni alla controparte **mind**.

- TransmitPerception(sender, perception, params) –> invia alla controparte una percezione generica con parametri collegati;
- FoundPerception(sender, entityName) –> invia alla controparte la percerzione "found" e come parametro il nome dell'entità trovata. Viene automaticamente inviata in risposta ad un'azione "search" (precedentemente inviata dalla mente)
- ArrivedToPerception(sender, entityName) –> invia alla controparte la percezione "arrived_to" e come parametro il nome dell'entità a cui si è arrivati. Viene automaticamente inviata in riposta ad un'azione "go_to" (precedentemente inviata dalla mente)
- StoppedPerception(sender) –> invia alla controparte la percezione "stopped". Viene automaticamente inviata in riposta ad un'azione "stop" (precedentemente inviata dalla mente)
- PickedPerception(sender, entityName) –> invia alla controparte la percezione "picked" e come parametro il nome dell'entità prelevata. Viene automaticamente inviata in riposta ad un'azione "pick_up" (precedentemente inviata dalla mente)
- ReleasedPerception(sender, entityName) –> invia alla controparte la percezione "released" e come parametro il nome dell'entità rilasciata. Viene automaticamente inviata in riposta ad un'azione "release" (precedentemente inviata dalla controparte mente)
- TouchedPerception(sender, entityName) –> invia alla controparte la percezione "touched" e come parametro il nome dell'entità toccata. Viene automaticamente inviata se il GameObject è dotato delle componenti "Rigibody" e "Collider".

Il metodo *TransmitPerception* rappresenta la funzionalità più modulabile per inviare una generica percezione alla mente, mentre i restanti metodi sono stati definiti perchè rappresentano azioni comuni a molti scenari simili al caso di studio preso in esempio (Sezione 4).

3.3.4 Gerarchia di oggetti complessi

Sviluppando progetti su Unity, si fa largo uso di gerarchie [31] per realizzare oggetti complessi. Per evitare l'utilizzo di un collegamento WebSocket per ogni componente dell'oggetto complesso si è deciso di limitare concettualmente alla testa (root) dell'oggetto il collegamento alla mente.

É stato realizzato uno script, estendibile, che permette anche alle sottoparti (child) di poter inviare percezioni alla mente, mantenendo l'informazione gerarchica valida e inviata assieme al messaggio.

```
public class SynapsisBodyChild : MonoBehaviour
{
   protected SynapsisBody RootBody;

   private void Awake()
   {
      // Metodo per avere l'istanza dello script che sta alla testa (root)
          del GameObject complesso
      RootBody = GetComponentInParent<SynapsisBody>();
   }

   public virtual string GetMindName()
   {
      return RootBody.name;
   }
}
```

Listing 3.4: SynapsisBodyChild script

Per mantenere la struttura gerarchica valida, viene generata una struttura ad albero dove la radice (root) conterrà il nome del GameObject collegato allo script e, automaticamente, viene popolato il sottoalbero seguendo la struttura gerarchica di Unity.

Ad ogni messaggio inviato alla controparte (mind) viene generato il path del GameObject che richiedere l'invio della percezione mappata in una stringa simile alla seguente: *"nomeGORadice.nomeGOFiglio.nomeGOSottofiglio"*

```
/// <summary>
```

```
/// Metodo per inviare le percezioni fisiche/visive del VirtualBody alla
    VirtualMind
/// </summary>
/// <param name="sender">Mittente</param>
/// <param name="perception">Breve descrizione del messaggio. ES: pronto,
    ostacolo, fermo, ...</param>
/// <param name="parameters">Lista di parametri collegati al
    contenuto</param>
public void TransmitPerception(string sender, string perception,
    List<object> parameters) {
   Message message = new Message(CreateHierarchyPath(sender, HierarchyTree),
       MindInfo.EntityName, perception, parameters);
   SendMessage(message);
}

/// <summary>
/// Genera ricorsivamente la struttura gerarchica del gameObject complesso
/// </summary>
/// <param name="go">GameObject di cui generare la struttura
    gerarchica</param>
/// <returns>L'albero della gerachia</returns>
private Tree<string> CreateSubTree(GameObject go) {
   Tree<string> subTree = new Tree<string>(go.name);
   Transform[] tList = go.GetComponentsInChildren<Transform>(); //NOTE
       prende anche se stesso quindi ho messo anche il filtro sul nome
   if (tList != null) {
      foreach (Transform t in tList) {
         if (t != null && t.gameObject != null &&
             !go.name.Equals(t.gameObject.name)) {
            subTree.AddChild(CreateSubTree(t.gameObject));
         }
      }
   }
   return subTree;
}

/// <summary>
/// Genera ricorsivamente il path della struttura gerarchica in base al
    mittente della percezione
/// </summary>
/// <param name="sender">nome del mittente</param>
/// <returns>Path nel formato "padre.figlio.sottofiglio..."</returns>
private string CreateHierarchyPath(string sender, Tree<string>
    hierarchyTree) {
   if (hierarchyTree.Value.Equals(sender)) {
      return hierarchyTree.Value;
   } else if (hierarchyTree.ChildrenCount == 0) {
      return "";
   } else {
      foreach (Tree<string> child in hierarchyTree.Children){
         string subPath = CreateHierarchyPath(sender, child);
         if (subPath != "") {
            return hierarchyTree.Value + "." + subPath;
```

```
45          }
        }
        return "";
48    }
}
```

Listing 3.5: Controllo gerarchia script

Ad esempio, nel caso di invio di una percezione da parte di un GameObject "mano", figlio di un GameObject complesso chiamato "robot", il nome finale del mittente sarà "robot.mano".

3.3.5 Inserimento WebSocket nel Event Loop di Unity

Il modello computazionale presente su Unity è di tipo Event Loop: questo significa che tutti gli elementi presenti nella scena sono vincolati ad uno specifico lifecycle (Immagine 3.8).

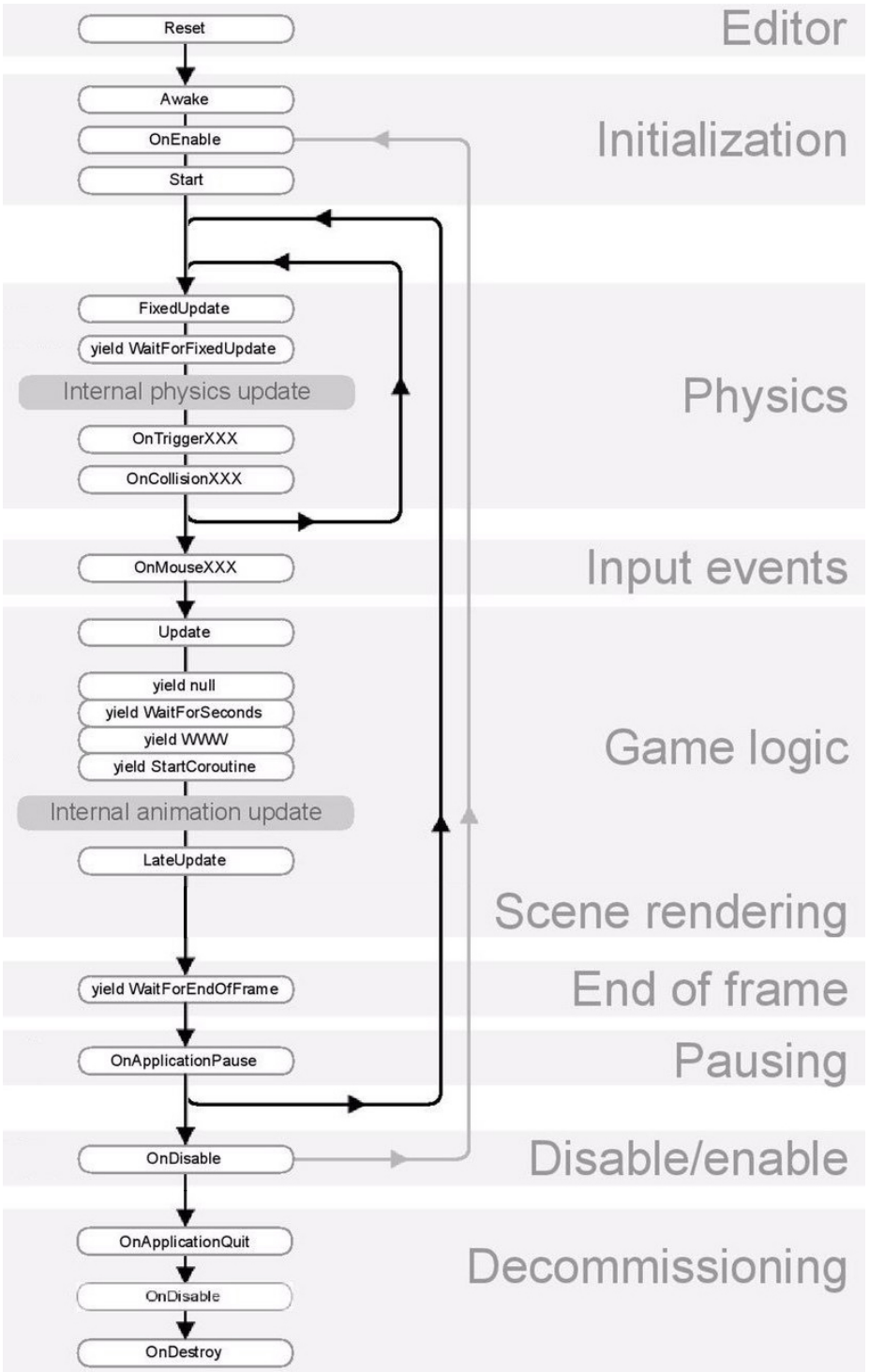

Figura 3.8: Unity Lifecycle

Per interagire internamente tra i componenti, Unity mette a disposizione delle API definite come "Event System"[30] con le quali è possibile inviare eventi agli oggetti nell'applicazione in base all'input, che si tratti di tastiera, mouse, tocco o input personalizzato.

