

JAM: Java Agent Middleware

Progetto di Laboratorio per il corso di
Programmazione in Rete e Laboratorio

Anno Accademico 2009/2010

Matteo Baldoni

Attenzione! Questo documento non è definitivo ma potrà essere aggiornato con il progredire del corso. La presente versione è del October 12, 2009.

Abstract

Queste pagine descrivono il laboratorio per il corso di *Programmazione in Rete e Laboratorio* per l'anno accademico 2009/2010. Il presente laboratorio ha come obiettivo la realizzazione di una semplice infrastruttura di rete basata su RMI per lo sviluppo di applicazioni che utilizzino a tecnologia ad agenti. L'infrastruttura è denominata Java Agent Middleware (JAM, in breve).

Novità e modifiche:

October 12, 2009: Data di inizio della stesura del documento.

October 12, 2009: Completata la prima versione.

October 12, 2009: Versione corrente.

1 Modalità di svolgimento e consegna del laboratorio

È fortemente consigliato svolgere il laboratorio durante il corso e quindi sostenere l'esame nella prima sessione d'esame dopo il corso stesso.

1.1 Iscrizione del gruppo di laboratorio

Si ricorda che per poter svolgere il laboratorio è *necessario*:

1. iscriversi al corso di “*Programmazione in Rete e Laboratorio a.a. 2009/2010*” presso la piattaforma di supporto on-line ai corsi di informatica:

<http://informatica.i-learn.unito.it/>

(l'autenticazione *login/password* è quella solitamente utilizzata per l'accesso ai servizi riservati);

2. iscriversi compilando l'apposita “pagina wiki” presente nel supporto on-line al corso:

<http://informatica.i-learn.unito.it/>

nel quale devono essere riportati i componenti del gruppo (*al più due, limite rigido!*) secondo lo schema riportato di esempio. Tale iscrizione ha il solo ed unico scopo di tenere traccia della composizione del gruppo di laboratorio.

1.2 Forum di discussione e supporto on-line al corso

Mediante il supporto on-line saranno disponibili forum di discussione dedicati per gli argomenti affrontati durante il corso e per scambiare opinioni tra i vari gruppi di lavoro e con il docente. Sarà anche utilizzato per inviare avvisi nonchè per la consegna del codice in vista dell'esame (si veda la successiva sezione).

L'iscrizione al forum principale è effettuata automaticamente, è possibile disiscriversi ma è consigliabile farlo solo a seguito del superamento dell'esame per poter sempre ricevere in modo tempestivo le comunicazioni effettuate dal docente. Per ognuno dei forum disponibili è disponibile un canale *RSS* che riporta gli ultimi dieci post effettuati.

1.3 Materiale da portare alla consegna del laboratorio

Il laboratorio è organizzato in sette *parti* da svolgere in maniera incrementale. La consegna del laboratorio avviene mediante la piattaforma di supporto on-line al corso e secondo i tempi riportati in tali pagine (in genere qualche giorno prima della prova orale) effettuando l'upload di un archivio *zipped* (su UNIX: `zip -r nome_archivio nome_direttorio`). L'archivio compresso deve contenere:

- una cartella di nome “parte_I”, contenente il codice sorgente e compilato della prima parte di laboratorio descritta nella Sezione 3.1 compreso il test della Sezione 3.1.1;
- una cartella di nome “parte_II”, contenente il codice sorgente e compilato della seconda parte di laboratorio descritta nella Sezione 3.2 compreso il test della Sezione 3.2.1;
- una cartella di nome “parte_IV”, contenente il codice sorgente e compilato della quarta parte di laboratorio descritta nella Sezione 3.4, tranne la parte descritta nella Sezione 3.4.4 ma compreso il test della Sezione 3.4.2;
- una cartella di nome “jam”, contenente:

- il codice sorgente e compilato del progetto completo di tutte le sue parte, con tutte le istruzioni necessarie ad un facile utilizzo del software stesso (un file README.txt);
- la documentazione generata con *javadoc*;
- gli esempi e i test presenti alle Sezioni 3.1.1, 3.2.1, 3.4.2, 3.7.2, 3.7.3 e 3.7.4 svolti e correttamente funzionanti secondo le modalità ivi descritte.

All'esame si dovrà avere con se:

1. il listato stampato del codice sorgente;
2. i diagrammi di classe UML ed una breve relazione sul lavoro svolto (la relazione è obbligatoria solamente per chi deciderà di sostenere l'esame nelle sessioni successive alla prima, febbraio 2010).

