ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Dedalo, una soluzione stratificata per il routing in labirinti: strati 1 e 3, rappresentazione di un labirinto in una simulazione virtuale.

Elaborato in:
Sistemi Distribuiti L-A

Relatore:
Prof. Andrea Omicini
Correlatori:
Ing. Alessandro Ricci
Ing. Cesare Monti

Presentata da:
Stefano Bromuri

II Sessione
Anno Accademico: 2003-2004

Indice

1 Introduzione

Lo scopo di questa introduzione è quello di fare una panoramica delle astrazioni e dei modelli da cui si è partiti per progettare Dedalo. Veranno analizzate le due definizioni classiche di agente e le motivazioni per cui si è scelta un filosofia di progetto Agent Oriented per risolvere il problema proposto in questa tesi: il routing di veicoli/attuatori/robot nei labirinti.

1.1 Cos'è un agente software?

Questa sottosezione spiega le due definizioni classiche che sono state attribuite alla astrazione di agente.

1.1.1 Agente come Attribuzione

Una definizione che si può dare alla astrazione di agente è quella di agente come attribuzione. Prima di arrivare alla definizione, un fatto importante da notare, riguardo alla recente ricerca e sviluppo di software, è come ci sia poca comunanza tra i diversi approcci. C'è qualcosa che però possiamo riconoscere come somiglianza famigliare tra questi svariati punti di vista. "*Agent is that Agent Does*" è uno slogan che può semplicisticamente catturare l'essenza della questione: non si può definire la filosofia ad agenti attraverso una lista di attributi, perché questi dipendono dal contesto in cui viene applicata tale filosofia. La distinzione chiave tra gli approcci esistenti, nasce quindi dalle aspettative e dai punti di vista di chi osserva il problema relativo al ruolo degli agenti. L'affermazione più frequente di coloro che propongono questa filosofia è che, come un algoritmo, piuttosto che in una procedura, può più facilmente essere espresso come una funzione/capacità di un oggetto, così anche un comportamento può più facilmente essere espresso tramite la filosofia ad agenti. Tenendo conto di quest'ultima affermazione, si nota che, nonostante il relativismo teorico al riguardo della questione, un filo conduttore comune a tutte le posizioni teoriche effettivamente esiste. Partendo dalla definizione da vocabolario, un agente viene descritto come: "*qualcuno che può agire o ha il potere e l'autorità di agire*" o anche "*il motivo per cui qualcosa avviene*". Il termine deriva dal participio presente del verbo latino *agere: guidare, condurre, agire o fare.* In base a queste definizioni, si può dire che, l'attribuzione comune fatta in tutti gli approcci, architetture o linguaggi, è che: *un software agent agisce in modo da portare a termine un certo task a cui è stato assegnato, ha una certa conoscenza di questo task e inferisce le modalità migliori con cui portarlo termine, in modo da non dover specificare ogni singolo tedioso passo.*

1.1.2 Agente come Descrizione

La seconda definizione cerca un approccio più descrittivo al problema, elencando i possibili attributi, creando così varie categorie di agenti a seconda della combinazione di questi attributi. Una prima descrizione di software agent, che molti ricercatori potrebbero ritrovare accettabile, è la seguente: *un entità software che funziona continuamente e autonomamente in un ambiente particolare, spesso abitato da altri agenti.* La richiesta di continuità e autonomia deriva dalla necessità che un agente sia capace di portare a termine le proprie attività in modo flessibile e che risponda ai cambiamenti dell'ambiente senza richiedere costanti indicazioni. In aggiunta, ci si aspetta che un agente collocato in un ambiente assieme ad altri agenti, sia in grado di cooperare con essi. Per definire più chiaramente quanto detto, è possibile specificare una serie di attributi che un agente può avere per portare a termine il task per cui è stato progettato:
Reattività: abilità di agire in base ai sensi.
Autonomia: comportamento proattivo ed autonomo.
Socialità: capacità di collaborare con gli altri agenti dell'ambiente.
Alto livello di conoscenza sulla comunicazione: la capacità di comunicare con gli esseri umani con un linguaggio quanto più simile a una conversazione, piuttosto che al simbolismo tipico dei linguaggi di programmazione.
Capacità di inferenza: agire sulla specifica del task fornitogli utilizzando la conoscenza a priori dei goal, al fine di andare oltre all'informazione ricevuta e creare flessibilità.
Continuità temporale: la capacità di mantenere l'identità e lo stato per lunghi periodi di tempo.
Adattività: capacità di imparare e migliorare nel tempo.
Mobilità: capacità di migrare da una piattaforma a un'altra. La combinazione di questi attributi porta a considerare varie tipologie differenti di agenti, fino a giungere agli agenti DAI (Distributed Artificial Intelligence). Si può concludere questo paragrafo, dicendo che l'interesse nei MAS(multi-agent-system) come disciplina di studio, come paradigma di programmazione e come tecnologia, sta costantemente crescendo nell'ambito della ricerca.

1.2 Perchè gli Agenti Software?

Le motivazioni principali per utilizzare una metodologia basata sugli agenti sono due:

1. Semplificare gli aspetti legati alla programmazione concorrente.

2. Superare gli approcci basati sulle UI(user interfaces).

1.2.1 Programmazione concorrente e agenti

I metodi di programmazione procedurali e OO si basano su precondizioni, postcondizioni e invarianti. Queste condizioni sono però fortemente indebolite in ambito distribuito: infatti è possibile prevedere una sintassi delle operazioni, ma la semantica viene completamente persa perché non è possibile specificare il dominio della comunicazione. Sono necessari sistemi, come i MAS, in cui sia possibile specificare il dominio del discorso, cioè l'ontologia. La tipica invocazione di metodi si basa sul Design by Contract: in uno scenario sincrono, l'oggetto (procedura) chiamante attende fino a che l'oggetto chiamato non ha finito il suo task. Chiaramente l'oggetto chiamato si deve assicurare che la precondizione del metodo da invocare sia rispettata prima di invocarlo. In ambito distribuito, per sfruttare il parallelismo, vengono utilizzate altre politiche: asincrone o sincronismo differito. La correttezza delle precondizioni diventa un predicato di sincronizzazione. Come si capisce il Design by Contract risulta essere fortemente indebolito. È necessario disaccoppiare creando delle entità che incapsulino il loro flusso di esecuzione: gli agenti. Le architetture ad agenti, nascono quindi anche per superare l'accoppiamento tipico del Design by Contract e aggiungere ortogonalità al sistema. Bisogna specificare che un MAS è più che un mucchio di agenti: la metafora di società può e deve essere utilizzata come modello di interazione. Come fanno a comunicare e a essere sociali delle entità indipendenti, che incapsulano il loro flusso di esecuzione, che esistono proprio per creare ortogonalità nel sistema, come gli agenti? Nell'OOP questo problema non si poneva, con lo stile imperativo si ordinava all'oggetto di compiere una azione in un certo punto del flusso di esecuzione, e la responsabilità era completamente scaricata sul progettista. È necessario introdurre la metafora di artefatto di coordinazione. Gli artefatti di coordinazione servono per abilitare, governare, promuovere l'interazione tra gli agenti, in accordo a certe leggi di coordinazione, a loro volta definite dal linguaggio di comunicazione, la sintassi usata per esprimere lo scambio di strutture, e dal linguaggio di coordinazione, il set di primitive e la loro semantica. In particolare, in questo testo, verrà considerato il caso particolare di TuCSoN (evoluzione di Linda), che permette di ottenere quella ortogonalità che si cercava lasciando ampia possibilità di comunicazione: basandosi sugli spazi di tuple le entità possono interagire senza avere la conoscenza a priori l'una dell'altra, consentendo qualsiasi forma di comunicazione e sincronizzazione.

1.2.2 Agenti e delegazione vs. User Interfaces

Una motivazione agli agenti software ci viene data dalla necessità di superare i problemi relativi alle UI. Per la maggior parte dei task dell'utilizzatore le UI sono oggi lo standard: consentendo la diretta manipolazione, una UI richiede di essere visibile, cosí che l'utente sia costantemente informato su ciò che può manipolare. Purtroppo la maggior parte dei vantaggi che si hanno con la manipolazione diretta scompaiono quando i task crescono in scala di complessità. Quando il sistema diventa molto grande e distribuito diventa difficile rappresentarlo tramite interfacce grafiche, diventa difficile cercare esattamente quello che serve in queste in base al browsing o al tipico metodo di indicizzazione. Utilizzando invece la delegazione agli agenti, l'utilizzatore non sarà più costretto a specificare ogni azione, l'intelligenza e la flessibilità dell'agente software consentirà di specificare delle linee guida e di dimenticare i dettagli. Molte delle operazioni eseguite ora dall'utilizzatore, potrebbero essere delegate ai vari agenti software per raggiungere adattività e orientamento ai task.

2 Presentazione e analisi del problema

2.1 Presentazione del problema

Dedalo è un progetto che vuole creare una soluzione al problema del movimento di un attuatore attraverso un labirinto qualunque utilizzando la filosofia di programmazione ad agenti e la ricerca operativa. Nota la topologia del labirinto, l'attuatore(o il robot/compomponente software), posto all'entrata o in qualsiasi altro punto, deve trovare l'uscita passando attraverso il cammino minimo. Si vuole inoltre che la soluzione sia flessibile e consenta di utilizzare qualunque tipologia di labirinto e qualunque tipologia di attuatore, slegandosi dal caso particolare, in modo da avere un framework da cui partire ogni volta ci si trovi di fronte a un problema di questo genere. Creare un framework ha una importanza strategica, infatti il progettista ha una base da cui partire quando si trova di fronte a un problema simile o magari già in parte risolto dal framework. Affinchè il framework non venga cambiato nella sua architettura logica, è fondamentale progettare in modo modulare. Di conseguenza si vuole che l'architettura di base del framework sia costruita in modo da essere il più generale ed estendibile possibile.

2.2 Analisi del problema

L'analisi del problema può essere svolta considerando la seguente serie di concetti:

- Entità
- Attuatori/robot
- Comportamenti delle entità
- Ambiente e sue caratteristiche

2.2.1 Analisi di entità e loro comportamenti

Leggendo il problema e analizzandolo, un possibile set di entità che costituisca il sistema può essere descritto attraverso i seguenti ruoli: un *analizzatore di labirinti, un risolutore di percorsi e un esploratore di percorsi*. Chiaramente sono possibili altri set di ruoli e di entità, il consiglio dall'analista al progettista è di utilizzare questa struttura a causa della sua modularità. Ovviamente nessuno vieta di fare diversamente come accorpare in una sola entità sia la risoluzione del percorso che la sua esplorazione: in tal modo le entità guadagnano in complessità di comportamento, ma il sistema perde in

possibilità di distrubuzione. Un *analizzatore di labirinti* ha il seguente comportamento: preso un labirinto in un formato qualunque(gif, jpg, uno stream di dati, una foto da elaborare...) ne deve estrapolare informazione e lo deve comunicare al *risolutore di percorsi*. Questo ha il seguente comportamento: ricevuta le informazioni sul labirinto, si mette in comunicazione con l'utente finale, chiedendogli la posizione da cui vuole far partire l'*esploratore di percorsi*, dopo averla ricevuta la comunica all'esploratore che si preoccupa di spostarsi nella posizione di partenza. A questo punto il risolutore di percorsi richiede all'utente finale dove spedire l'esploratore all'interno del labirinto, riceve il punto in questione, calcola il cammino minimo per raggiungere la destinazione e spedisce il percorso all'esploratore, il quale naturalmente provvede a spostarsi lungo il cammino ricevuto. Il problema fondamentale appare chiaro e lampante: la coordinazione tra le entità. Il progettista dovrà assicurarsi di trovare un buon pattern per la comunicazione tra le varie entità e l'utente finale, in modo che non venga scaricata sull'utente finale la responsabilità della sincronizzazione e del buon funzionamento del sistema nella sua completezza. La divisione dei compiti in questo set di entità suggerisce un'altra di non meno importanza. Esulando ancora di più dal caso particolare, il progettista potrebbe decidere di considerare una società o più società di entità cooperanti tra loro. Chiaramente, dato che il fine ultimo di questa trattazione consiste nel creare un sistema distribuito per la risoluzione del routing attraverso labirinti, è consigliabile rendere il sistema più stratificato e quindi più distribuibile possibile.