La tecnologia di comunicazione, utilizzata per far ricevere informazioni (azioni) al corpo, ha portato a servirsi del sopra citato "Event System" di Unity. Questo perchè il canale WebSocket viene computazionalmente istanziato in un processo separato rispetto al normale lifecycle; inoltre perchè la politica presente in Unity non autorizza all'accesso diretto di thread separati a tutto ciò che è istanziato sul Main Thread[2].

```
using System.Collections.Generic;
using UnityEngine.EventSystems;

namespace SynapsisLibrary
{
   public interface ISynapsisEventTarget : IEventSystemHandler
   {
      /// <summary>
      /// Metodo da invocare quando l'entità controparte (mind) ècollegata
      /// </summary>
      void CounterpartEntityReady();

      /// <summary>
      /// Metodo da invocare quando l'entità controparte (mind) èscollegata
      /// </summary>
      void CounterpartEntityUnready();

      /// <summary>
      /// Metodo da invocare quando arriva un'azione generica
      /// </summary>
      /// <param name="action">Azione da eseguire</param>
      /// <param name="parameters">Parametri</param>
      void IncomingAction(string action, List<object> parameters);

      /// <summary>
      /// Medoto da invocare in caso di una azione di ricerca
      /// </summary>
      /// <param name="entityName">nome dell'entità da cercare</param>
      void SearchAction(string entityName);

      /// <summary>
      /// Medoto da invocare per andare verso una certa entità
      /// </summary>
      /// <param name="destinationEntityName">nome dell'entità da
          raggiungere</param>
      void GoToAction(string destinationEntityName);
```

[2]Thread principale di Unity

```
        /// <summary>
38      /// Metodo da invocare per fermare il movimento del corpo
        /// </summary>
        void StopAction();
41
        /// <summary>
        /// Metodo da invocare per raccogliere un corpo estraneo
44      /// </summary>
        /// <param name="entityName">nome dell'entità da raccogliere</param>
        void PickUpAction(string entityName);
47
        /// <summary>
        /// Metodo da invocare per rilasciare un corpo estraneo
50      /// </summary>
        /// <param name="entityName">nome dell'entità da rilasciare</param>
        void ReleaseAction(string entityName);
53  }
}
```

Listing 3.6: ISynapsisEventTarget

Per questo motivo è stato fatto uso dell'interfaccia *IEventSystemHandler*, disponibile nelle API di Unity, per realizzare l'interfaccia *ISynapsisEventTarget* attraverso la quale è possibile definire quali metodi sono utilizzabili anche da thread secondari.

Questa interfaccia viene implementata dallo script *SynapsisBody*, così, nel momento in cui il middleware invia un messaggio, il processo contenente il canale WebSocket è in grado di invocare il relativo metodo (azione).

3.4 Comunicazione

É necessario precisare che nel MAS, l'interpretazione dell'ambiente da parte di un agente si basa su concetti semantici che formano un'astrazione del mondo fisico in virtuale. Le rappresentazioni di questi stessi concetti nella GE spesso hanno un livello di astrazione diverso rispetto a ciò che è più adatto per gli agenti. Ad esempio, il concetto di "persona seduta su una sedia", associabile ad un belief del tipo *"seduta(persona,sedia)"* di un agente, può essere rappresentato nella GE dalla posizione di un personaggio in prossimità di una sedia in combinazione con le posizioni di ciascun componente, formando una posizione seduta.

3.4.1 Protocollo messaggi

Il protocollo di comunicazione prevede un solo messaggio per ogni interazione. Bisogna tenere presente che, se un messaggio viene inviato dal lato MAS sicuramente sarà un azione; se un messaggio viene inviato dal lato GE, sarà invece una percezione.

Il messaggio è così strutturato:

- **Sender**: indica il nome dell'entità (corpo o mente) che invia il messaggio;
- **Receiver**: indica il nome dell'entità (corpo o mente) che deve ricevere il messaggio;
- **Content**: contenuto principale del messaggio, corrisponde all'identificativo dell'azione o percezione inviata, ad esempio *touched* e *search*;
- **Params**: array di parametri associati alla percezione o azione. Data la diversa tipologia di linguaggi presenti nel sistema è stato deciso di supportare solo i tipi primitivi quali stringhe, numeri e booleani;
- **TimeStats**: array ordinato di statistiche temporali.

La struttura del messaggio ha contribuito ad una facile associazione al formato JSON [14] generato utilizzando le librerie GSON [17], all'interno di Synapsis e JaCaMo, e Json.Net [20], all'interno di Unity, che permettono di serializzare e deserializzare una specifica classe a Json e viceversa.

```
{
  "Sender": "test_mind", // nome del mittente
  "Receiver": "test_body", // nome del ricevente
  "Content": "go_to", // contenuto principale del messaggio
  "Params": [// array di parametri
    1.00,
    -5.43,
    "fast"
  ],
  "TimeStats": [// array ordinato di statistiche temporali
    1563478689301,// mills all'invio
    1563478689310,// mills alla ricezione del middleware
    1563478689311,// mills all'invio dal middleware
    1563478689320// mills alla ricezione
  ]
}
```

Listing 3.7: JSON di un generico messaggio

Il listato 3.7 rappresenta un esempio di messaggio, in formato JSON, che mente e corpo si possono scambiare all'interno del sistema.

3.4.1.1 Interpretare i dati statistici

La separazione tra le diverse parti del sistema (Game Engine, Middleware e Sistema Multi-Agente) ha portato alla necessità di inserire delle statistiche temporali per venire a conoscenze dei tempi di trasferimento dei messaggi.

Per interpretare i dati statistici è necessario conoscere la loro natura: è stato utilizzato il timestamp (in millisecondi) dal 1 Gennaio 1970 UTC[3] per segnare il momento esatto di invio/ricezione del messaggio.

Dato, per esempio, il precedente array di tempistiche, è possibile calcolare i tempi in questa modalità:

1. **Trasmissione da MAS/GE a Synapsis(attore)**: differenza tra il secondo ed il primo valore, quindi: *1563478689310 - 1563478689301 = 9 mills*;
2. **Trasmissione tra attori interni a Synapsis**: differenza tra il terzo ed il secondo valore, quindi: *1563478689311 - 1563478689310 = 1 mills*;
3. **Trasmissione da Synapis a MAS/GE**: differenza tra il quarto ed il terzo valore, quindi: *1563478689320 - 1563478689311 = 9 mills*.

Il tempo totale di trasmissione del messaggio è calcolabile sommando i risultati precedentemente ottenuti, in questo caso 19 mills.

3.5 Struttura completa del sistema

Viene ora proposto il sistema con tutte le componenti definite in precedenza.

[3]link al metodo Java. Per C# è stato realizzato un metodo ad hoc che funziona allo stesso modo.

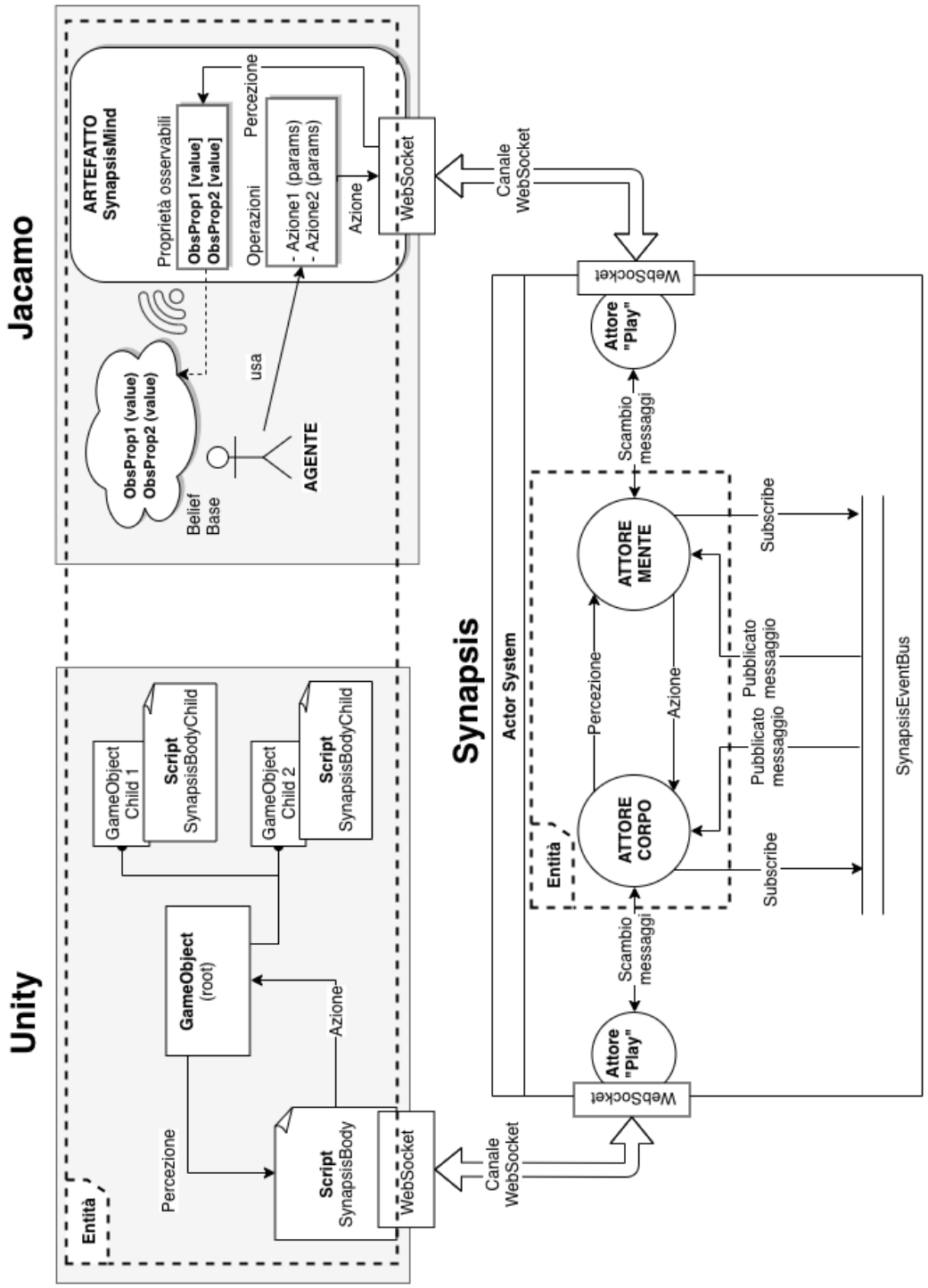

Figura 3.9: Struttura completa del sistema

Capitolo 4

Caso di studio

Giunti a questo punto, si ha tutto il necessario per iniziare a considerare l'utilizzo del sistema per alcuni problemi reali. Il caso di studio in esame riguarda lo scenario dei "Recycling Robots", descritto nel seguito.