Si consiglia di preparare le varie cartelle richieste durante lo svolgimento del laboratorio e *non alla fine*. Infatti, alcune parti richiedono di modificare o rivedere precedenti parti.

Il codice sarà scaricato dalla piattaforma al momento della discussione del laboratorio ma è comunque consigliabile portarsi con se una copia nel caso si verificasse qualche inconveniente.

1.4 Modalità d'esame

L'esame è *orale* ed *individuale* anche se il laboratorio è svolto in collaborazione con un'altra persona. Per sostenere l'esame ad un appello è obbligatorio prenotarsi on-line:

<https://www.educ.di.unito.it/studenti/>

L'esame ha l'obiettivo di verificare sia il corretto svolgimento del laboratorio che la comprensione delle parti di "teoria" correlata al laboratorio stesso.

Per poter discutere il laboratorio è necessario aver prima superato la prova scritta relativa al modulo di teoria.

L'ultima sessione utile per sostenere la prova orale di laboratorio sarà la sessione di settembre 2010.

1.5 Parti facoltative

È importante ricordare che il laboratorio potrà essere modificato e/o personalizzato per chi deciderà di sostenere l'esame nelle sessioni successive alla prima (febbraio 2010). In particolare, è da intendersi *facoltativo* lo svolgimento delle parti precedute dalla dicitura "*La seguente parte è facoltativa per la prima sessione d'esame*" per chi sostiene l'esame di laboratorio negli appelli della sessione di febbraio 2010 mentre è da intendersi *obbligatorio* per chi sostiene l'esame negli appelli successivi.

1.6 Calcolo del voto finale

Il voto per lo scritto e per il laboratorio sarà espresso in trentesimi. Il voto finale sarà formato dalla media pesata del voto della prova scritta e del laboratorio, secondo il loro contributo in CFU, e cioè

$$\text{voto finale} = (\text{voto dello scritto} * 2 + \text{voto del laboratorio}) / 3$$

La lode è attribuita nel caso si sia mostrata particolare brillantezza durante la parte di teoria e il laboratorio.

1.7 Validità del presente testo di laboratorio

Il presente testo di laboratorio vale solo fino all'inizio del corso di Programmazione in Rete e Laboratorio (parte di laboratorio) dell'anno accademico 2010/2011.

2 Descrizione generale del laboratorio

Il laboratorio è organizzato in diverse *parti* da svolgere in maniera incrementale. Lo scopo è la realizzazione una infrastruttura che permetta l'implementazione di applicativi basati sulla tecnologia ad agenti in Java. Per maggiori informazioni sui sistemi ad agenti si veda [10]. L'infrastruttura è ispirata al Java Agent Development Framework (JADE) di TILab [9].

2.1 Agenti e sistemi multiagente

(Questa breve introduzione è tratta dai lucidi del prof. Martelli [6])

Un agente è un sistema computazionale situato in un ambiente, e capace di azioni autonome nell'ambiente per raggiungere i suoi obiettivi di progetto.

Nella maggior parte dei domini un agente non ha completa conoscenza dell'ambiente e completo controllo su di esso. L'ambiente può evolvere dinamicamente indipendentemente dall'agente, le azioni dell'agente possono fallire. In generale, si assume che l'ambiente sia non-deterministico. Il problema principale per un agente è quello di decidere quale delle sue azioni dovrà eseguire per soddisfare meglio i suoi obiettivi di progetto. Un agente intelligente deve avere un *comportamento flessibile* per riuscire a soddisfare i propri obiettivi di progetto. Con flessibile si intende:

Reattivo. Gli agenti intelligenti sono capaci di percepire il loro ambiente e di rispondere in modo tempestivo ai cambiamenti che si verificano in esso.

Pro-attivo. Gli agenti intelligenti sono capaci di esibire un comportamento goal-directed prendendo l'iniziativa.

Sociale. Gli agenti intelligenti sono capaci di interagire con altri agenti (cooperare, negoziare, ...).