2.2.2 Ambiente e sue caratteristiche

Ambiente nel nostro caso diventa sinonimo di labirinto: un labirinto può essere costruito con qualunque materiale, può essere un componente software, può essere con muri semicircolari, o dritti, verticali con o senza spessore, può essere un insieme di linee su un piano bidimensionale, oppure un insieme di muri in un piano tridimensionale. Di fronte a questa notevole varietà di casi, salta subito all'occhio un fatto: non si può dipendere dalla particolare forma del labirinto. Chiaramente però alcune regole sulla morfologia del labirinto vanno formulate: il cammino all'interno del labirinto deve poter essere rappresentabile con un grafo connesso (altrimenti non sarebbe possibile navigare il labirinto) al fine di poter applicare al grafo l'algoritmo di Dijkstra (o altri algoritmi), per il cammino minimo da un nodo a un altro nodo.

2.2.3 Il ruolo degli attuatori

Per attuatore si intende un particolare tipo di entità che ha il compito di svolgere una azione all'interno dell'ambiente in è posta. Una buona *rappresentazione* di attuatore è quella del robot dotato di ruote. Il campo non si riduce ovviamente solo a quest'ultima rappresentazione, ma come esempio per spiegare cosa si intende con la parola *attuatore*, è piuttosto calzante. In questo caso ogni attuatore diventa in realtà il corpo di un esploratore di percorsi. Ogni attuatore è situato in un ambiente a lui consono. Per come è stato definito l'ambiente nel framework di Dedalo, esso risulta completamente osservabile, quasi statico, ma non deterministico: un attuatore potrebbe trovare un ostacolo nel labirinto, oppure un altro attuatore che gli sbarra la strada. A quel punto dipenderà dal particolare attuatore se notificare o meno l'ostacolo a un esploratore di percorsi. È chiaro che quanto detto per le varie tipologie di labirinto, vale anche per gli attuatori: non si vuole in nessun caso dipendere dalla tecnologia utilizzata per rappresentare un attuatore, sia che si tratti di un robot o di un componente software, la complessità dell'interfacciamento deve essere nascosta all'utente finale.

3 Architettura Logica

In fase di analisi si è parlato di entità con un comportamento, con un ruolo e che comunicano tra loro: l'astrazione migliore per rappresentare entità di questo tipo, come discusso nell'introduzione, è quella di agente. Nella introduzione, si sono viste due possibili definizioni di agente, notando che possono esistere varie famiglie di agenti in base alla attribuzione o descrizione che decidiamo di dare. È sicuramente conveniente considerare in prima istanza alcune delle possibili architetture logiche di agente, al fine di spiegare meglio quale tipologia di agenti fa parte di Dedalo, e quale è il percorso naturale per definire tale tipologia.

3.1 Architettura logica di un agente generico

Si può partire col considerare la definizione che Stuart Russel/Peter Norvig, dal libro Artificial Intelligence, danno di un agente generico [1, page 17,18]:" *An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.*"

È una definizione ancora troppo generica per descrivere il comportamento degli agenti di Dedalo e l'architettura logica del sistema. Questa definizione introduce però il concetto di sensore e quello attuatore: i sensori sono il mezzo attraverso cui un agente riceve informazioni dall'esterno, mentre un attuatore è il mezzo attraverso cui l'agente compie azioni sull'esterno, modificandolo. Nel caso di Dedalo, sensore ed attuatore sono dei canali di comunicazione da cui ricevere e spedire informazione. Le percezioni che l'agente ha attraverso questi canali di comunicazione verso il mondo, possono essere elaborate per far reagire l'agente in base a delle precise regole interne. Si vorrebbe che l'agente fosse in grado di avere un comportamento razionale nei confronti del mondo. Per spiegare meglio questa ultima affermazione è necessario introdurre il concetto di *razionalità* e vedere come si relaziona al concetto di agente.

3.2 Definizione di razionalità

Si consideri la storia completa di ogni percezione di un agente: matematicamente parlando, si può dire che il comportamento dell'agente è descritto da quella che chiamiamo *agent function* che mappa ogni sequenza di percezioni in una azione. Si può immaginare di creare un tabulato per ogni agent function, dato un qualunque agente. A questo punto, con alcuni elementi in mano, si può accennare quello che si intende per Rational Agent. Dal libro Artificial Intelligence di Stuart Russel e Peter Norvig [1, page 21,22] : "*A rational*

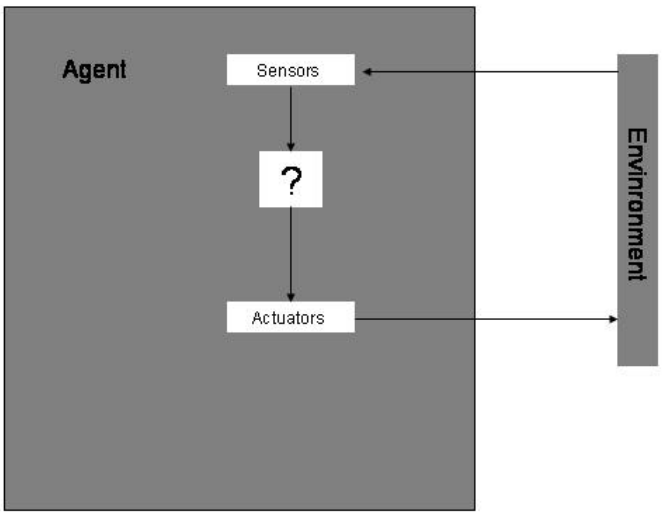

Figura 1: Architettura logica di un agente generico

agent is a one that does the right thing - conceptually speaking, every entry the table for the agent function is filled out correctly"
Questa definizione lascia dei punti interrogativi in sospeso: qual'è il criterio attraverso cui si decide qual'è la cosa giusta da fare a un certo determinato punto nel tempo? È necessario definire meglio quello che si intende per razionalità e quali sono i criteri attraverso cui poter dire che un agente si comporta razionalmente. La razionalità di un agente a un dato istante dipende da 4 cose:

- La misura di performance che definisce il criterio di successo
- Il comportamento prioritario dell'agente nell'ambiente
- Le azioni che gli agenti possono compiere
- Le sequenze di percezioni fino a un certo istante

Si può ora raffinare la definizione che abbamo dato di Rational Agent, sempre dal libro di Russel e Norvig [1, page 23,24]:

“ *For each possible percepts sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*”
Quindi per concludere: la razionalità di un agente dipende dalla rappresentazione che questo ha del mondo, dalla sequenza e accuratezza delle percezioni, dalle possibilità che ha di modificare e di imparare dall'ambiente, infine dal suo scopo nell'ambiente, il *task*, e dal dominio di questo cioè il *task environment*.

3.3 Task Environment

Dopo aver parlato di razionalità è necessario parlare di *task enviroment*. Un task è il problema di cui l'agente razionale è la soluzione, il task environment è il dominio del problema e può avere alcune proprietà. Un task environment può essere multi agente, parzialmente osservabile, stocastico, dinamico, discreto, episodico. Multi agente perché si può avere più di un agente che interagisce con l'ambiente. A causa di questa caratteristica il task environment può diventare: stocastico, perché un agente potrebbe non sapere cosa stanno facendo i suoi pari in ogni determinato momento; dinamico, perché un agente potrebbe continuare a compiere azioni in contemporanea agli altri; episodico e non sequenziale, perché il comportamento di un agente può non dipendere da una sequenza predeterminata di percezioni, trovandoci in ambiente dinamico e multi agente; parzialmente osservabile, in quanto un agente che si trova nel suo ambiente ne conosce le regole e la rappresentazione, ma non sa se incontrerà un ostacolo nella sua strada, non sa nemmeno se incontrerà un altro agente; infine il Task Environment può essere discreto: potrebbero essere osservabili solo alcuni aspetti dell'ambiente, mentre alcuni essere tralasciati.

3.4 Agenti DAI

Dopo aver parlato di razionalità, agenti razionali e task environment, si può parlare di una particolare famiglia di agenti. Gli agenti DAI(Distributed Artificial Intelligence) sono descritti dalla seguente lista di attributi:

- ha una rappresentazione del mondo
- è situato nel mondo
- risolve un problema che richiede intelligenza

- delibera e pianifica
- flessibile
- adattabile
- impara

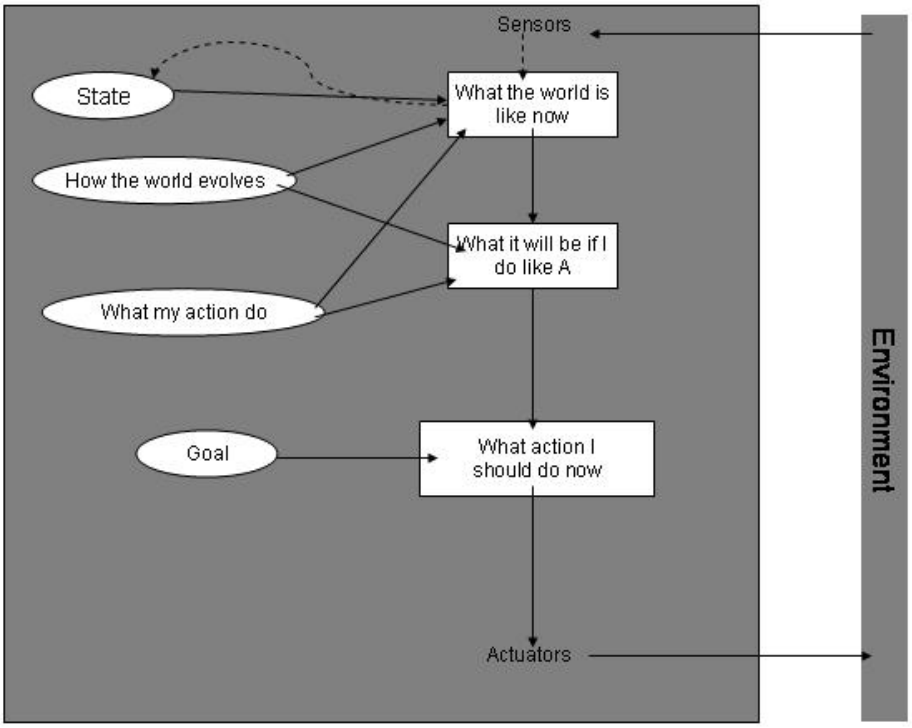

Figura 2: Architettura logica di un agente DAI

Un agente DAI ha una attività e uno scopo da perseguire. Ha bisogno quindi di una sorta di information goal che specifichi le situazioni che sono desiderabili. Cerca e Pianifica , al fine di raggiungere il proprio fine. Gli agenti di Dedalo cercando di seguire il pattern offerto dagli agenti DAI, con tutte le ovvie limitazioni del fatto che in Dedalo il task environment è sì multi agente, ma parzialmente statico, in quanto si assume che la topologia del labirinto sia conosciuta e non cambi nel tempo. Questo comporta il trovarsi in una

situazione ideale in cui l'agente in questione non ha bisogno di controllare le proprie percezioni per scegliere l'azione migliore da compiere, al fine di perseguire il proprio goal, cioè navigare il labirinto. L'architettura logica di un Agente-DAI è rappresentata in figura 2.