4.1 Recycling Robots

Come si può intuire dal nome, la scena contiene dei robot, i quali hanno il compito di riciclare la spazzatura presente nell'ambiente portandola nel rispettivo bidone. Il compito generale di un robot è divisibile in un ciclo di sotto-obiettivi, ad esempio:

1. Cercare la spazzatura;
2. Andare verso la spazzatura trovata;
3. Prendere la spazzatura appena raggiunta;
4. Cercare il bidone;
5. Andare verso il bidone trovato;
6. Riciclare la spazzatura.

In questo particolare scenario è stato deciso di simulare la presenza di diversi tipi di spazzatura (plastica, vetro e carta) e, di conseguenza, sono stati creati diversi tipi di robot e bidoni.

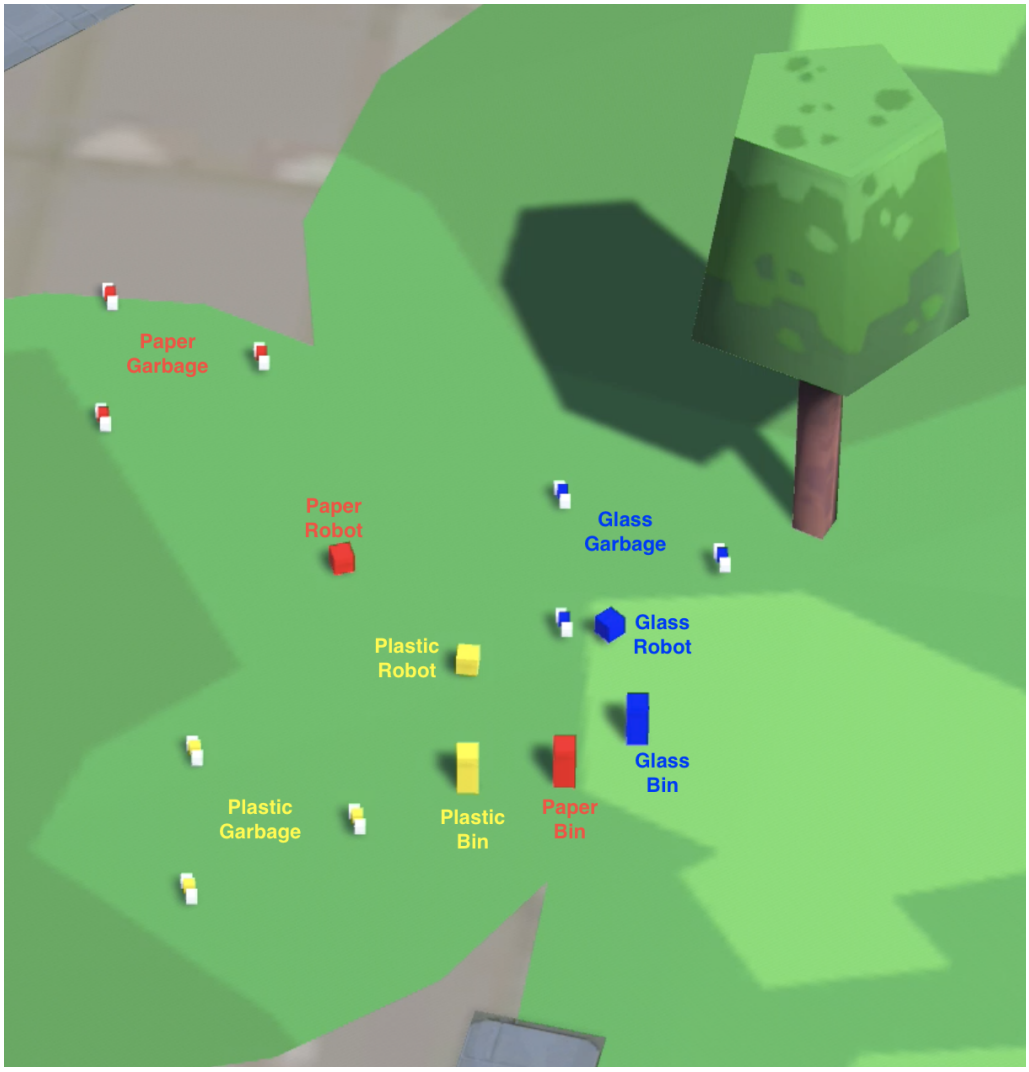

Figura 4.1: Scenario Recycling Robots

Ogni oggetto in scena è stato considerato come entità, con l'unica differenza che solo i robot sono dotati di autonomia attraverso un agente JASON ad essi collegato. La spazzatura ed i bidoni sono entità che, lato MAS, si fermano al concetto di artefatto. Per realizzare la scena è stato utilizzato WRLD [37] che fornisce mappe 3D costruite usando dati geografici di alta qualità da poter utilizzare per la creazione di visualizzazioni 3D, esecuzione

di simulazioni, o per lo sviluppo di giochi o esperienze dinamiche, basati sulla posizione geografica.

4.1.1 Robot

In questa sezione sono presenti i listati realizzati per definire il corpo e la mente dell'entità Robot.

4.1.1.1 Corpo

```
public class Robot : SynapsisBody
{
   private Material RobotMaterial;
   void Start()
   {
      RobotMaterial = GetComponent<Renderer>().material;
   }
   public override void CounterpartEntityReady(){}

   public override void CounterpartEntityUnready(){}

   public override void IncomingAction(string action, List<object>
       parameters)
   {
      switch (action)
      {
         case "robot_type":
            switch (parameters[0])
            {
               case "plastic":
                  RobotMaterial.color = Color.yellow;
                  break;
               case "paper":
                  RobotMaterial.color = Color.red;
                  break;
               case "glass":
                  RobotMaterial.color = Color.blue;
                  break;
            }
            break;
      }
   }
}
```

Listing 4.1: Robot

Come da listato 4.1, è stato realizzato uno script *Robot*, che estende *SynapsisBody*, il quale viene collegato al GameObject che rappresenta il robot

nella scena Unity. La presenza di API predefinite, all'interno di *SynapsisBody*, ha ampiamente coperto tutte le attività che il robot deve svolgere in questo particolare scenario, difatti l'unica azione da definire è stata la colorazione del robot in base alla tipologia di spazzatura che è in grado di riciclare.

4.1.1.2 Mente

```
public class RobotMind extends SynapsisMind {

   private static final String ROBOT_TYPE = "robot_type";

   protected void init(final String agentName, final String url, final int
       reconnectionAttempts, final Object[] params) {
      this.defineObsProperty(ROBOT_TYPE, params[0]); // prendo la tipologia
          di spazzatura dai parametri custom
      super.init(agentName, url, reconnectionAttempts);
   }

   @Override
   public void counterpartEntityReady() {
      if (this.hasObsProperty(ROBOT_TYPE)) {
         String type = this.getObsProperty(ROBOT_TYPE).stringValue();
         this.doAction(ROBOT_TYPE, type);
      }
   }

   @Override
   public void counterpartEntityUnready() {}

   @Override
   public void parseIncomingPerception(String sender, String perception,
       ArrayList<Object> params) {}

}
```