Spesso agli agenti vengono attribuite anche altre proprietà come ad esempio, la *mobilità*, la capacità di spostarsi fra i nodi di una rete; l'*apprendimento*, la capacità di imparare per migliorare le proprie prestazioni. Spesso gli agenti sono modellati per mezzo di concetti cognitivi basati su stati mentali come credenze (*beliefs*), conoscenze (*knowledge*), desideri, intenzioni. I modelli più noti sono quelli basati su *belief-desire-intention* (*BDI*, agenti basati su modelli intenzionali): *beliefs* corrispondono all'informazione che l'agente ha sul mondo. Può essere incompleta o non corretta. *Desires* rappresentano la situazione che l'agente vorrebbe realizzare. *Intentions* rappresentano i desideri che l'agente si è impegnato (*committed*) a realizzare. Il comportamento di un agente intelligente è descritto sinteticamente dal ciclo in Figura 1.

```

1 while (true) {
2   acquisisce una nuova percezione;
3   aggiorna le credenze (belief);
4   genera un insieme di opzioni (desideri);
5   sceglie fra queste una che intende realizzare (intenzione)
6   e per cui si impegna (commitment);
7   trova un piano per soddisfare l'intenzione scelta;
8   esegue il piano;
9 }

```

Figure 1: Ciclo di un agente intelligente.

Per poter interagire, gli agenti in un sistema multiagente devono comunicare fra loro. Il meccanismo di comunicazione normalmente usato è quello dello scambio di messaggi. L'invio o la ricezione di un messaggio può essere visto come una azione eseguita dall'agente. Sono stati definiti dei *linguaggi di comunicazione fra agenti* in cui le azioni comunicative sono modellate basandosi sulla teoria filosofica delle azioni: *inform*, *request*, *query-if*, *ask*, *tell*, ecc.

Ci sono ovvie somiglianze fra agenti e oggetti: entrambi sono entità computazionali che incapsulano uno stato, sanno eseguire delle azioni (metodi), e comunicano via scambio di messaggi. Ci sono però anche differenze significative: gli oggetti non hanno autonomia. Se un oggetto o ha un metodo pubblico, ogni altro oggetto può invocare questo metodo, e o, quando gli viene richiesto, deve eseguirlo. Viceversa, *i* chiede ad un agente *j* di eseguire un'azione, non è garantito che l'agente *j* la eseguirà (l'agente *j* è autonomo: *Objects do it for free. Agents do it because they want to*).

Un'altra differenza riguarda la nozione di comportamento flessibile (reattivo, proattivo, sociale). Il modello standard ad oggetti non considera la costruzione di sistemi che presentino questi tipi di comportamento. Inoltre i sistemi multiagente sono inerentemente distribuiti: ogni agente ha un flusso di controllo indipendente (processo o thread).

Le architetture software sono sempre più costituite da componenti che interagiscono dinamicamente attraverso protocolli complessi. L'interazione è probabilmente la caratteristica più importante del software complesso. La programmazione ad agenti viene vista da molti come un nuovo paradigma di programmazione, naturale evoluzione di altri paradigmi di programmazione, in partico-

lare della programmazione ad oggetti. Sono già state sviluppate metodologie per analisi e progetto agent-oriented, metodi formali di specifica e verifica di sistemi ad agenti e tecniche di implementazione.

2.2 Introduzione agli agenti in JAM

L'infrastruttura da realizzare è basata sulla tecnologia RMI di Java [5], si veda la Figura 2. Questa sarà costituita da un ambiente a runtime, denominato

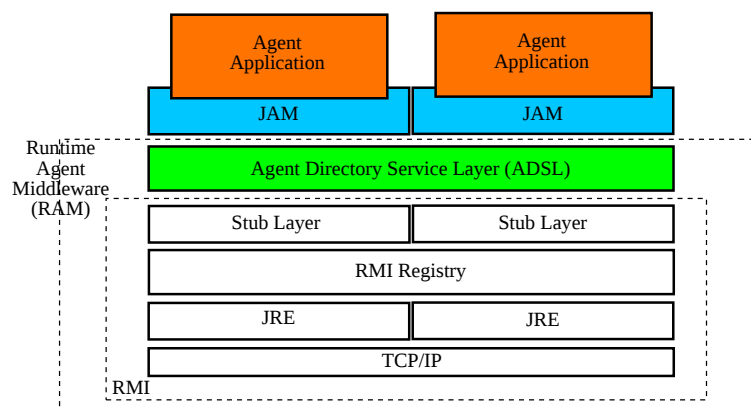


Figure 2: Descrizione generale del progetto da realizzare.

Runtime Agent Middleware (RAM, nel seguito). Il RAM sarà realizzato direttamente sull'infrastruttura RMI offerta da Java mediante una serie di classi che costituiranno lo *Agent Directory Service Layer (ADSL*, nel seguito). Il RAM permetterà di *ospitare, cercare, trovare, eseguire* e soprattutto scambiare messaggi tra gli agenti presenti in un dato momento. Agenti che saranno definiti mediante un insieme di opportune classi, che costituiscono un altro degli obiettivi del laboratorio, denominate *Java Agent Middleware (JAM*, nel seguito).