3.5 Dove si colloca Dedalo?

3.5.1 Distributed Problem Solving

Dedalo si colloca in un particolare campo della Intelligenza artificiale distribuita, cioè nel Distributed Problem Solving. DPS è il nome che si applica alla branca dell'intelligenza distribuita, che pone l'enfasi nella collabarazione tra agenti per risolvere un problema nel migliore dei modi. A causa della distribuzione delle risorse come la conoscenza, la capacità, l'informazione e l'esperienza tra gli agenti, un agente in un DPS system non è in grado di portare a termine da solo i task, cioè il compito assegnatogli, oppure al più è in grado di portarli a termine bene solo se lavora assieme ad altri agenti. Il risolvere problemi distribuiti richiede sia *coerenza di gruppo* che *competenza*. La coerenza di gruppo è difficile da realizzare tra agenti, proprio a causa della loro natura autonoma, ma senza è impossibile trovare una soluzione a un problema distribuito attraverso agenti multipli. In ogni caso ci si aspetta che un certo grado di coerenza sia già stato raggiunto: gli agenti in questo tipo di problemi, sono progettati per lavorare assieme. Il DPS si concentra anche sulla competenza, seguendo il principio dell'orchestra: per suonare una sinfonia non basta la voglia di suonare assieme, serve anche un certo grado di abilità nello strumento giusto.
Scendendo nel particolare, Dedalo sceglie una ben specifica strategia appartenente al dominio dei DPS, e cioè la decomposizione dei task: i task più grandi vengono suddivisi in sotto-task e portati a termine da differenti agenti.

3.5.2 Lo scambio di messaggi nei DPS

Riguardo allo scambio di messaggi, Dedalo utilizza una architettura a [3, page 37,38] "*Facilitator/Broker*". Nelle architetture di questo tipo, un agent broker fa l'intermediario tra un agent Requester e un agent Provider. In realtà Dedalo utilizza una versione estesa di questo pattern: una architettura a Blackboard. Questo tipo di architettura permette la cooperazione tra agenti e la conoscenza delle risorse comuni attraverso una lavagna virtuale. In questo modo si hanno alcuni vantaggi: ogni agente non è obbligato a conoscere tutti gli altri agenti, creando ortogonalità nel sistema, conseguentemente i messaggi scambiati diminuiscono in numero, non devono essere sincroni e sopratutto

non dipendo dall'infrastruttura in cui è stato creato l'agente. Per raggiungere questo scopo, come artefatto di coordinazione è stato utilizzato TuCSoN, che non solo offre una lavagna virtuale come deposito di informazione, ma si comporta anche come un nodo nella rete, rendendo massima la possibilità di distribuzione del software.

3.5.3 Gli agenti di Dedalo

Avendo in mano tutte i concetti necessari e il pattern a cui ci si vuole ispirare per la realizzazione degli agenti, possiamo specificare dove questi si collocano nello schema della architettura. Si possono anche fare ipotesi su dove si andrebbero concettualmente a collocare le possibili estensioni. Si deve specifiacare che gli agenti del framework sono collocati in base a quattro strati, di cui si discuterà in breve. Inoltre, in Dedalo gli agenti hanno tutti uno scopo da perseguire, sono quindi tutti pensati come agenti goal oriented, cercando di seguire il pattern del'Agente DAI: ogni agente deve essere distribuibile sulla rete e avere un Goal ben preciso, dipendendente dal proprio Task Environment. Come detto, il task environment scelto per il framework base di Dedalo è parzialmente statico, questo semplifica molto i problemi di percezione e l'architettura logica del sistema. Niente vieta di estendere il framework e la sua architettura logica in modo che il task environment sia dinamico, rendendo più complesse le percezioni e, di conseguenza, le risposte alle percezioni. Vengono di seguito elencati i layer di Dedalo assieme alla descrizione del loro ruolo, e vengono elencati anche gli agenti che verranno realizzati in base a questa struttura.

- Layer 3: appartengono a questo layer tutti gli agenti che inizializzano il sistema attraverso le risorse, rappresentate da tuple, che si vogliono condividere con altri agenti e tutti gli agenti che implementano una particolare politica di condivisione delle risorse.

- Layer 2: appartengono a questo layer tutti gli agenti che operano sulle tuple appartenenti al grafo del labirinto e precedentemente caricate in TuCSoN da agenti di layer 3.

- Layer 1: appartengono a questo layer tutti gli agenti che utilizzano attuatori per spostarsi o modificare l'ambiente.

- Layer 0: il layer 0 rappresenta l'ontologia del sistema, le politiche di comunicazione e coordinazione fra una o più società di agenti.

Avevamo specificato 3 agenti per rappresentare il framework di base di Dedalo.

- Init: attraverso le risorse rappresentanti un labirinto, ne estrapola il grafo corrispondente e inizializza TuCSoN. È anche responsabile di decidere quanti attuatori possono muoversi contemporaneamente sul labirinto

- PathSolverAgent: dialoga con l'utente per sapere quali sono le sue richieste riguardo allo spostamento dell'attuatore nel labirinto, calcola il cammino minimo per raggiungere la posizione e dialoga con ExplorerAgent, fornendogli, attraverso TuCSoN, i punti del percorso da seguire.

- ExplorerAgent: dialoga con agenti di tipo PathSolverAgent al fine di ricevere i punti su cui spostare l'attuatore. Muove l'attuatore attraverso un proxy che ha il compito di nascondere la complessità del particolare attuatore utilizzato.

In figura 3 è presente lo schema di Dedalo.

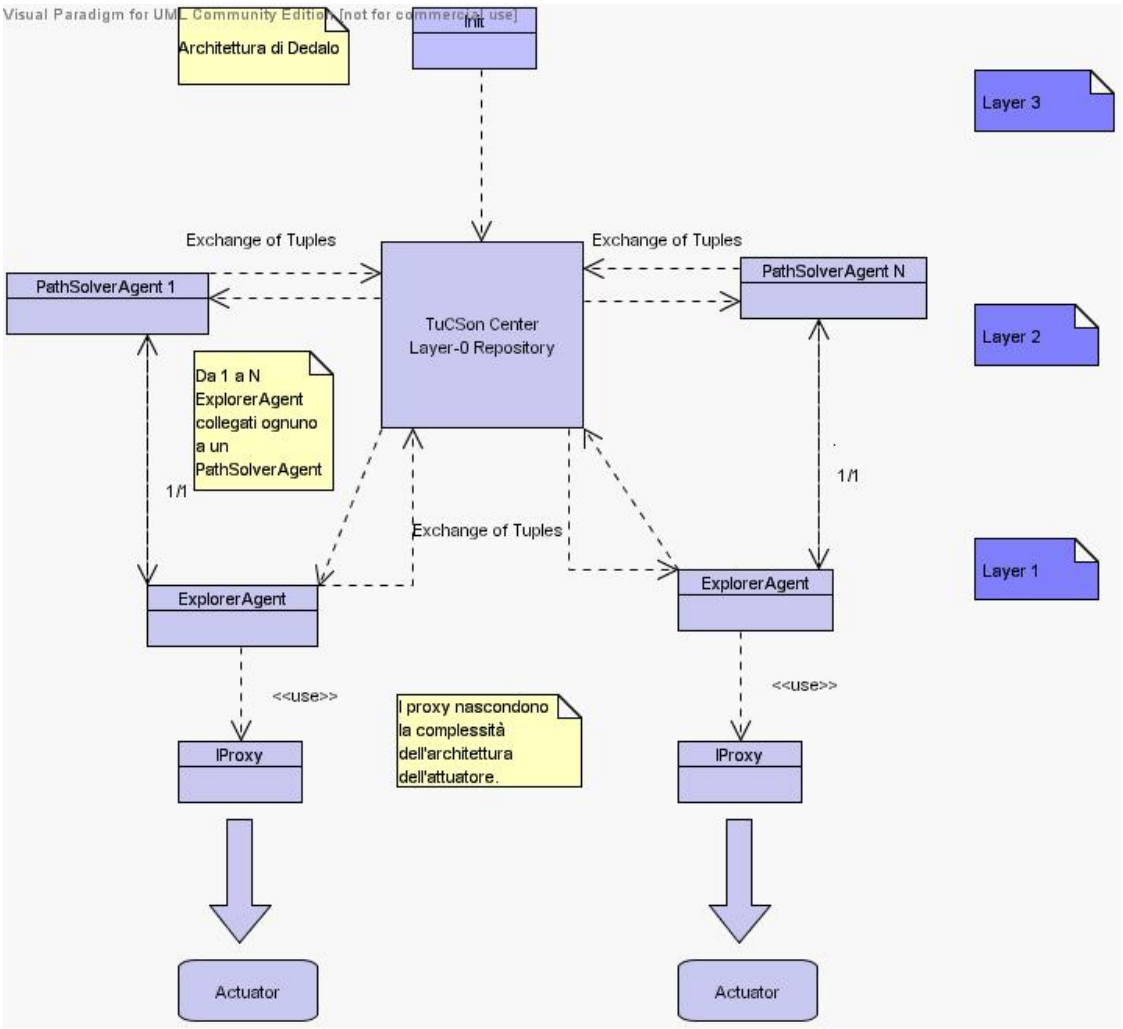

Figura 3: Architettura logica di Dedalo

4 Progetto del framework base di Dedalo

In questa sezione vengono presentati i quattro layer di Dedalo con specificato il ruolo di ogni layer nell'economia del sistema e alcuni possibili esempi di agenti e funzionalità appartenenti a ogni layer, oltre al progetto del comportamento degli agenti che saranno realizzati per ogni layer.

4.1 Layer 3

Gli agenti appatenenti al Layer 3 di Dedalo, si occupano della gestione delle risorse comuni che vengono inserite sotto forma di tuple in TuCSoN. Essendo la rappresentazione del grafo associato al labirinto in forma di tuple, diviene anche esso una forma di risorsa/conoscenza comune a disposizione di ogni agente che si collega al particolare nodo di TuCSoN. È chiaro che operando in questo modo, si rende gli agenti dei Layer inferiori completamente indipendenti dalla particolare risorsa che rappresenta il labirinto, oppure dalla particolare conformazione con cui questo è costruito: basta infatti rendere gli agenti degli strati inferiori in grado di ricostruire il grafo associato al labirinto attraverso le tuple. Quindi TuCSoN assume un doppio ruolo in questo contesto: il primo è quello di artefatto di comunicazione attraverso cui scambiare messaggi, il secondo quello di deposito della conoscenza comune.
Gli agenti del Layer 3 si occupano, oltre che di depositare la conoscenza e le risorse che dovranno condividere tutti gli agenti della architettura di Dedalo, anche dell'inizializzazione dell'artefatto di comunicazione: TuCSoN non offre solamente le funzionalità di una blackboard, ma può anche essere reso reattivo a particolari eventi, programmandone il comportamento. Per spiegare meglio questa affermazione consideriamo il seguente esempio: un agente si collega a TuCSoN e richiede (attraverso una operazione In(Tupla) applicata a un oggetto di tipo DefaultContext fornito con la libreria di TuCSoN) di ricevere una particolare tupla presente nella lavagna; TuCSoN, precedentemente programmato a reagire in un particolar modo a quella richiesta, prima di consegnare la tupla all'agente connesso, crea una tupla di backup sulla lavagna. Sfruttare adeguatamente questa caratteristica di TuCSoN, può portare a raggiungere comportamenti molto complessi: un altro agente, vedendo la tupla di backup, potrebbe dedurne che esiste un suo pari che in quel momento utilizza la risorsa a cui lui stesso vuole accedere, creando così una notevole capacità di inferenza; oppure, più semplicemente, un agente vedendo la tupla di backup potrebbe dedurne che la risorsa in quel momento è occupata e che mentre aspetta può impegnare il suo tempo portando a termine altri incarichi.
Per quanto riguarda il meccanismo di estensione, sia del framework in gen-

erale, che del layer 3 in particolare, si può affermare che esistono due strade: la prima delle due si basa sul meccanismo del già citato del task sharing, cioè di fronte a una estensione del task originale, la parte nuova viene portata a termine da un nuovo agente, estendendo la società degli agenti appartenenti al layer. Altrimenti, dato che il framework di Dedalo è costruito in Java, un'altra strada è sfruttare le proprietà di polimorfismo e ereditarietà proprie di questo linguaggio. È necessario far notare che la difficoltà nel seguire una o l'altra strada potrebbe variare molto a seconda del caso: nel primo caso la difficoltà riguarda il creare un agente che abbia le competenze necessarie per lavorare in coordinazione agli agenti dello stesso layer; nel secondo caso il problema diventa estendere un comportamento, mantenendone la coerenza e preservando la competenza di comunicare con gli agenti dello stesso layer e dei layer inferiori o superiori. Non si può affermare che uno dei due approcci sia meglio dell'altro, infatti l'efficacia della soluzione, in ognuno dei due casi, dipende dal task che si vuole risolvere, dal task environment, dal livello di distribuzione della computazione che si vuole ottenere, dalle intenzioni del progettista, dai vincoli imposti dagli altri agenti della architettura.