Listing 4.2: Artefatto del robot

Il listato 4.2 definisce l'artefatto utilizzato nel MAS per collegarsi al middleware. Estendendo la classe *SynapsisMind* è stato necessario definire una sola azione specifica che riguarda la colorazione del corpo del robot in base alla tipologia di spazzatura che lui è in grado di riciclare.

```
// Initial beliefs and rules

synapsis_url("ws:/localhost:9000/").
synapsis_mind_class("robots.RobotMind").
reconnection_attempts(5).
```

```
6
// Initial goals
!createSynapsisMind(["plastic"]).
9
// Plans
+synapsis_counterpart_status(Name, C): .my_name(Me) & .substring(Me,Name) <-
12    ?my_synapsis_mind_ID(MyArtID);
    if (C == true){
       synapsisLog("Controparte collegata -> Mettiamoci al
           lavoro!!!")[artifact_id(MyArtID)];
15     !!recycle;
    } else {
       .drop_all_intentions;
18     synapsisLog("Controparte non collegata")[artifact_id(MyArtID)];
    }.

21  +picked_up_by(C, Name) <-
    ?my_synapsis_mind_ID(MyArtID);
    if (C == true){
24     synapsisLog("Attenzione! Spazzatura prelevata da -> ", Name)
           [artifact_id(MyArtID)];
       ?robot_type(Type);
       if (.substring(Type,Name)){
27        synapsisLog("Nessun problema l'ho presa io -> ", Me)
              [artifact_id(MyArtID)];
       } else {
          ?found(Garbage);
30        !stopFocusExternalSynapsisMind(Garbage);
          removeAllRuntimeObservableProperties [artifact_id(MyArtID)];
          .drop_all_intentions;
33        stopAction [artifact_id(MyArtID)];
       }
    }.
36
+stopped <-
    .wait(500);
39  !!recycle.

+found(Name) <-
42    ?my_synapsis_mind_ID(MyArtID);
    synapsisLog("Ho visto questa entità -> ", Name)[artifact_id(MyArtID)];
    !focusExternalSynapsisMind(Name);
45  synapsisLog("Vado verso l'entità vista -> ",Name)[artifact_id(MyArtID)];
    goToAction(Name)[artifact_id(MyArtID)]. // azione per andare verso
        l'entità (in questo caso èla spazzatura)

48  +arrived_to(Name) <-
    ?my_synapsis_mind_ID(MyArtID);
    synapsisLog("Sono arrivato a questa entità -> ",
        Name)[artifact_id(MyArtID)];
51  !!recycle.

+picked(Name) <-
```

```
54    ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("Ho raccolto questa entità -> ", Name)[artifact_id(MyArtID)];
      synapsisLog("Cerco un bidone")[artifact_id(MyArtID)];
57    searchAction("bin") [artifact_id(MyArtID)].

  +released(Name) <-
60    ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("Ho rilasciato questa entità -> ",
          Name)[artifact_id(MyArtID)];
      synapsisLog("Riciclo la spazzatura -> ", Name)[artifact_id(MyArtID)];
63    recycleMe; // operazione dell'artefatto Garbage
      removeAllRuntimeObservableProperties[artifact_id(MyArtID)];
      !stopFocusExternalSynapsisMind(Name);
66    .wait(2000);
      !!recycle.

69  +!recycle: picked(Garbage) & found(Bin) & arrived_to(Bin) &
       robot_type(Type) & bin_type(Type) <-
      ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("Stessa tipologia di bidone -> ", Bin)[artifact_id(MyArtID)];
72    releaseAction(Garbage)[artifact_id(MyArtID)];
      !stopFocusExternalSynapsisMind(Bin).

75  +!recycle: picked(Garbage) & found(Name) & arrived_to(Name) &
       robot_type(Type) & bin_type(OtherType) <-
      ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("Il bidone non èdella mia stessa tipologia' ->
          ",Name)[artifact_id(MyArtID)];
78    !stopFocusExternalSynapsisMind(Name);
      removeRuntimeObservableProperty("found")[artifact_id(MyArtID)];
      removeRuntimeObservableProperty("arrived_to")[artifact_id(MyArtID)];
81    synapsisLog("Cerco un bidone")[artifact_id(MyArtID)];
      searchAction("bin") [artifact_id(MyArtID)].

84  +!recycle: found(Name) & arrived_to(Name) & robot_type(Type) &
       garbage_type(Type) <-
      ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("Stessa tipologia di spazzatura -> ",
          Name)[artifact_id(MyArtID)];
87    pickUpAction(Name)[artifact_id(MyArtID)]. // azione per prendere
          spazzatura

  +!recycle: found(Name) & arrived_to(Name) & robot_type(Type) &
       garbage_type(OtherType) <-
90    ?my_synapsis_mind_ID(MyArtID);
      synapsisLog("La spazzatura non èdella mia stessa tipologia' ->
          ",Name)[artifact_id(MyArtID)];
      !stopFocusExternalSynapsisMind(Name);
93    removeRuntimeObservableProperty("found")[artifact_id(MyArtID)];
      removeRuntimeObservableProperty("arrived_to")[artifact_id(MyArtID)];
      synapsisLog("Cerco della spazzatura")[artifact_id(MyArtID)];
96    searchAction("garbage")[artifact_id(MyArtID)].
```

```
+!recycle <-
   ?my_synapsis_mind_ID(MyArtID);
   synapsisLog("Sono libero... cerco della
       spazzatura")[artifact_id(MyArtID)];
   searchAction("garbage")[artifact_id(MyArtID)]. //azione per cercare
       spazzatura

// inclusione dell'asl che contenente belief e plan di base per synapsis. è
    possibile collegare anche un file asl all'interno di un JAR
{ include("jar:file:/Users/luca/mas-ge-jacamo/recyclingRobots/lib/
SynapsisJaCaMo.jar!/agt/synapsis/synapsis_base_agent.asl") }

{ include("$jacamoJar/templates/common-cartago.asl") }
{ include("$jacamoJar/templates/common-moise.asl") }
```

Listing 4.3: Agente del robot

Il listato 4.3 rappresenta l'agente JASON realizzato per definire l'automazione del robot. I beliefs iniziali sono utilizzati per definire il collegamento al middleware, mente il goal iniziale serve ad istanziare l'artefatto realizzato dell'immagine precedente 4.2.

Il plan *synapsis_ counterpart_ status(Name, C)* è stato utilizzato per avviare l'obiettivo di "riciclare" del robot, mentre i successivi piani (*stopped,found(Name),arrived_ to(Name)*, *picked(Name)*, *released(Name)*) sono stati definiti per reagire alle percezioni inviabili dal corpo. I piani che iniziano per *recycle* rappresentano tutte le fasi che il robot può utilizzare per riciclare la spazzatura.

Nella parte finale del listato si può vedere in che modo è stata effettuata l'importazione dell'agente base presente nella libreria per JaCaMo.

4.1.2 Bidone

In questa sezione sono presenti i listati realizzati per definire il corpo e l'artefatto dell'entità Bidone.

4.1.2.1 Corpo

```
public class Bin : SynapsisBody
{

   Material BinMaterial;

   void Start(){
```

```
      BinMaterial = GetComponent<Renderer>().material;
   }

   public override void CounterpartEntityReady(){ }

   public override void CounterpartEntityUnready(){ }

   public override void IncomingAction(string action, List<object>
       parameters)
   {
      object[] parametersArray = parameters.ToArray();

      if (action.Equals("bin_type"))
      {
         switch (parametersArray[0])
         {
            case "plastic":
               BinMaterial.color = Color.yellow;
               break;
            case "paper":
               BinMaterial.color = Color.red;
               break;
            case "glass":
               BinMaterial.color = Color.blue;
               break;
         }
      }
   }
}
```

Listing 4.4: Script per il corpo del bidone

Come da listato 4.4, è stato realizzato un script *Bin*, che estende *SynapsisBody*, il quale viene collegato al GameObject che rappresenta il bidone nella scena Unity. L'unica azione da definire è stata la colorazione del GameObject "bidone" in base alla tipologia di spazzatura da lui accettata.

4.1.2.2 Mente

```
public class BinMind extends SynapsisMind {

   private static final String BIN_TYPE = "bin_type";

   private static final String MOCK_CLASS = "BinMock";

   protected void init(final String name, final String url, final int
       reconnectionAttempts, final Object[] params) {
      // prendo la tipologia di bidone dai parametri custom
      this.defineObsProperty(BIN_TYPE, params[0]);
```

```
11      super.init(name, url, reconnectionAttempts);
        //this.createMyMockEntity(MOCK_CLASS);
    }
14
    @Override
    public void counterpartEntityReady() {
17      if (this.hasObsProperty(BIN_TYPE)) {
            String type = this.getObsProperty(BIN_TYPE).stringValue();
            this.doAction(BIN_TYPE, type);
20      }
    }

23  @Override
    public void counterpartEntityUnready() {}

26  @Override
    public void parseIncomingPerception(String sender, String perception,
        ArrayList<Object> params) {}

29  }
```

Listing 4.5: Artefatto del bidone

Il listato 4.5 definisce l'artefatto utilizzato nel MAS per collegarsi al middleware. Estendendo la classe *SynapsisMind* è stato necessario definire una sola azione specifica che riguarda la colorazione del corpo del bidone in base alla tipologia di spazzatura da lui accettata.

4.1.3 Spazzatura

In questa sezione sono presenti i listati realizzati per definire il corpo e l'artefatto dell'entità Spazzatura.

4.1.3.1 Corpo

```
1   public class Garbage : SynapsisBody
    {
        private Material GarbageMaterial;
4       private Transform InitialParent;

        void Start(){
7           //NOTE memorizzo il mio padre iniziale
            InitialParent = transform.parent;
            GarbageMaterial = GetComponent<Renderer>().material;
10      }
        public override void CounterpartEntityReady(){ }
```

```
13    public override void CounterpartEntityUnready(){ }

      public override void IncomingAction(string action, List<object>
          parameters){
16       object[] parametersArray = parameters.ToArray();

         switch (action)
19       {
            case "garbage_type":
               switch (parametersArray[0])
22             {
                  case "plastic":
                     GarbageMaterial.color = Color.yellow;
25                   break;
                  case "paper":
                     GarbageMaterial.color = Color.red;
28                   break;
                  case "glass":
                     GarbageMaterial.color = Color.blue;
31                   break;
               }
               break;
34          case "recycle_me":
               // Disattivo la spazzatura (simula il suo riciclaggio)
               gameObject.SetActive(false);
37             SynapsisLog("Mi riciclo");
               break;
         }
40    }

      private void OnTransformParentChanged()
43    {
         if (transform.parent != null &&
             !transform.parent.Equals(InitialParent)){
            TransmitPerception(name, "picked_up_by", new List<object>() { true,
                transform.parent.name }); //TODO mettere questa percezione
                nelle API?
46       } else if (transform.parent == null && InitialParent != null) {
            //NOTE serve per riposizionare la spazzatura alla parentela iniziale
            transform.parent = InitialParent;
49       }
      }
}
```

Listing 4.6: Script per il corpo della spazzatura

Come da listato 4.6, è stato realizzato uno script *Garbage*, che estende *SynapsisBody*, il quale viene collegato al GameObject che rappresenta la spazzatura nella scena Unity. Le azione da definire sono state la colorazione del GameObject "spazzatura" in base alla tipologia di spazzatura rappresentata e l'azione *recycle_me* per disattivare (riciclare) il GameObject dalla

scena.

Il metodo *OnTransformParentChanged* viene utilizzato per sapere quando la spazzatura viene raccolta da un robot cosi da inviare la percezione *picked_up_by* alla mente. Questa percezione viene utilizzata dal robot per capire quando è entrato in possesso della spazzatura che ha raggiunto.