4.1.1 Progetto dell'agente Init

Init è un agente software pensato per appartenere al Layer 3 di Dedalo. Il suo comportamento è quello di un goal oriented agent il cui scopo è estrapolare il grafo associato a un labirinto e inserire tale grafo sotto forma di tuple in TuCSoN. Per estrapolare si intende: analizzare, secondo regole prefissate, un labirinto di cui si conosce la topologia, ricavando in tale modo nodi e archi relativi al grafo associato al labirinto. Come è già stato detto in fase di architettura logica, assumere come conosciuta la topologia di un ambiente, significa considerare tale ambiente stabile, o meglio quasi statico nel tempo. La parte riguardante la ricerca operativa è già risolta, non è necessario progettare anche questo aspetto: si può utilizzare la libreria VRP(vehicle Routing Problem) sviluppata dall'Ing. Luca Gardelli, in cui è presente la realizzazione, in termini di oggetti, delle entità nodo, arco e grafo. Al fine di rendere l'agente Init indipendente dal particolare algoritmo utilizzato per risolvere una particolare tipologia di labirinto, è bene delegare il compito agli oggetti che implementano l'interfaccia IMazeSolver: legandosi a una interfaccia, piuttosto che a uno specifico oggetto, si ottiene flessibilità e riutilizzabilità del codice, a patto che il contratto specificato nell'interfaccia sia rispettato dall'oggetto che la implementerà. In figura 4 è rappresentato il diagramma UML di Init e dell'interfaccia IMazeSolver. Il comportamento a runtime dell'agente Init è il seguente:

1. In run() viene richiamato il metodo init().

2. In init() viene ordinato a un oggetto implementante l'interfaccia IMaze-Solver di estrapolare il grafo associato a un labirinto tramite il metodo ComputeGraph().

3. Traduce il grafo precedentemente ottenuto in tuple e le carica in un nodo di TuCSoN attraverso un oggetto di tipo TuCSoNContextId, che permette di specificare la locazione del nodo nella rete.

4. Inizializza i token di comunicazione, rappresentati anch'essi da tuple, attraverso il metodo protetto DecideHowManySpeaking().

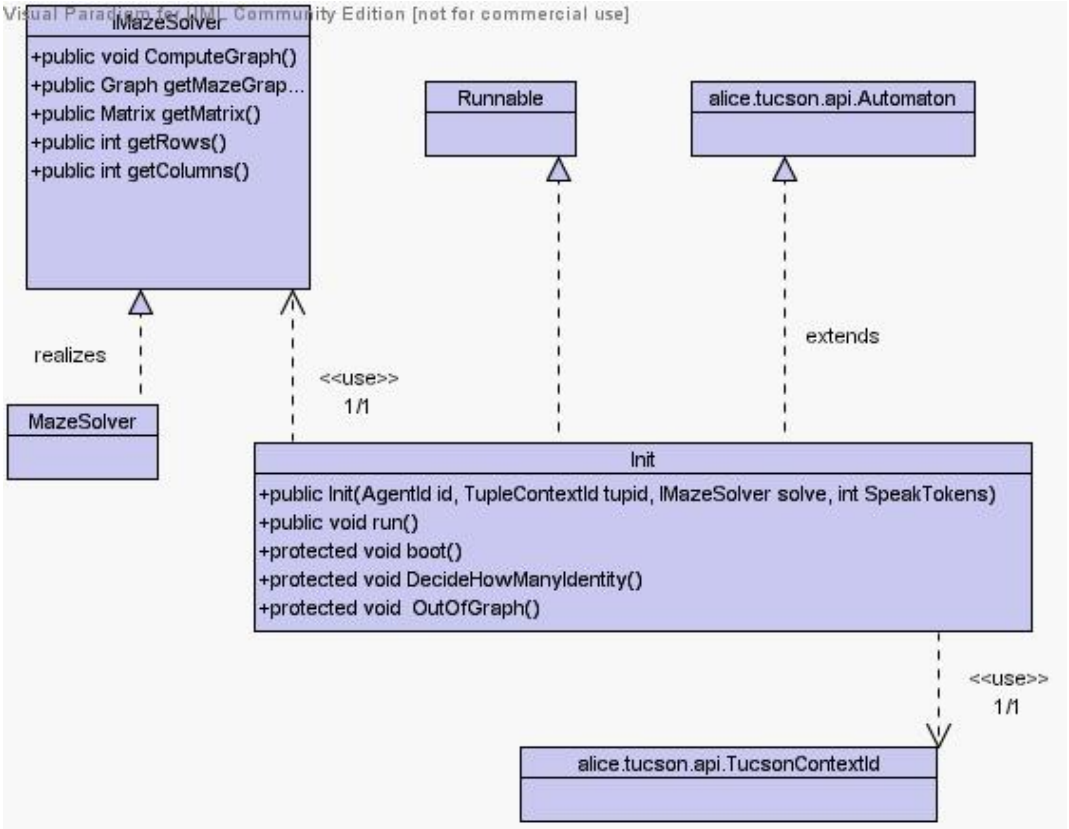

Figura 4: Diagramma UML di Init ed IMazeSolver

4.2 Layer 2: cenni

La trattazione completa del Layer 2 sarà eseguita in una tesi differente. Per seguire il filo logico del discorso, verrà comunque spiegato il ruolo degli agenti del layer 2 nella struttura di Dedalo.

Gli agenti appartenenti al layer 2 rappresentano, come gli agenti del layer 3, una società di agenti eterogenei. La tipologia di task a loro commissionati differisce molto da quelli commisionati ad agenti appartenenti al layer 3. In questo layer uno dei task degli agenti è quello di prendere conoscenza del grafo associato al labirinto attraverso le tuple in TuCSoN e applicarvi gli algoritmi classici dei problemi di ricerca operativa come: TSP, branch and bound, Dijkstra, Prim, Kruskal. Dopo aver applicato tali algoritmi, l'agente potrebbe comunicare a un operatore umano il risultato, oppure comunicarlo ad un altro agente al fine di portare a termine una particolare azione (ad esempio spostarsi nel labirinto da un nodo del grafo associato a un altro seguendo il cammino minimo) o entrambe le cose. Oltre all'applicazione di algoritmi sui grafi al fine di ricavarne informarzione da scambiare, un ruolo possibile degli agenti del layer 2 potrebbe essere quello di gestore del grafo: applicando una particolare politica di navigazione del grafo (es: nearest neighbour), potrebbe, attraverso il meccanismo dello scambio di messaggi, creare un comportamento proattivo del sistema, in cui i vari agenti del layer 2 calcolano i percorsi indicati dal gestore del grafo a seconda dell'algoritmo che utilizzano, invece che i percorsi specificati da un operatore umano. In modo equivalente, seguendo l'approccio di estensione del comportamento, si potrebbe estendere gli agenti che si hanno già (solo PathSolverAgent per ora) e renderli proattivi per comportarsi nello stesso modo.

4.2.1 Progetto dell'agente PathSolverAgent

Anche la trattazione del progetto dell'agente PathSolverAgent verrà considerata nei particolari in una tesi differente. In questa tesi viene fornita comunque un spiegazione del comportamento di questo agente e il diagramma UML.

1. Si collega a TuCSoN e prende visione del grafo, ricostruendolo attraverso le tuple.
2. Cerca un Agente di Layer 1 con cui comunicare.
3. Attende informazioni relative al nodo in cui posizionare l'atturatore.
4. Avverte l'agente di Layer 1 su come posizionare l'attuatore.
5. Attende informazioni da parte di un operatore umano, relative al nodo in cui spedire l'attuatore.
6. Spedisce i punti all'agente di layer 1 che servono per raggiungere, nel piano, il nodo specificato dall'operatore umano.
7. si rimette in attesa di informazioni dall'operatore umano.

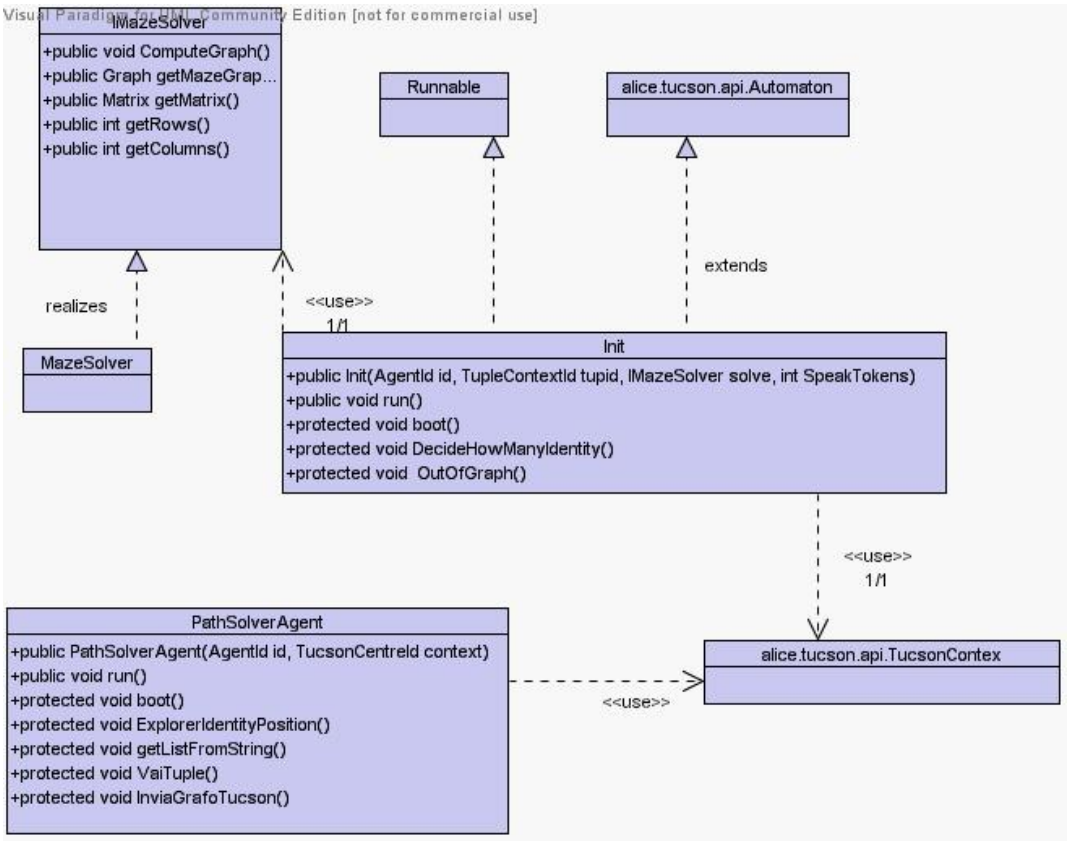

Figura 5: Diagramma UML di un PathSolverAgent

4.3 Layer 1

La società di agenti del Layer 1 è popolata da goal oriented agent che sono a stretto contatto con gli attuatori e la scena in cui agiscono questi ultimi. La complessità comportamentale degli agenti dipende molto dalla complessità dell'attuatore che utilizzano: risulta difficile esulare dal caso particolare. Inoltre ogni attuatore si interfaccia al calcolatore in cui risiede l'agente in maniera differente: una porta seriale, una porta a infrarossi, Bluetooth, con una socket, con un oggetto CORBA, oppure addiritturra attraverso TuCSoN stesso, se l'attuatore ha i connotati di un agente. Se risulta impossibile creare un caso generale per ogni diverso tipo di attuatore, risulta invece possibile nascondere la complessità dell'interfacciamento, in modo che due attuatori che si interfacciano in modo differente col calcolatore, ma che presentano stesse funzionalità, vengano utilizzati senza alcuna differenza da un agente in grado di sfruttare quelle particolari funzionalità. Sfruttando lo stesso principio, di fronte a un attuatore che offre più possibilità di quelle che l'agente