4.1.3.2 Mente

```
public class GarbageMind extends SynapsisMind {

   private static final String PICKED_UP_BY = "picked_up_by";
   private static final String GARBAGE_TYPE = "garbage_type";

   private static final String MOCK_CLASS = "GarbageMock";

   protected void init(final String name, final String url, final int
       reconnectionAttempts, final Object... params) {
      this.defineObsProperty(PICKED_UP_BY, false, "");
      // prendo la tipologia di spazzatura dai parametri custom
      this.defineObsProperty(GARBAGE_TYPE, params[0]);

      super.init(name, url, reconnectionAttempts);
      // this.createMyMockEntity(MOCK_CLASS);
   }

   /**
    * Azione per riciclare questa spazzatura
    */
   @OPERATION
   void recycleMe() {
      this.doAction("recycle_me", new ArrayList<>());
   }

   @Override
   public void counterpartEntityReady() {
      if (this.hasObsProperty(GARBAGE_TYPE)) {
         String type = this.getObsProperty(GARBAGE_TYPE).stringValue();
         this.doAction(GARBAGE_TYPE, type);
      }
   }

   @Override
   public void counterpartEntityUnready() {}

   @Override
   public void parseIncomingPerception(String sender, String perception,
       ArrayList<Object> params) {}
}
```

Listing 4.7: Artefatto della spazzatura

Il listato 4.7 definisce l'artefatto utilizzato nel MAS per collegarsi al middleware. Estendendo la classe *SynapsisMind* è stato necessario definire due azioni specifiche: la prima riguarda la colorazione del corpo della spazzatura in base alla tipologia di spazzatura rappresentata; mentre la seconda permette di riciclare sè stessa. L'ultima azione viene utilizzata dal robot dopo aver portato la spazzatura nel bidone corretto.

4.1.4 Esempio interazione

Viene ora illustrata l'interazione tra corpo e mente dell'entità robot durante la ricerca del bidone.

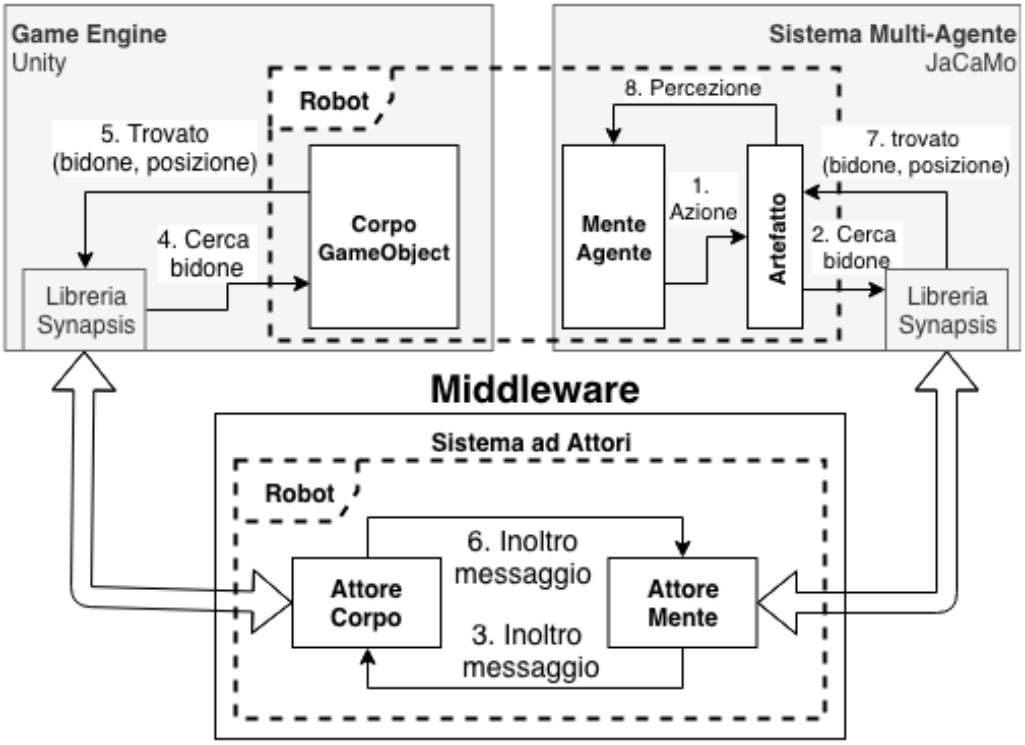

Figura 4.2: Esempio di comunicazione tra mente e corpo

L'immagine 4.2 mostra il flusso ordinato di interazioni per l'esempio appena descritto. La mente per svolgere il plan *"Cercare il bidone"* vuole inviare al proprio corpo l'azione *"Cerca bidone"*. La richiesta di svolgere l'azione inizia dall'utilizzo dell'operazione presente nell'artefatto personale dell'agente[1], in possesso del canale per comunicare con il middleware.

L'artefatto, quindi, invia il messaggio al middleware. L'attore "mente", presente nel middleware alla ricezione delle informazioni, inoltra le stesse all'attore "corpo", unico possessore del riferimento all'entità "corpo" presente su Unity in grado di inoltrare il messaggio al corpo "reale".

Alla ricezione del messaggio, l'entità corpo (GameObject) attua l'azione richiesta e risponde alla mente (Agente) inviandogli la percezione generata, ad esempio *"trovato(nomeBidone)"*.

A questo punto, la percezione viene mandata all'attore "corpo" nel middleware che, a sua volta, la inoltrerà all'attore "mente" e, di conseguenza, all'artefatto collegato. L'artefatto, nel momento in cui riceve la percezione, aggiunge quest'ultima alle sue proprietà osservabili che, automaticamente, aggiorneranno la BeliefBase dell'agente.

L'ultimo passaggio rappresenta il punto cruciale per completare il collegamento tra corpo e mente, dato che in questa maniera l'agente ha ricevuto la percezione dal proprio corpo.

4.1.4.1 Video dello scenario

É possibile visualizzare lo scenario realizzato attraverso il video presente nel repository del middleware 5.

[1] previa associazione dei due

Capitolo 5

Conclusioni e Sviluppi Futuri

Il sistema sviluppato rappresenta un'implementazione preliminare di integrazione tra MAS e GE, e sfruttando le potenzialità offerte da Unity, si ha la conferma, ancora una volta, di come le due tecnologie – quella dei Game Engine e quella dei MAS – si sposino bene insieme. L'obiettivo primario di migliorare lo stato dell'arte dell'integrazione tra GE e MAS è stato raggiunto, mettendo a disposizione del programmatore un middleware e librerie per Unity e JaCaMo con le quali è possibile realizzare scenari relativamente complessi. Rimangono, tuttavia, margini di miglioramento e ulteriori studi da compiere.

Per quanto riguarda il sistema nella sua interezza sarebbe interessante inserire SpatialTuples come mezzo abilitante la coordinazione, già utilizzato nei lavoro [1] ed utilizzare il Web Socket Secure (WSS)[1] per migliorare la sicurezza complessiva delle comunicazioni. Sempre dal punto di vista della comunicazione sarebbe da aggiungere la possibilità di inviare strutture complesse (oggetti) nei parametri dei messaggi scambiati.

Per quanto riguarda il middleware sarebbe sicuramente utile la realizzazione di una interfaccia grafica nella quale visualizzare le varie statistiche di invio e ricezione messaggi (raccolta delle tempistiche già presente), le entità presenti collegate, quali sono le entità mock, al fine di avere un vero e proprio centro di controllo del middleware.

In merito alla diffusione del sistema si potrebbero realizzare nuove librerie per aumentare l'estensione a più GE e/o MAS, cosi da non renderelo

[1] connessione criptata attraverso TLS/SSL

ad uso esclusivo di Unity e JaCamo ma, ad esempio, anche per altre GE (Unreal Engine, GameMaker, CryEngine) ed altri Multi-Agent System. A tal proposito si dovrebbbe aggiornare la libreria per JaCaMo supportandola alla sua ultima versione JaCaMo 0.8.

Un altro interessante spunto di riflessione sui possibili lavori futuri riguarda la distribuzione: tutte le parti che compongono il sistema offrono diverse modalità di eseguire gli applicativi su diversi dispositivi. A tal proposito, Unity offre nativamente il supporto al Multiplayer che potrebbe essere sfruttato per indagare più a fondo in questa direzione.

Materiale online

Synapsis

Repository con middleware e librerie disponibili all'indirizzo:
https://gitlab.com/lucapascu/mas-ge-middleware

Progetto JaCaMo

Repository con caso di studio e ambiente di test disponibili all'indirizzo:
https://gitlab.com/lucapascu/mas-ge-jacamo

Progetto Unity

Repository con caso di studio e ambiente di test disponibili all'indirizzo:
https://gitlab.com/lucapascu/mas-ge-jacamo

Appendice A

Synapsis

A.1 Setup e Avvio

Passi da seguire per configurare l'ambiente idoneo a Synapsis:

1. Installare sbt;
2. Scaricare il repository **mas-ge-middleware** utilizzando i link nella sezione 5;
3. Da terminale, navigare fino alla cartella **synapsis-middleware** interna al repository;
4. Utilizzare il comando **sbt compile** per effettuare una prima compilazione del progetto.

Passi da seguire per avviare Synapsis:

1. Da terminale, navigare fino alla cartella **synapsis-middleware** interna al repository;
2. Utilizzare il comando **sbt run** per avviare il progetto.
3. Aprire un browser ed andare all'indirizzo http://localhost:9000/ per verificare l'effettivo avvio del progetto[1].

[1]La pagina principale è ancora un template senza funzionalità, serve solo a capire se il middleware è online

A.2 MockActor

Il MockActor è stato realizzato con l'obiettivo di velocizzare la fase di sviluppo, dato che permette di creare "finti" (dall'inglese "mock") attori che sostituiscono gli attori realmente collegati ad un'entità esterna.

Lo scenario ideale per l'utilizzo di questa modalità è quello di sviluppatori in grado di utilizzare solo una tecnologia, tra Unity e JaCaMo, e che attraverso la realizzazione di MockActor specifici riescano a sopperire alla necessità di uno sviluppo simultaneo.

A.2.1 Come utilizzare il MockActor

I passi da seguire per realizzare "finti" attori sono i seguenti:

1. Creare una classe java che estenda MockActor,
2. Posizionare la classe dentro il package **actor.mock** presente nel middleware,
3. Implementare i metodi astratti come mostrato nel listato A.1.

```
public class TestMock extends MockActor {

   private TestMock(final String type, String name, SynapsisEventBus
       eventBus) {
      super(type, name, eventBus);
   }

   public static Props props(String type, String name, SynapsisEventBus
       eventBus) {
      return Props.create(TestMock.class, () -> new TestMock(type, name,
          eventBus));
   }

   /**
    * Metodo invocato per ogni messaggio ricevuto dal MockActor
    * @param content contenuto principale del messaggio
    * @param params parametri del messaggio
    */
   @Override
   public void parseIncomingMessage(String content, ArrayList<Object>
       params) {
   }

   /**
```

```
     * Metodo invocato alla connessione della controparte
22    */
    @Override
    public void counterpartJoined() {
25      super.counterpartJoined();
    }

28  /**
     * Metodo invocato alla disconnessione della controparte
     */
31  @Override
    public void counterpartLeaved() {
      super.counterpartLeaved();
34  }
}
```

Listing A.1: Esempio di attore che estende la classe *MockActor*

Il metodo *parseIncomingMessage*, invocato ad ogni messaggio ricevuto dall'attore, permette allo sviluppatore di decidere come gestire tali messaggi che possono essere azioni o percezioni inviate dall'attore controparte e, quindi dall'entità esterna collegata al middleware.

A disposizione dello sviluppatore sono presenti alcuni metodi, già implementati all'interno della classe *MockActor*, per interagire con l'attore controparte (mente/corpo) e quindi con l'entità ad esso collegata. Il listato A.2 contiene tutti i metodi utilizzabili con i relativi commenti.

```
1   /**
     * Metodo per inviare un messaggio(risposta) generico alla controparte
     * @param content contenuto principale del messaggio
4    * @param params parametri del messaggio
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
7   public void sendResponse(String content, ArrayList<Object> params, long
        delay) {...}

    /**
10   * Metodo per inviare un messaggio di auto-distruzione alla controparte
     * @param delay tempo di attesa (ms) prima dell'invio della risposta
     */
13  public void selfDestruction(long delay) {...}

    /**
16   * Metodo per inviare una azione di ricerca alla controparte
     * @param target nome (anche parziale) dell'entità da cercare
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
19   */
    public void searchAction(String target, long delay) {...}
```

```
22  /**
     * Metodo per inviare una azione di movimento alla controparte
     * @param destination nome dell'entità da raggiungere
25   * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
    public void goToAction(String destination, long delay) {...}
28
    /**
     * Metodo per inviare una azione di stop alla controparte
31   * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
    public void stopAction(long delay) {...}
34
    /**
     * Metodo per inviare una azione di prelevamento alla controparte
37   * @param target nome dell'entità da prendere
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
40  public void pickUpAction(String target, long delay) {...}

    /**
43   * Metodo per inviare una azione di rilasio alla controparte
     * @param target nome dell'entità da rilasciare
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
46   */
    public void releaseAction(String target, long delay) {...}

49  /**
     * Metodo per inviare una percezione di ritrovamento alla controparte
     * @param entityName nome dell'entità trovata
52   * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
    public void foundPerception(String entityName, long delay) {...}
55
    /**
     * Metodo per inviare una percezione di arrivo alla controparte
58   * @param destination nome dell'entità raggiunta
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
61  public void arrivedToPerception(String destination, long delay) {...}

    /**
64   * Metodo per inviare una percezione di stop alla controparte
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
67  public void stoppedPerception(long delay) {...}

    /**
70   * Metodo per inviare una percezione di prelevamento alla controparte
     * @param entityName nome dell'entità prelevata
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
73   */
    public void pickedPerception(String entityName, long delay) {...}
```

```
76  /**
     * Metodo per inviare una percezione di rilascio alla controparte
     * @param entityName nome dell'entità rilasciata
79   * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
    public void releasedPerception(String entityName, long delay) {...}
82
    /**
     * Metodo per inviare una percezione di contatto alla controparte
85   * @param entityName nome dell'entità toccata
     * @param delay tempo di attesa (ms) prima dell'invio del messaggio
     */
88  public void touchedPerception(String entityName, long delay) {...}
```

Listing A.2: Metodi per interagire con la controparte

Il metodo *sendResponse* è la condizione per inviare una generica risposta alla controparte e, quindi, utilizzabile per inviare un messaggio al corpo o alla mente. É da notare la presenza di interazioni predefinite dal punto di vista del contenuto del messaggio, difatti tutti i metodi che finiscono per *Action* e per *Perception* inseriscono automaticamente il contenuto principale del messaggio (sezione 3.4.1). Ad esempio, nel caso di *searchAction* il contenuto sarà *"search"*. L'obiettivo di questi metodi è quello di mettere subito a disposizione dello sviluppatore un primo set di Azioni e Percezioni già associate a una logica predefinita nelle restante parte del sistema.

A.2.2 Architettura del sistema con MockActor

Questa modalità di fast prototyping, modifica la architettura del sistema (Figure A.1 e A.2) in basa a quale tipologia di *MockActor* viene utilizzato.

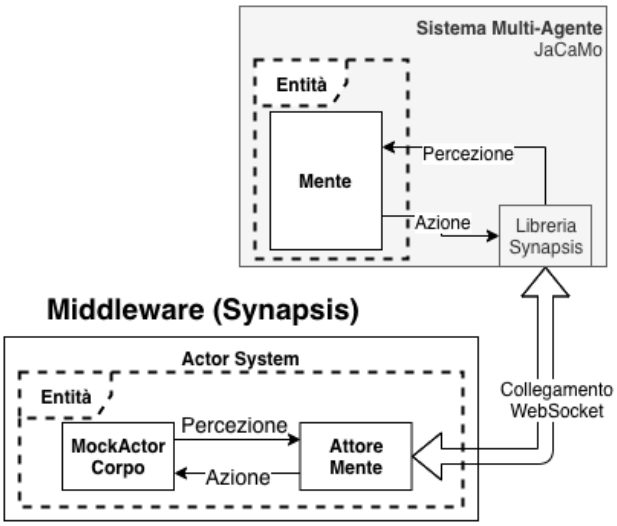

Figura A.1: Architettura con un MockActor di tipo *body*

Con l'utilizzo di un "finto" attore che rappresenta la parte di entità "corpo" l'architettura del sistema si modifica di conseguenza, visto che viene meno la parte di Game Engine (GE). Le percezioni e le risposte alle azioni ricevute vengono simulate dal *MockActor* di tipo "corpo" realizzato, che potrà essere successivamente sostituito dall'entità "corpo" in futuro realizzata sulla GE.

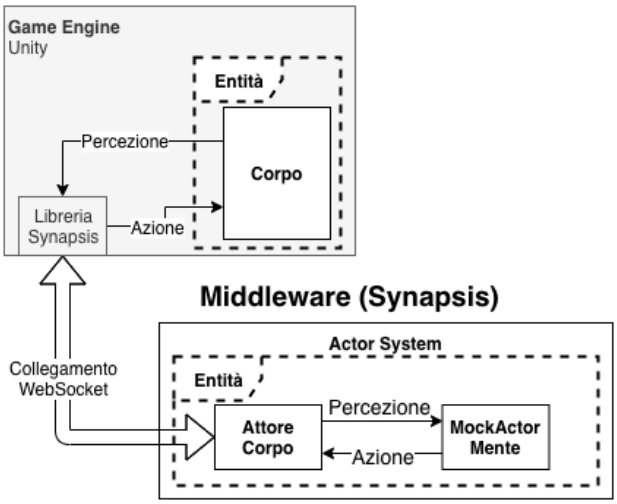

Figura A.2: Architettura con un MockActor di tipo *mind*

Con l'utilizzo di un "finto" attore che rappresenta la parte di entità "mente", l'architettura del sistema si modifica di conseguenza, visto che viene meno la parte di Sistema Multi-Agente (MAS). L'autonomia viene simulata dal *MockActor* di tipo "mente" realizzato che potrà essere successivamente sostituito dall'entità "mente" realizzata sul MAS.

A.2.3 Istanziare MockActor

A.2.3.1 Istanziare dall'interno del middleware

Lo sviluppatore ha a disposizione due metodi all'interno della classe **Application**, che permettono di istanziare uno o più "finti" attori precedentemente realizzati.

```
public class Application extends Controller {

   @Inject
   public Application(ActorSystem actorSystem, Materializer materializer) {
      ...
      this.spawnMockActor(TestMock.class,Shared.ENTITY_BODY_KEY,"prova");
      //oppure
      this.spawnMockActors(TestMock.class,Shared.ENTITY_MIND_KEY,"test",2);
   }

   /**
    * Metodo per istanziare un attore che estende MockActor
```

```
     * @param source Classe Java che rappresenta l'attore
     * @param type Tipologia di entità (mind / body)
     * @param name Nome dell'entità
     */
    private void spawnMockActor(Class<? extends MockActor> source, String
        type, String name) {
      Props props = Props.create(source, type, name, this.synapsisEventBus);
      this.actorSystem.actorOf(props);
    }

    /**
     * Metodo per istanziare N attori che estendono MockActor
     * @param source Classe Java che rappresenta l'attore
     * @param type Tipologia delle entità (mind / body)
     * @param baseName Nome base delle entità -> il nome risultato sara
        baseName1,...,baseNameN
     * @param number Numero di entità da istanziare
     */
    private void spawnMockActors(Class<? extends MockActor> source, String
        type, String baseName, int number) {
      for (int i = 1; i <= number; i++) {
        this.spawnMockActor(source, type, baseName + i);
      }
    }
}
```

Listing A.3: Metodi per istanziare un MockActor nel middleware

I metodi *spawnMockActor* e *spawnMockActors* permettono di creare, all'interno del sistema, generici attori dato che accettano classi Java che estendano la classe *MockActor*. Questi metodi, come da listato A.3, vanno utilizzati nel costruttore della classe **Application** per essere certi di istanziare gli attori all'avvio dell'applicazione.