può gestire, queste verrebbero nascoste, lasciando all'agente unicamente la possibilità di accedere a quelle funzionalità che è in grado di gestire. Le funzionalità non sfruttate potrebbero esser date in mano ad un altro agente, seguendo il principio del task sharing (in questo caso, dividere il compito di controllare un attuatore). Java consente di ottenere questi risultati: la complessità dell'interfacciamento con l'attuatore può essere nascosta implementando il pattern del Proxy attraverso una classe wrapper. Quest'ultima, per massimizzare la riutilizzabilità del codice, può implementare un'interfaccia, in modo che a parità di funzionalità, ma con interfacciamento diverso all'attuatore, l'agente software sia ancora adeguato. Infine, per ottenere la compatibilità tra un agente che può sfruttare un numero limitato di funzionalità e un attuatore molto complesso, mantenendo il pattern del Proxy, si può utilizzare l'estensione multipla delle interfacce. Quest'ultimo concetto è spiegato meglio in figura 6.

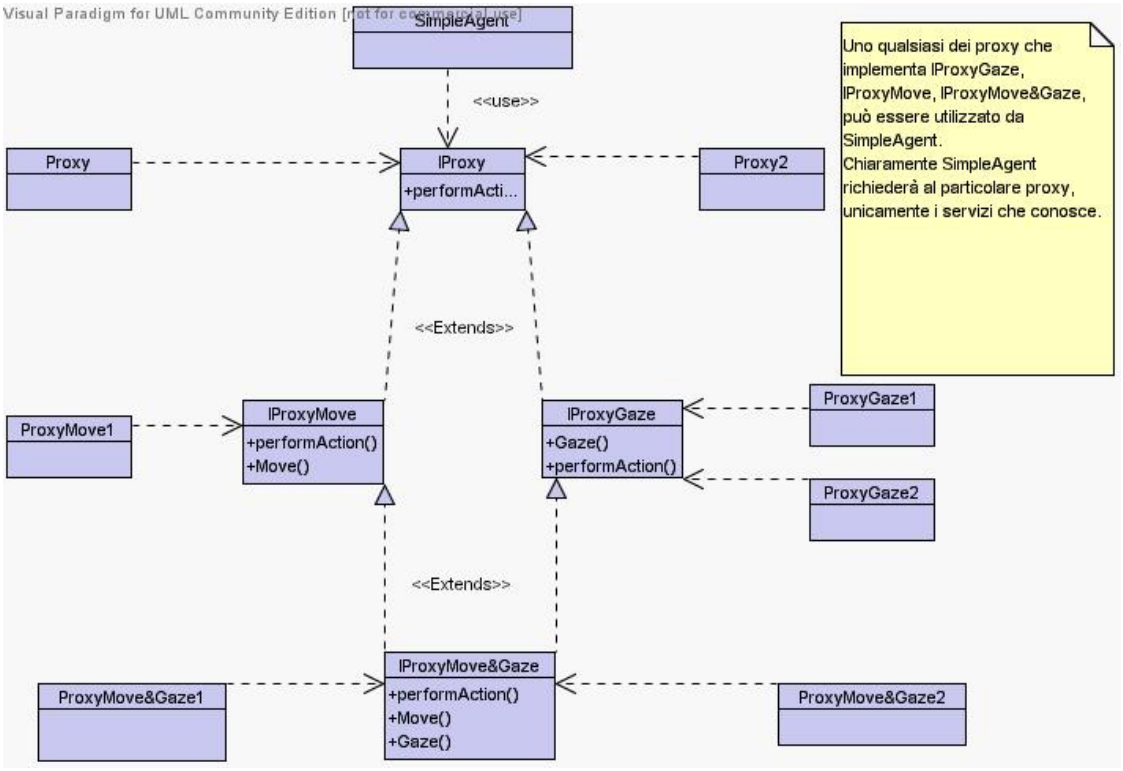

Figura 6: Schema esplicativo riguardo l'uso dei Proxy

4.3.1 Progetto dell'agente ExplorerAgent

ExplorerAgent è un agente che appartiene concettualmente al Layer 1 di Dedalo. Comunica con un PathSolverAgent attraverso TuCSoN: in base a uno scambio di tuple riceve da un PathSolverAgent il percorso che deve compiere in termini di punti da un nodo all'altro del grafo associato al labirinto. Al fine di nascondere la complessità dell'interfacciamento con l'attuatore, l'agente ExplorerAgent utilizza un interfaccia IProxyMove, che estende l'interfaccia IProxy. IProxyMove sarà implementata da un qualsiasi oggetto proxy funzionante come classe wrapper, fornendo all'esterno i servizi dell'attuatore. In particolare ExplorerAgent è in grado di sfruttare le potenzialità di un

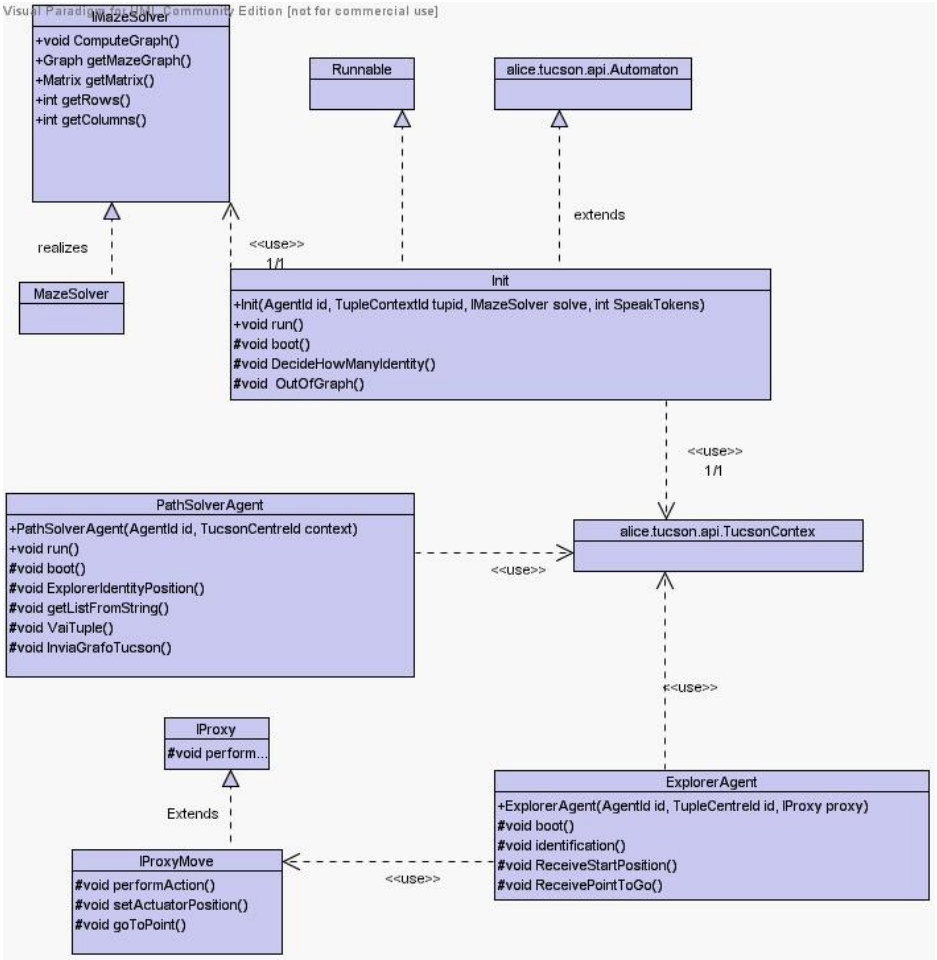

Figura 7: Diagramma UML di Dedalo

qualsiasi attuatore dotato della capacità di spostarsi su un piano. Per quanto riguarda il funzionamento dell'agente: Il metodo init() richiama i metodi

Identification() e ReceiveStartPosition(). Il primo, rispettivamente, serve a inserire una tupla di identificazione in TuCSoN così che un PathSolverAgent possa identificarlo come destinatario del discorso; il secondo metodo serve a richiedere la posizione in cui collocare l'attuatore a un PathSolverAgent. La funzione ReceivePointtoGo(), infine, serve per richiedere ciclicamente i punti in cui spedire l'attuatore. In figura 7 è possibile osservare il diagramma UML completo.

4.4 Layer 0: cenni

Anche questa parte verrà approfondita in una tesi differente. Si può comunque dire riguardo a questo layer, che ha lo scopo di rappresentare il dominio su cui si svolgono i dialoghi tra agenti in TuCSoN. Ogni qual volta si vuole rendere più complesso il dialogo nel sistema, è necessario estendere il dominio di comunicazione e di conoscenza, estendendo il set di tuple che rappresenta per l'appunto il sistema. Per comprendere fino in fondo il framework di Dedalo, nonostante siano solo dei cenni, è comunque necessario definire il set di tuple attraverso cui avverranno le comunicazioni tra gli agenti, verrà rappresentata la conoscenza e verranno rappresentate le risorse.

- NumeroNodi(Nnodi): rappresenta il numero totale di nodi del grafo
- NumeroArchi(NArchi): rappresenta il numero totale di archi del grafo
- arco(ArcId,FistNodeID,SecondoNodeId,StringofPoint): rappresenta l'arco nelle specifiche definite nella libreria Vehicle Routing Problem dell'ing. Gardelli
- Nodo(x,y,LogicalId,PhysicalId): rappresenta il nodo nelle specifiche definite nella libreria Vehicle Routing Problem dell'ing Gardelli
- EXPLORERIDENTITY(identity): rappresenta l'identità di un particolare ExplorerAgent
- EXPLORERPOSITION(identity,position): rappresenta la posizione dell'ExplorerAgent, specifica attraverso l'id logico del nodo in cui sarà posizionato l'attuatore.
- SpeakActToken(free): se presente nel tuple context rappresenta la disponibilità della risorsa di comunicazione, nel qual caso sia condivisa, oppure la possibilità di proferire parola, o meglio di richiedere punti a un PathSolverAgent, da parte di un ExplorerAgent.

- Vai(EXPLORERIDENTITY,NodoDiArrivo): rappresenta la tupla da cui un agente PathSolverAgent deve estrapolare il nodo di arrivo in cui spedire l'ExplorerAgent assegnatogli.
- PointToGo(EXPLORERIDENTITY,x,y): rappresenta la tupla con cui un PathSolverAgent comunica a un ExplorerAgent dove spostare l'attuatore.

5 Simulazione virtuale di un labirinto

In questa sezione viene considerata la costruzione di una simulazione virtuale attraverso cui testare le potenzialità del framework di Dedalo. Per risolvere questo particolare problema è stata utilizzata la libreria Java3D, rappresentando attuatori e labirinto come componenti software in un mondo virtuale a 3 dimensioni. Al fine di essere completi nella trattazione, veranno presentati alcuni brevi cenni sulla filosofia di programmazione su cui si impernia la libreria Java3D e come questi hanno influenzato la costruzione del mondo virtuale utilizzato nella simulazione. Subito dopo la breve trattazione su Java3D segue la risoluzione del caso particolare accennato: costruire un labirinto virtuale, con attuatori virtuali, al fine di testare il comportamento di Dedalo in condizioni *ideali*.

5.1 Cosa sono le API Java3D?

Le Java3D API sono una gerarchia di classi Java che servono come interfaccia per interagire con grafica tridimensionale. Java3D è una estensione standard di Java 2 JDK. Le API forniscono una collezione di costrutti di alto livello per creare e manipolare la geometria 3D. Java3D fornisce le funzioni per creare immagini, visualizzare grafica, fare animazioni e interagire con la grafica 3D.

5.1.1 Creare un universo virtuale in Java3D

In Java3D un universo virtuale è creato a partire da uno Scene Graph.
Quest'ultimo è creato usando istanze di Java3D classes. Lo scene graph è assemblato da oggetti per definire la geometria, i suoni, le luci, le locazioni e l'apparenza degli effetti visivi. Una definizione comune di grafo è quella di una struttura dati composta da nodi e archi. Un nodo è un dato, un arco è una relazione tra i nodi. Nello scene graph i nodi sono istanze di classi Java3D, gli archi rappresentano due tipi di relazioni tra le classi. La più comune è la relazione padre-figlio. In Java3D ci sono due tipologie di nodo: Group Node e Leaf Node. Un Group Node, può avere un qualsiasi numero di figli, ma un solo padre. Un Leaf Node può avere un padre, ma nessun figlio. L'altro tipo di relazione è la referenza. Una referenza associa un NodeComponent con un Nodo dello Scene Graph, un NodeComponent non si considera però appartenente allo Scene Graph. Un oggetto NodeComponent definisce gli attributi geometrici e dell'apparenza per renderizzare l'oggetto visivo. Un Java3D Scene Graph è costruito di oggetti di tipo Node in una relazione padre-figlio, formando una struttura ad albero. Nella struttura c'è una radice, gli altri nodi sono accessibili seguendo gli archi dalla radice. Gli

archi di un albero non hanno cicli. Lo Scene Graph è formato dagli alberi che hanno come radice l'oggetto Locale, che a sua volta è in relazione parent-child con un oggetto Virtual Universe. Esiste solo un path dalla radice (l'oggetto Locale) fino a un LeafNode. Il percorso da una radice a uno specifico LeafNode è lo Scene Graph Path. C'è uno Scene Graph Path per ogni LeafNode nello Schene Graph. Ogni Scene Graph Path specifica completamente le informazioni relative alla sua foglia come orientamento, descrizione e locazione dell'oggetto visivo.

5.1.2 Architettura di uno Scene Graph

La rappresentazione grafica di uno Scene Graph può servire come un tool di sviluppo o documentazione per un programma Java3D. Dopo che la rappresentazione è completa, diventa una specifica per il programma. Risulta anche essere una concisa rappresentazione del mondo.

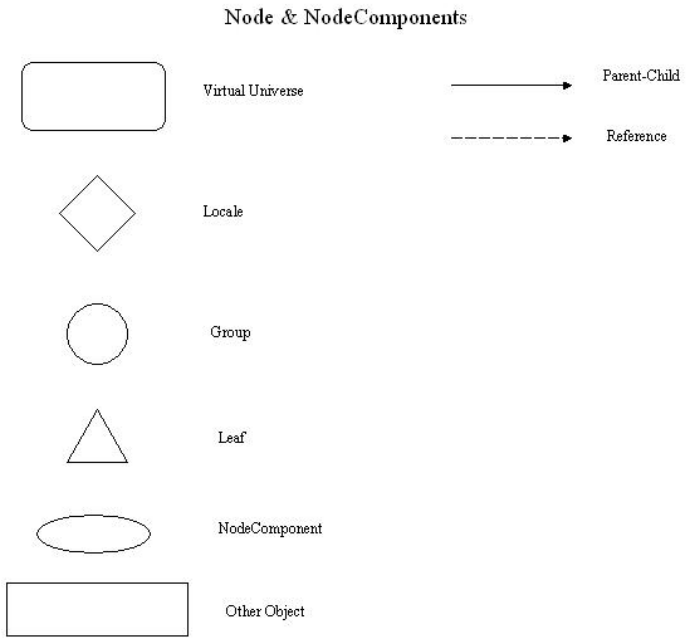

Figura 8: Convenzione per la rappresentazione di uno Scene Graph

In figura 8 è presente la convenzione utilizzata per rappresentare lo schema di uno Scene Graph.

5.2 Architettura del labirinto virtuale

Seguendo le convenzioni e la filosofia appena descritta, possiamo definire la struttura del labirinto virtuale che abbiamo intenzione di creare. Generalmente in Java3D lo Schene Graph ha due rami (branches) principali denominati Content branch e View branch, a seconda del ruolo che vengono a ricoprire. Il primo contiene gli oggetti da visualizzare, il loro colore, dove sono messi nello spazio e altro ancora; il secondo contiene tutto quello che riguarda "il vedere" la scena, ad esempio il punto di vista della telecamera. Sia il Content branch che il View branch sono istanze della classe BranchGroup, un particolare tipo di Node Group. Si creerà quindi un istanza dell'oggetto Locale come radice del nostro grafo e si aggiungeranno i due BranchGroup (BG nello schema) in relazione parent-child con la radice. Ai due BG andranno attaccati rispettivamente al primo gli oggetti da visualizzare, al secondo la telecamera (nello schema rappresentata dall'oggetto ViewPlatform). Si è deciso, in questo caso particolare, di rappresentare il labirinto come se diviso in caselle, composte ognuna da un pavimento (F nello schema) e da 4 muri (W nello schema). Ognuno di questi elementi appena citati è in relazione parent-child con un TransformGroup (TG nello schema), un particolare tipo di Node Group che permette di applicare delle trasformazioni geometriche ai figli di questo nodo. Ogni TG a sua voltà può essere in relazione parent-child con un altro TG. Come si è appena detto tutti i figli o le foglie di un TG, subiscono una trasformazione geometrica in base a una matrice Transform3D in relazione di referenza con il particolare TG Node. Avendo bisogno di rappresentare una sorta di scacchiera, il pavimento di ogni casella e conseguentemente i muri, devono essere creati in una zona ben precisa del nostro mondo virtuale, o meglio del nostro Locale. Per fare questo abbiamo bisogno di un TG per traslare le caselle, e di due TG per traslare i muri sulla casella e poi sui lati nord, sud, est, ovest, assumendo come nuove coordinate, dopo la prima traslazione, il centro del pavimento traslato. Lo schema della scena è rappresentato in figura 9. Notare che questo è solo lo schema di costruzione del labirinto, in seguito lo schema verrà modificato per considerare anche gli attuatori. Come detto prima ci sono due BranchGroup nello schema, in quello di destra mettiamo le visuali con cui vogliamo vedere la scena (ViewPlatform) in quello di sinistra mettiamo ciò che vogliamo visualizzare. Vediamo che il TG di cui ViewPlatform è figlio, ha una relazione di referenza con un oggetto nominato KeyboardBehavior. I behavior sono degli oggetti che rappresentano il comportamento del nodo a cui si referenziano (e quindi anche di tutto ciò che è figlio del nodo), in base a stimoli esterni particolari, come in questo caso la pressione dei pulsanti della tastiera che fa spostare la telecamera nella scena in quanto figlia del nodo TG in referenza

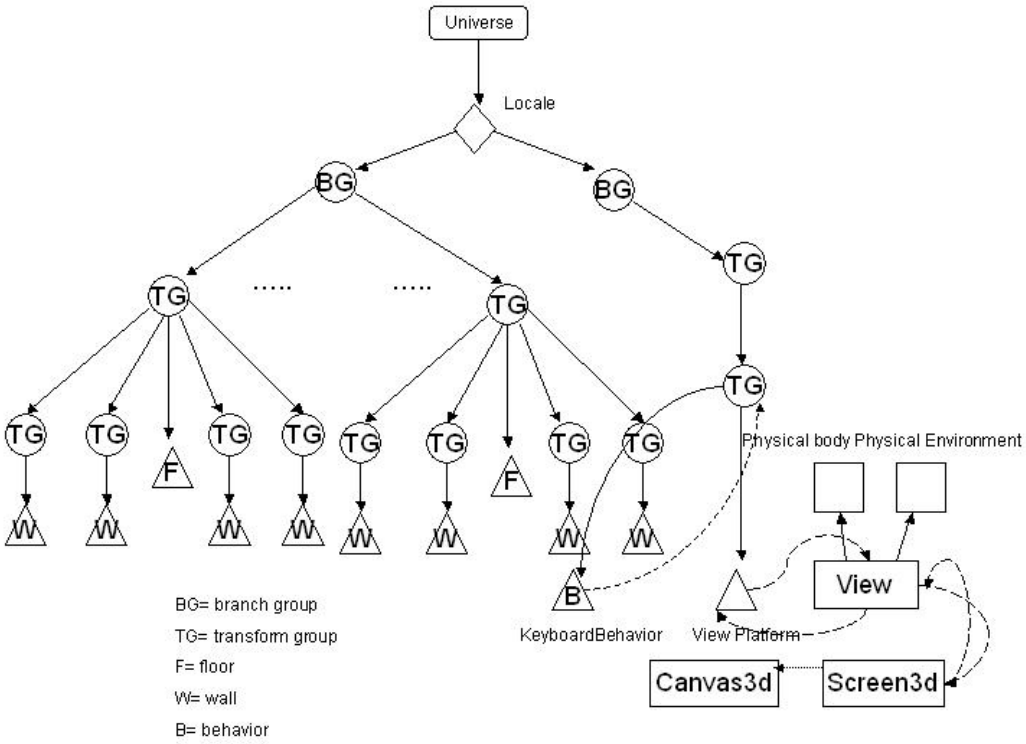

Figura 9: Schema di costruzione del Labirinto

con KeyboardBehavior.

5.3 Caso particolare del labirinto virtuale

Il caso particolare presenta un problema in più rispetto al problema generale risolto dal framework di Dedalo: bisogna infatti costruire un labirinto virtuale in tre dimensioni. Quello che si può fare è sfruttare la natura dichiarativa della conoscenza all'interno dell'artefatto di comunicazione: così come i PathSolverAgent ricostruiscono il grafo associato al labirinto, si può estendere la società degli agenti di Dedalo in modo che un agente, con il ruolo di creare una scena, crei un labirinto virtuale in base alle informazioni presenti su TuCSoN. Per fare questo però non basta un grafo associato al cammino interno nel labirinto: si perde l'informazione sulla disposizione dei muri e sulla morfologia particolare del labirinto. É necessario che l'agente Init del Layer 3 di Dedalo venga esteso al fine di rendere disponibile anche l'informazione relativa alla morfologia del Labirinto. Chiaramente essendo un caso particolare, sarà necessario scegliere anche una tipologia di labirinto da ri-

solvere, o meglio da cui estrapolare il grafo, con un MazeSolver adeguato. È infatti stato scelto, come caso esempio, un labirinto formato unicamente da muri orrizzontali e verticali, divisibile in piastrelle quadrate omegenee tra loro, rappresentato attraverso un'immagine bidimensionale in bianco e nero, in formato GIF. Per quanto riguarda la rappresentazione del labirinto in tre dimensioni, la costruzione della scena rispecchierà l'architettura decisa nel paragrafo precedente a questo è rappresentata in figura 9.