A.2.3.2 Istanziare dall'esterno del middleware

Per istanziare MockActor runtime sono state fornite delle API nelle rispettive librerie di JaCaMo e Unity che comunicano direttamente con il middleware e permettono la creazione di questi attori attraverso l'invio di un messaggio predefinito.

Su Unity è stato realizzato un processo che utilizza l'interfaccia grafica dell'IDE (figura A.3). Collegando al GameObject uno script che estende *SynapsisBody* si ottiene il seguente risultato:

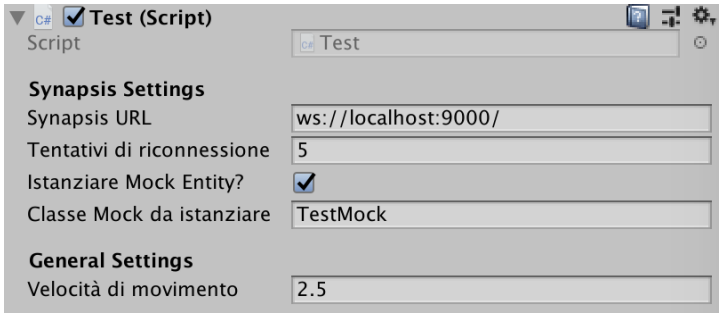

Figura A.3: Interfaccia grafica di gestione dello script

I parametri *"Istanziare Mock Entity?"* e *"Classe Mock da istanziare"*, permettono al GameObject di creare il messaggio da inviare al middleware appena stabilito il collegamento WebSocket.

Per JaCaMo sono presenti due modalità. La prima è effettuabile attraverso l'invocazione di uno specifico metodo presente nell'artefatto *SynapsisMind*.

```
package synapsis;

public abstract class SynapsisMind extends Artifact {

   /**
    *  Operazione per istanziare il proprio mock actor nel middleware
    * @params className nome della classe java dell'attore da istanziare
    */
   @OPERATION
   public void createMyMockEntity(final String className) {...}
}
```

Listing A.4: Metodo per istanziare MockActor dall'artefatto

La seconda utilizza uno specifico plan presente nell'agente *SynapsisBaseAgent* che a loro volta utilizza l'operazione (metodo) del listato precedente.

```

// Plan per creare il proprio MockActor su Synapsis
+!createMySynapsisMockEntity(MockClassName): focused(_,N,_) &
    synapsis_base_name(BaseName) & .substring(BaseName,N) <-
   ?my_synapsis_mind_ID(ArtId);
   .my_name(Me);
   createMyMockEntity(MockClassName) [artifact_id(ArtId)];
   synapsisLog("Inviata richiesta di creazione entità mock:", Me , "->
       classe:", MockClassName) [artifact_id(ArtId)].
```

Listing A.5: Piano per istanziare MockActor

In entrambe le situazioni, lo sviluppatore deve fare attenzione a scrivere correttamente il nome della classe da istanziare ed a posizionare la classe Java nel corretto package (actors.mock).

Appendice B

JaCaMo

B.1 Utilizzare la libreria

La distribuzione della libreria realizzata per JaCaMo viene effettuata attraverso la generazione di un *Jar*[1] contenente classi Java e file *"asl"*. Questa modalità non permette di includere le dipendenze esterne [17][23], che devono essere importate manualmente nel progetto.

La libreria prodotta, *SynapsisJaCaMo.jar*, è disponibile sul repository del middleware (sezione 5) e per utilizzarla è sufficiente importarla nel nuovo progetto JaCaMo.

B.2 Realizzare Artefatti Synapsis custom

Come spiegato nella sezione 3.2.2, la classe *SynapsisMind* definisce un artefatto con le funzionalità di comunicazione con il middleware. Per realizzare un artefatto custom è sufficiente creare una classe Java che estenda *SynapsisMind*.

```
public class TestSynpasisArtifact extends SynapsisMind {

  private int counter = 0;

  protected void init(final String name, final String url, final int
      reconnectionAttempts, final Object[] params) {
    super.init(name, url, reconnectionAttempts);
```

[1] **J**ava **AR**chive

```
    }

    /**
     * Esempio di operazione custo
     */
    @OPERATION
    void azionePersonalizzata() {
       this.doAction("Azione", this.counter);
       this.counter++;
    }

    /**
     * Metodo invocato al collegamento con l'entità corpo
     */
    @Override
    public void counterpartEntityReady() {...}

    /**
     * Metodo invocato alla disconnessione con l'entità corpo
     */
    @Override
    public void counterpartEntityUnready() {...}

    /**
     * Metodo invocato per ogni messaggio (percezione) ricevuto
     */
    @Override
    public void parseIncomingPerception(String sender, String perception,
        ArrayList<Object> params) {...}

}
```

Listing B.1: Artefatto Synapsis custom

Il listato B.1 contiente un esempio di artefatto custom che estende la classe *SynapsisMind*. Il metodo *init(..)* è il costruttore invocato, secondo le logiche di CArtAgO, durante la creazione di un generico artefatto. Su Synapsis, questo metodo viene utilizzato per fornire all'artefatto informazioni utili (indirizzo e tentativi di riconnessione) per effettuare il collegamento WebSocket, quindi nell'artefatto custom è necessario richiamare il metodo originario utilizzando il costrutto *super*.

Con l'estensione della classe *SynapsisMind* viene automaticamente richiesta la definizione dei metodi che permettono la gestione di informazioni in arrivo alla mente:

- counterpartEntityReady –> Invocato al collegamento con l'entità corpo;

- counterpartEntityUnready –> Invocato alla disconnessione con l'entità corpo;
- parseIncomingMessage –> Invocato ad ogni messaggio (percezione) ricevuto dal corpo;

Per definire azioni specifiche, in aggiunta alle azioni già presenti spiegate nella sezione 3.2.2, è necessario definire un nuovo metodo, con la notazione @OPERATION (per renderlo utilizzabile dall'agente), ed invocare il metodo *doAction* per inviare il messaggio al corpo. Nel listato B.1 è presente il metodo *azionePersonalizzata* come esempio di azione custom.

Per la modalità di collegamento tra artefatto ed agente è necessario utilizzare anche la successiva sezione.

B.3 Realizzare Agenti Synapsis custom

Per realizzare agenti Synapsis custom è necessario importare l'agente *synapsis_ base_ agent.asl* presente nella libreria.

```
// inclusione di un file .asl (vengono importati beliefs e plans).
{include("jar:file:percorso/.../Synapsis.jar!/agt/synapsis/synapsis_base_agent.asl")}
```

Listing B.2: Includere file .asl da Jar

Nel listato B.2 è presente il comando, messo a disposizione da JaCaMo, per importare file .asl presenti all'interno di Jar. Questa riga è da aggiungere al nuovo agente che si sta realizzando. Con l'operazione precedente si hanno a diposizione tutti i beliefs, goals e plan di *synapsis_ base_ agent*.

```
// Initial beliefs and rules

synapsis_url("ws:/localhost:9000/").
synapsis_mind_class("artifacts.TestSynapsisArtifact").
reconnection_attempts(5).

// Initial goals
!createSynapsisMind(["test",1,false]).

// Plans
+synapsis_counterpart_status(Name, C): .my_name(Me) & .substring(Me,Name) <-
   ?my_synapsis_mind_ID(MyArtID);
   if (C == true){
      synapsisLog("Controparte collegata")[artifact_id(MyArtID)];
      !!provaAzione;
```

```
16    } else {
         synapsisLog("Controparte non collegata")[artifact_id(MyArtID)];
      }.
19
   // azione personale
   +!provaAzione <-
22    azionePersonalizzata[artifact_id(MyArtID)].

   // inclusione dell'asl che contenente belief e plan di base per synapsis è
       possibile collegare anche un file asl all'interno di un JAR
25 { include("synapsisJaCaMo/synapsis_base_agent.asl") }
```

Listing B.3: Agente Synapsis custom

Il listato B.3 rappresenta un agente custom pronto all'utilizzo. I beliefs iniziali sono necessari ad effettuare il collegamento WebSocket. Da notare è il belief *synapsis_mind_class("artifacts.TestSynapsisArtifact")* utilizzato da *synapsis_base_agent* per sapere quale classe rappresenta il proprio artefatto.

Il plan *+synapsis_counterpart_status(Name, C)* viene utilizzato per conoscere lo stato di collegamento con la controparte corpo. Questo plan viene scatenato dall'aggiornamento delle proprietà osservabili dell'artefatto secondo le informazioni ricevute dal middelware. Per completare il ciclo è stato definito il plan *+!prova* per utilizzare l'operazione precedentemene definita nell'artefatto B.1.

Appendice C

Unity

C.1 Utilizzare la libreria

La distribuzione della libreria realizzata per Unity viene effettuata attraverso la generazione di un *Asset packages* ossia collezione di file e dati, realizzati da progetti Unity, compressi e memorizzati in un unico file, simile ad una cartella compressa. All'interno del package, la struttura dei file e i metadati rimangono gli stessi dell'originale così da rendere possibile la distribuzione del codice sorgente e delle dipendenze esterne.

La libreria prodotta, *Synapsis.unitypackage*, è disponibile sul repository del middleware (sezione 5) e per importarla è necessario creare un nuovo progetto Unity, da menu "Assets"->"Import Package"->"Custom Package" e selezionare la libreria.