5.4 Architettura logica estesa

Come detto nella presentazione del caso particolare, l'agente Init del Layer 3

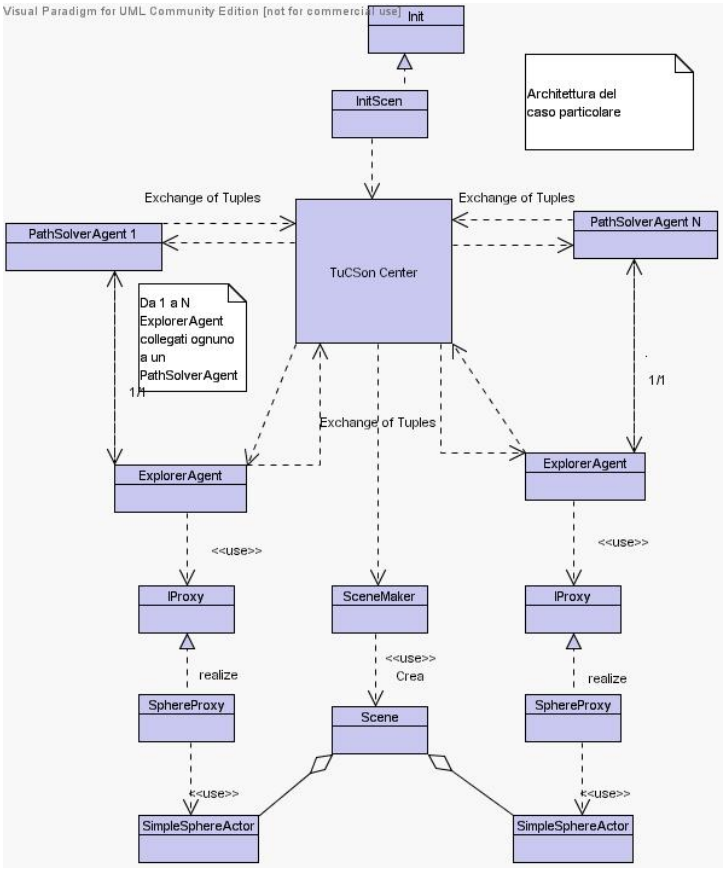

Figura 10: Architettura logica della soluzione particolare

deve essere esteso al fine di ottenere anche l'informazione sulla morfologia del labirinto che ci permetta in qualche modo di mantenere l'informazione sui muri. Nel paragrafo di presentazione del problema si è parlato di estendere Init in modo che depositi in TuCSoN, oltre al grafo, anche l'informazione

significativa per ricostruire una rappresentazione in tre dimensioni. Inoltre vogliamo anche estendere il Layer 1 in modo che la scena venga costruita da un agente e infine vogliamo che l'attuatore sia un componente software in grado di spostarsi nel labirinto virtuale, dotato cioè del servizio di movimento, che verrà gestito da un ExplorerAgent attraverso un oggetto che implementa opportunamente l'interfaccia IProxyMove. L'architettura logica estesa è rappresentata in figura 10.

5.5 Progetto della soluzione particolare

Come si è detto si devono definire nuovi componenti rispetto al framework di Dedalo: SceneMaker, SimpleSphereActor, SphereProxy, Scene. Altri componenti invece si devono solo estendere: Init. Iniziando da quest'ultimo, per estenderlo in modo che inserisca in TuCSoN anche la conoscenza relativa ai muri, è necessario scegliere un MazeSolver che presa la GIF in questione ne estrapoli informazioni utili.

5.5.1 MazeSolver

La figura scelta come labirinto per questa trattazione è rappresentata in figura 11. Si possono fare una serie di considerazioni sull'immagine: in primo

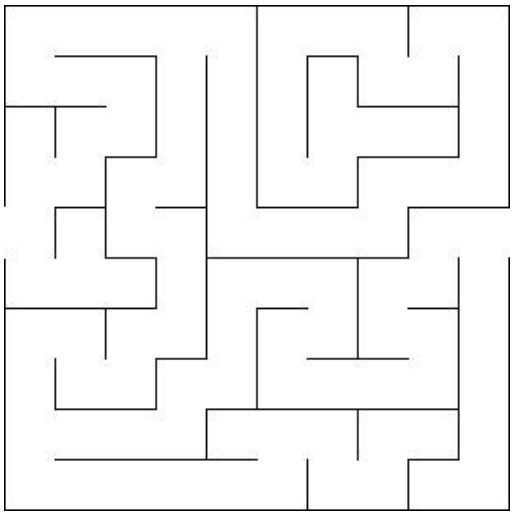

Figura 11: Un semplice labirinto preso come esempio

luogo risulta evidente che si può dividere in caselle ognuna di una tipologia diversa a seconda del numero e della disposizione dei muri su ogni casella. Questo significa che possiamo rappresentare il labirinto molto semplicemente

attraverso una matrice bidimensionale di oggetti di tipo Casella che mantengono l'informazione sui muri e sulla tipologia delle caselle dell'immagine. La matrice avrà un numero di righe e colonne pari rispettivamente al numero di caselle orrizzontali e verticali. Per ottenere le informazioni che servono, basterà scandire l'immagine prima casella per casella, e poi su ogni lato della casella, in base a delle semplici trasformazioni geometriche. Per quanto riguarda la creazione del grafo relativo al labirinto in questione, anche in questo caso è necessario fare delle scelte, esistono infatti più grafi possibili associabili a un'immagine di questo tipo, dipendenti, nella loro costruzione, dalla regola per la scelta dei nodi e degli archi. Al fine di non rendere troppo pesante la computazione, si è scelta una soluzione rispecchiante il fatto che alcune tipologie di Caselle del labirinto hanno delle particolarità rispetto alle altre:

1. Hanno 3 vie non sbarrate e una sola chiusa da un muro.

2. Hanno un vicolo cieco, con tre vie chiuse e una sola aperta.

Si è scelto di considerare queste due tipologie di caselle *interessanti* e utilizzarle come nodi del grafo, mentre tutte le altre caselle compongono gli archi di congiunzione tra i nodi del labirinto. Per creare il grafo di riferimento, è stata utilizzata la libreria VRP dell'ing. Luca Gardelli, estendendone la classe Arc, con la classe ListArc, in quanto si necessitava anche della lista dei punti che compongono l'arco, identificando come punti tutte le caselle, comprese quelle utilizzate per i Nodi, al fine di avere un sistema di coordinate cartesiane da utilizzare per lo spostamento dell'attuatore dotato di ruote. Dopo aver computato matrice e grafo, InitScene può depositare la conoscenza al riguardo di questi due componenti in TuCSoN.

5.5.2 InitScene

InitScene è l'estensione dell'agente Init, oltre a inserire in TuCSoN le tuple riguardanti il grafo, inserisce anche le tuple che identificano la matrice associata all'immagine da rappresentare in 3 dimensioni. Questo comporta una naturale estensione del dominio di comunicazione, cioè del Layer 0, attraverso le tuple del tipo:
Casella(numerocolonne,numerorighe,muronord,muroest,murosud,muroovest)
dove muronord, muroest, murosud e muroovest sono la rappresentazione in stringa di valori booleani. Lo stesso discorso sulla quasi staticità della conoscenza fatto nella presentazione della architettura logica del sistema e nella presentazione di Init vale anche qui: rappresentare la conoscenza attraverso delle tuple riguardanti un ambiente topologicamente stabile, significa dare una percezione statica o quasi del mondo a cui si applicano gli agenti.

5.5.3 SceneMaker

SceneMaker è un agente, che ha il compito di leggere da TuCSoN le tuple riguardanti la matrice del labirinto, ricostruire la matrice e creare quindi l'oggetto Scene.
La scelta di rappresentare la conoscenza relativa alla topologia del labitinto in maniera quasi statica, influenza ovviamente anche la costruzione della scena, la quale non varierà nel tempo. Una delle due motivazioni per cui si fa questa scelta, dipende dal fatto che un'immagine, a meno di situazioni particolari, non cambia nel tempo, per cui dipendendo la conoscenza da tale immagine, anche questa risulta essere statica o quasi, non bisogna dimenticare il fatto che più agenti possono far parte della scena e muoversi in essa, senza conoscere le intenzioni degli altri. La seconda motivazione riguarda la natura di questo caso particolare: abbiamo infatti specificato che, al fine di testare le potenzialità del framework, ci saremmo messi in condizioni ideali. Ovviamente, non si può sperare di essere sempre in condizioni ideali. Il dilemma è presto risolto, infatti, se si volesse un task environment dinamico, basterebbe ampliare la società di agenti che popola questo framework. Una soluzione potrebbe essere la seguente: per quanto riguarda il Layer 3, basta che una particolare estensione di Init, oppure un nuovo agente, si occupino di controllare continuamente se la conformazione generale del labirinto sta cambiando o meno e modificare di conseguenza la conoscenza relativa al labirinto; un altro agente di Layer 2 potrebbe controllare se le tuple (o in questo caso particolare anche la matrice) rappresentanti il grafo del labirinto sono cambiate o meno, avvertendo attraverso TuCSoN, il cambiamento avvenuto a tutti gli agenti di ogni layer della società. In questo modo, grazie alla rappresentazione dichiarativa della conoscenza offerta da TuCSoN, si ottiene una reazione dinamica e immediata al cambiamento dell'ambiente in base alla percezione che si ha di esso.

5.5.4 Scene

Scene è una classe Java che estende JFrame, inizializzata e richiamata da un SceneMaker. Questo JFrame ha la funzione della tela da disegno su cui verrà visualizzato il labirinto tridimensionale. Il labirinto segue le specifiche di costruzione già discusse, in più la classe Scene consente di aggiungere degli attori alla Scena, che verranno visualizzati in base alla loro forma geometrica. Rispetto allo schema presentato nella figura 9, bisogna quindi aggiungere un ramo in più al BranchGraph di sinistra. In figura 12 è presente la modifica da apportare.

Come si vede dalla figura, il nodo da aggiungere è incapsulato in un

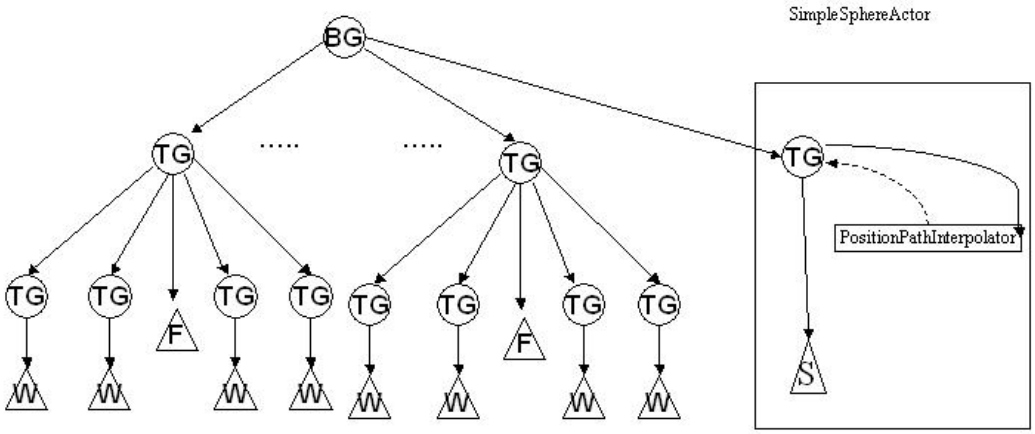

Figura 12: Modifica da apportare allo schema del labirinto

oggetto della classe SimpleSphereActor. Quest'ultima rappresenta l'attuatore/attore che da un lato verrà utilizzato da un ExplorerAgent e che è rappresentato come una sfera sul labirinto. Al fine di rendere anche visivamente la rappresentazione della simulazione, ogni SimpleSphereActor possiede un PositionPathInterpolator. Gli interpolatori sono delle estensioni particolari dei behavior della libreria Java3D che come si era detto, sono delle classi che reagiscono modificando la scena in base a un particolare stimolo. La trattazione completa del funzionamento di un interpolator, esula abbastanza dagli scopi di questa tesi, basti sapere che, come i behavior di cui abbiamo parlato nella sezione riguardante Java3D, essi reagiscono a uno stimolo modificando in qualche modo la scena. In particolare gli interpolator reagiscono ripetendo una animazione un numero n di volte in base al passare del tempo. L'animazione di cui si occupa il PositionPathInterpolator, è quella di spostare gli oggetti foglie di un particolare TG (TransformGroup) da un punto a un altro.