C.2 Realizzare Script custom

Come spiegato nella sezione 3.3, la classe *SynapsisBody* è uno script che contiene le funzionalità di collegamento al middleware. Per realizzare uno script custom è sufficiente creare una nuova classe che estenda *SynapsisBody*.

```
public class Test : SynapsisBody
{
    /// <summary>
    /// Metodo invocato per ogni messaggio inviato dalla controparte (mind)
    /// </summary>
    /// <param name="action">Azione da svolgere</param>
```

```
    /// <param name="parameters">Parametri aggiuntivi</param>
    public override void IncomingAction(string action, List<object>
        parameters){...}

    /// <summary>
    /// Metodo invocato quando la controparte (mind) ècollegata
    /// </summary>
    public override void CounterpartEntityReady(){...}

    /// <summary>
    /// Metodo invocato quando la controparte (mind) èscollegata
    /// </summary>
    public override void CounterpartEntityUnready(){...}
}
```

Listing C.1: Esempio di script che estende SynapsisBody

I metodi *CounterpartEntityReady* e *CounterpartEntityUnready* vengono utilizzati per conoscere lo stato di collegamento con la controparte mente. Questi metodi vengono automaticamente invocati in base alle informazioni ricevute dal middleware.

Collegando lo script sopra elencato al GameObject compaiono, nella sezione "Inspector" di Unity", le configurazioni necessarie per definire il collegamento al middleware, nell'editor di Unity.

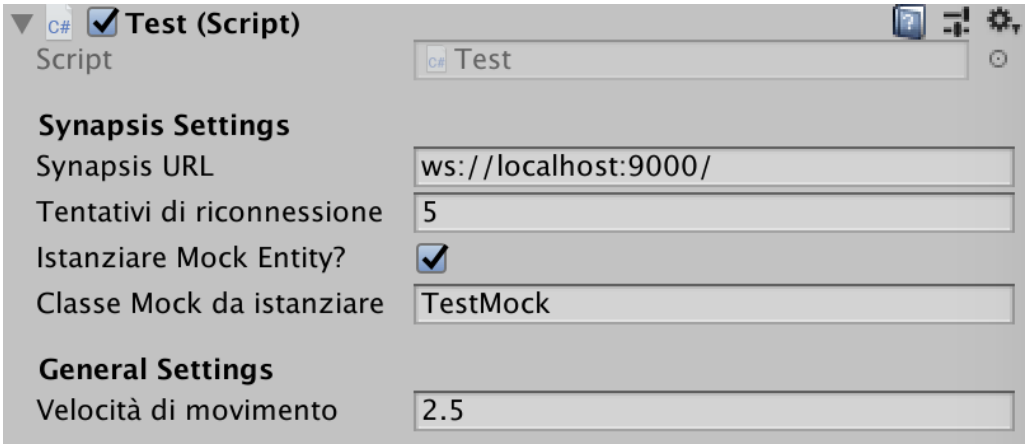

Figura C.1: Configurazioni dello Script Test che estende SynapsisBody

Le configurazioni riguardano il collegamento al middleware, la necessità di instanziare una MockEntity e la velocità di movimento dell'entità (se deve effettuare spostamenti nell'ambiente).

Ringraziamenti

Giunto alla conclusione di questo cammino, mi trovo in difficoltà a scrivere i ringraziamenti...

Il motivo è semplice, vorrei ringraziare uno per volta tutte le persone conosciute in questi anni e dirgli come hanno contribuito a farmi diventare la persona che sono ora ma questa paginetta non sarà sufficiente. Tranquilli ho una soluzione anche per voi!

Ale e Filo, siete le prime persone con le quali sia riuscito a creare una squadra di smanettoni, grazie per avermi insegnato che la taverna di casa è un ottimo posto per costruire App e robot in compagnia. #RepairCityTornerà

UniboMotorsport, il Team che mi ha completamente stravolto le ultime cinque estati (per fortuna solo quelle). Assieme a voi ho riscoperto la mia passione per il Motorsport, dalla convergenza, allo stare a sedere ore in macchina per creare un sedile, e sopratutto al vivere di redbull per reggere le fatiche di trasferte infinite. Abbiamo girato gran parte dell'Europa, pure il Brasile (che matti), ma mai mi scorderò di voi e di Collamarini, il luogo dove tutto prendeva forma. #KeepPushing

Marco e Scat, il duo del dormire in camper al contrario (ingegneri si nasce). Con voi ho passato i più bei momenti della FSAE e devo ammettere che si è creata un'amicizia speciale (a base di pappardelle al cinghiale) a cui tengo e che continuerò a coltivare con molto piacere. #QuelliDelRotax

Noemi, l'unica persona in grado di cambiarmi la giornata, dire grazie è riduttivo <3. Sei riuscita a fare la tua tesi e nello stesso tempo aiutarmi a correggere questa (migliorerò l'utilizzo degli apostrofi). Oggi siamo ad 1 anno e 250 giorni, come minimo ne voglio altrettanti #C'èQualcosaDiGrande #IlTuoPesarese

Family (Lucky ovviamente incluso), siete stati la fonte di energia di tutto questo, grazie per avermi supportato e soprattutto sopportato. #GrazieDi-Tutto #ReLucky

Mariani e Omicini, incredibilmente motivanti, mi avete dato una carica (e mano) pazzesca lungo tutto questo ultimo sprint. #ProfDelFuturo

Come detto in precedenza non riesco a ringraziare tutti, ma una cosa la posso fare: chiunque si senta escluso è libero di autoinvitarsi ovunque io sia e chiedermi di offrirgli una "pizza" (sappiamo tutti quale intendo) ed io sarò lieto di metterci pure la maionese sopra (un po' come se fosse la mia firma).#UnaRossinièpersempre

Elenco delle figure

Listings

Bibliografia

[1] A. Bagnoli. *Game Engines e MAS: Spatial Tuples in Unity3D*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2018.

[2] J. Blair and F. Lin. An approach for integrating 3d virtual worlds with multiagent systems. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, March 2011.

[3] BNG. WebSockets: A Guide. http://buildnewgames.com/websockets/.

[4] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6), 2013. Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[5] R. H. Bordini, H. J. Fred., and M. J. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007.

[6] M. Cerbara. *Stato dell'arte della progettazione automatica di programmi per robot*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[7] I. Fette and A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. http://www.rfc-editor.org/rfc/rfc6455.txt.

[8] M. Fuschini. *Tecnologie ad Agenti per Piattaforme di Gaming: un caso di studio basato su JaCaMo e Unity*. PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[9] J. Goodwin. *Learning Akka*. Packt Publishing Ltd, 2015.

[10] J. Gregory. *Game Engine Architecture*. CRC Press, 3. edition, 2019.

[11] I. Horswill. UnityProlog: A mostly ISO-compliant Prolog interpreter for Unity3D. https://github.com/ianhorswill/UnityProlog, Aug. 2015.

[12] J. Hunt. *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing, Cham, 2018.

[13] L. Inc. Play: The high velocity web framework for java and scala. https://www.playframework.com/, 2018.

[14] E. International. Standart ECMA-404: The JSON Data Interchange Syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, Dec. 2017.

[15] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. Gamebots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45(1), 2002.

[16] I. Lightbend. Akka: toolkit for building highly concurrent, distributed, and resilient message-driven applications for java and scala. https://akka.io/, 2019.

[17] G. LLC. Gson: A Java serialization/deserialization library to convert Java Objects into JSON and back. https://github.com/google/gson/graphs/contributors, Aug. 2008.

[18] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. RFC 6202, RFC Editor, April 2011. http://www.rfc-editor.org/rfc/rfc6202.txt.

[19] S. Mariani and A. Omicini. Game engines to model MAS: A research roadmap. In C. Santoro, F. Messina, and M. De Benedetti, editors, *WOA 2016 – 17th Workshop "From Objects to Agents"*, volume 1664 of *CEUR Workshop Proceedings*, pages 106–111. Sun SITE Central

Europe, RWTH Aachen University, 29–30 July 2016. Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016).

[20] Newtonsoft. Json.NET: Popular high-performance JSON framework for .NET. https://www.newtonsoft.com/json, Sept. 2007.

[21] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3), Dec 2008.

[22] A. Omicini and F. Zambonelli. MAS as complex systems: A view on the role of declarative approaches. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies*, volume 2990 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, May 2004. 1st International Workshop (DALT 2003), Melbourne, Australia, 15 July 2003. Revised Selected and Invited Papers.

[23] Oracle Corporation. Project Tyrus. https://tyrus-project.github.io/index.html, 2017.

[24] N. Poli. *Game Engines and MAS: BDI & Artifacts in Unity.* PhD thesis, Alma Mater Studiorum - Università di Bologna, 2017.

[25] P. Prasithsangaree, J. Manojlovich, S. Hughes, and M. Lewis. Utsaf: A multi-agent-based software bridge for interoperability between distributed military and commercial gaming simulation. *SIMULATION*, 80(12), 2004.

[26] A. S. Rao, M. P. Georgeff, et al. Bdi agents: from theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[27] A. Ricci, M. Viroli, and A. Omicini. Cartago: A framework for prototyping artifact-based environments in mas. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems III*, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[28] STA. websocket-sharp: A C# implementation of the WebSocket protocol client and server. http://sta.github.io/websocket-sharp/, Oct. 2010.

[29] U. Technologies. Colliders. https://docs.unity3d.com/Manual/CollidersOverview.html, 2019.

[30] U. Technologies. Event system. https://docs.unity3d.com/Manual/EventSystem.html, 2019.

[31] U. Technologies. Hierarchy. https://docs.unity3d.com/Manual/Hierarchy.html, 2019.

[32] U. Technologies. Rigidbody. https://docs.unity3d.com/ScriptReference/Rigidbody.html, 2019.

[33] U. Technologies. Transform. https://docs.unity3d.com/ScriptReference/Transform.html, 2019.

[34] U. Technologies. Unity. https://unity.com/, 2019.

[35] J. van Oijen, L. Vanhée, and F. Dignum. Ciga: A middleware for intelligent agents in virtual environments. In M. Beer, C. Brom, F. Dignum, and V.-W. Soo, editors, *Agents for Educational Games and Simulations*, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[36] Vocabolario Treccani. Treccani. http://www.treccani.it/vocabolario/.

[37] WRLD Team. WRLD. https://www.wrld3d.com/, 2015.

[38] F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, Nov. 2004. Special Issue: Challenges for Agent-Based Computing.