5.5.5 SimpleSphereActor e SphereProxy

SimpleSphereActor è sia un attore che fa parte di una scena tridimensionale Java3D, e quindi ha una rappresentazione geometrica che nel particolare è una sfera nera, sia un attuatore che offre dei servizi utilizzabili da un agente. Questi servizi riguardano la possibilità di posizionarlo a piacere o di spostarlo da un punto a un altro punto adiacente (tramite il PositionPathInterpolator). Per rispettare l'architettura logica del framework, l'attuatore deve essere incapsulato in un IProxy, SphereProxy per l'esattezza, che implementa l'interfaccia IProxyMove (che a sua volta estende IProxy) dando la possibilità all'ExplorerAgent di spostare l'attuatore a piacimento sulla scena. Nella figura 13 è disponibile un'immagine del sistema a tempo di esecuzione.

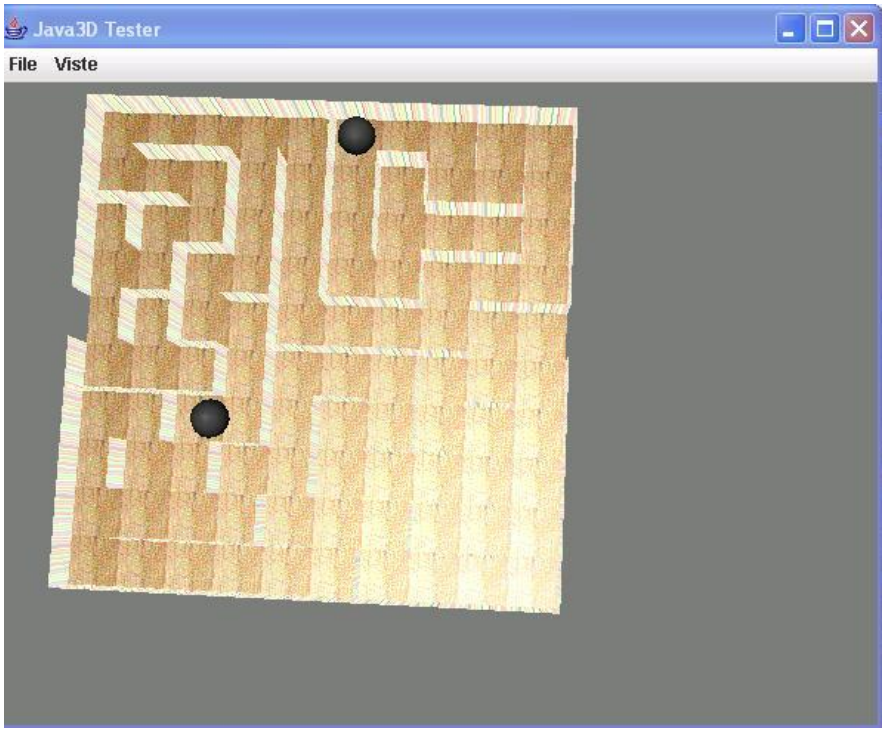

Figura 13: Dedalo in esecuzione

6 Conclusioni

6.1 Vantaggi, limiti e soluzione dei limiti

Costruire un framework Agent Based presenta alcune notevoli differenze rispetto alla classica programmazione a oggetti: in questa infatti tutto dipende dal Design by Contract, scaricando sullo sviluppatore la responsabilità di mantenere la coerenza del sistema. Un software Agent Oriented ingloba anche il suo flusso di esecuzione oltre al comportamento, e la coerenza del sistema viene creata attraverso la cooperazione e la comunicazione tra gli agenti. Tutto ciò porta a ovvi vantaggi in ambito distribuito. Nella vecchia logica del Design by Contract, si perdeva la visibilità di insieme di fronte a interfacce composte da migliaia di componenti, conducendo a situazioni impredicibili, cosa alquanto negativa in ambito distribuito dove la coordinazione tra le entità è fondamentale. Sicuramente la strada da seguire nel prossimo decennio sarà lo sviluppo di ambienti di lavoro in cui le nuove astrazioni di agente e società sostituiranno gradualmente, sopratutto in ambito distribuito, quella di interfaccia e oggetto. L'astrazione oggetto e il paradigma di programmazione imperativo, non sono più l'unica alternativa possibile: il meccanismo della richiesta, riassunto nella frase "An agent can say no", e della delega, stanno sempre più scalzando la logica dell'"impero ut fiat". Uno dei perchè di questo progetto, è sperimentare le nuove astrazioni, per dimostrare il grado di flessibilità possibile, ma anche i possibili limiti ancora non risolti: un limite che è apparso evidente nella trattazione riguarda l'estendibilità del comportamento degli agenti. Estendere un comportamento non è come estendere, nella vecchia maniera, un oggetto: per rendere più complesso un agente, si deve fare i conti con la coerenza del sistema, la coerenza del comportamento dell'agente, la sua competenza, la capacità di comunicare con gli altri agenti del sistema. Si potrebbe semplicisticamente affermare che il meccanismo di estensione è fortemente indebolito se si cerca di utilizzare il paradigma di programmazione ad agenti con un linguaggio come Java, dove l'estensioni delle classi è un meccanismo fondamentale. In realtà, la limitazione del meccanismo di estensione nasce dal fatto che si sta cercando di esprimere delle astrazioni diverse da quello per cui è stato concepito il particolare linguaggio: il Design by Contract si basa sul sincronismo, un unico flusso di esecuzione, tracciabile come una retta nel tempo (o come un cerchio anche se il ciclo infinito è considerato un errore nella letteratura classica). Nel meccanismo del Design by Contract manca completamente il concetto di *stato*, per questa motivazione diventa quasi impossibile estendere il comportamento di un agente, dal momento che è stato progettato in un certo modo. Si prenda come esempio il PathSolverAgent del framework di

Dedalo nel suo flusso di esecuzione, diviso in passi atomici:

1. Recupera il Grafo da TuCSoN
2. Recupera l'identità di un agente ExplorerAgent
3. Colloquia con un essere umano per richiedere il nodo a cui spedire un ExplorerAgent
4. Calcola il cammino minimo dal nodo in cui si trova attualmente l'ExplorerAgent
5. Spedisce i punti all'ExplorerAgent
6. Ricomincia dal punto 3.

Queste azioni sono comprese in un ciclo infinito,ma se si volesse aggiungere una funzionalità? Come ad esempio un comportamento proattivo dell'agente che senza diverse richieste da parte dell'utente, decide da se il percorso in cui spedire l'agente, una cosa semplice da dirsi, ma non facile da farsi: a che punto del ciclo si inserisce una operazione di questo genere? Se inserito prima del dialogo con l'utente, il dialogo non è ancora avvenuto e il comportamento proattivo descritto andrebbe avanti all'infinito, saltando sempre il dialogo con l'utente. Se lo inserito dopo il dialogo con l'utente, ha ancora meno senso, perchè il dialogo è gia avvenuto. Insomma bisogna per forza inventarsi degli espedienti. Una possibilità è estendere il dominio di comunicazione e spostare il dialogo con l'utente su un altro argomento: l'agente richiede se deve comportarsi proattivamente o meno prima di iniziare il secondo dialogo per richiedere il cammino da risolvere. Può sembrare effettivamente quello che è: uno spostare il problema a monte per mantenere il senso della comunicazione. L'unica soluzione possibile a problemi di questo tipo, anche se non di immediato impiego, risulta essere tradurre in qualche modo l'astrazione di stato e applicarla nella progettazione degli agenti. In questa maniera si può ottenere una potente incentivazione del metodo di estensione, ricordando però che quello che si sta estendendo non è un oggetto, ma un agente. Trattando questo problema si arriva anche a capire un'altro possibile connotato di un agente, e anche quale sarà la direzione seguita da questa filosofia di progettazione: è un bene progettare un agente dal comportamento complesso, come un automa a stati finiti, solo in questo modo ci si può distaccare completamente dai vincoli imposti dal Design by Contract, tipico della maggior parte dei linguaggi di programmazione esistenti. Progettando attraverso questa nuova astrazione, il comportamento dell'agente diventa in realtà una serie di decisioni in base a delle condizioni di stato, quindi non

più un solo flusso di esecuzione possibile, ma tanti flussi di esecuzione quante sono le possibili decisioni. In uno scenario di questo genere, l'estensione non diventa più un problema, inserendo un nuovo stato e una nuova condizione di stato, il comportamento dell'agente può essere opportunamente esteso.

6.2 Una possibile visione futura

La prima domanda che ci si pone di fronte a un progetto come questo è: "A che serve?". La prima risposta immediatamente dopo: "Al massimo servirà come una dimostrazione del livello di capacità raggiunto nell'ingegneria informatica". In realtà non è così. Si pensi in quante situazioni della vita reale si ha a che fare con un grafo connesso associato a un percorso e si capirà quale è il nocciolo della questione. Si consideri un caso particolare: il navigatore di un'automobile. In un navigatore è presente solo un certo numero di mappe stradali fornite al momento dell'installazione dell'hardware, mentre le altre mappe sono da installare in seguito, spendendo altri soldi. Se invece si distribuisse Dedalo nella rete, caricando in TuCSoN tuple riguardanti le mappe stradali esistenti, lo scenario cambierebbe: un agente del software dedicato potrebbe accedere a internet e scaricare di volta in volta la mappa necessaria. Potrebbe essere un PathSolverAgent particolarmente evoluto, che dopo aver scaricato la mappa, consiglia anche la strada migliore da seguire per raggiungere la destinazione. Non solo: essendo la mappa rappresentata da delle tuple su una lavagna virtuale, il tutto risulterebbe molto ottimizzato, in quanto l'informazione da scaricare sarebbe limitata. Si può pensare anche a qualcosa di più semplice: un sistema di monitoraggio per ambienti chiusi, con telecamere che si spostano da una stanza a un'altra in base a un percorso tracciato sul soffitto. In questo caso, avendo il framework base, il percorso e almeno una telecamera, il sistema è già realizzabile con minime modifiche. Per finire, si consideri un'ipotesi fantascientifica in campo biomedico: il sistema circolatorio umano è a tutti gli effetti aprrossimabile a un grafo connesso. Se si avesse un robot tanto piccolo da poter entrare in questo sistema, si potrebbe utilizzare Dedalo per navigarlo in modo che faccia il monitoraggio di punti chiave considerati come nodi principali del sistema.

Riferimenti bibliografici

[1] Stuart Russel, Peter Norvig, 2003. *Artificial Intelligence*, Prentice Hall.

[2] Dave Collins, 1996. *Designing object-oriented user interfaces* Benjamin/Cummings Publishing Company, Inc.

[3] Michael Luck, Vladimír Marík, Olga Stepankova, Robert Trappl, 2001. *Multi-Agent Systems and Applications* Springer.

[4] Jeffrey M. Bradshaw, 1998. *Software Agents* The MIT Press.

[5] The aliCE research Team, 2002. *TuCSoN Documentation*

[6] Alessadro Ricci, Andrea Omicini, Mirko Viroli, 2002. *Extending ReSpecT for Multiple Coordination Flows*

[7] Alessandro Ricci, Andrea Omicini, Mirko Viroli, Giovanni Rimassa, 2003. *Integrating Objective and Subjective Coordination in FIPA: A Roadmap to TuCSoN*

[8] Alessandro Ricci, 2004. *ENGINEERING SYSTEMS with COORDINATION MODELS and TECHNOLOGIES*

[9] Giovanni Rimassa, 2004. *From Distributed Objects to Multi-Agent Systems*Whitestein Technologies

[10] Dennis J. Bouvier, 2002. *Getting Started with the Java 3D API*Sun Mycrosystems.