In JAM un agente è un oggetto, istanza di una particolare classe che estende la *JAMAgent* e che descrive il proprio *stato interno*. Il comportamento di un agente è definito da un insieme di oggetti di tipo *JAMBehaviour* che implementano il metodo *action* e che hanno accesso, mediante opportuno puntatore, allo stato dell'agente proprietario di tali comportamenti. In un certo senso, tale metodo rappresenta le “azioni” che definiscono il comportamento dell'agente stesso. Il codice (Java) di ogni metodo *action* è eseguito su un thread dedicato (vedi Figura 3 e Figura 4). Il metodo *action* è eseguito una volta soltanto se la classe che implementa il metodo estende *JAMSimpleBehaviour*, è invece eseguito ciclicamente fino a che il metodo *done* non è invocato, se la classe estende *JAMWhileBehaviour*.

In generale un agente in JAM è definito mediante la classe che descrive il proprio stato interno (la classe che estende *JAMAgent* e fornisce le funzionalità comunicative) e le classi che definiscono il suo comportamento. Ad esempio, le

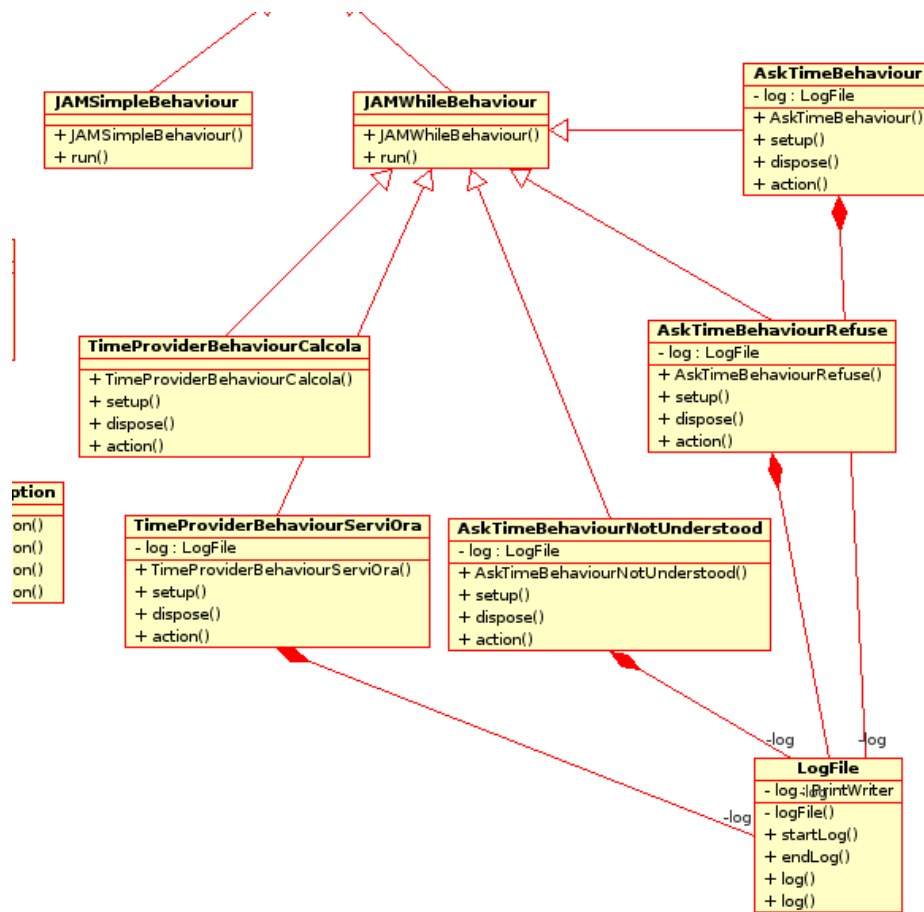


Figure 3: Diagramma a classi dell'organizzazione di un agente e del suo comportamento in JAM.

seguenti sono tre classi che definiscono tre comportamenti diversi:

```

1 class ComportamentoMioAgente extends JAMWhileBehaviour {
2   [ ... ]
3   public ComportamentoMioAgente(JAMAgent myAgent) {
4     super(myAgent);
5     [ ... ]
6   }
7   [ ... ]
8   public void setup() {
9     [ ... ]
10  }
11  public void dispose() {
12    [ ... ]
13  }
14  public void action() {
15    [ ... ]
16  }
17 }

```

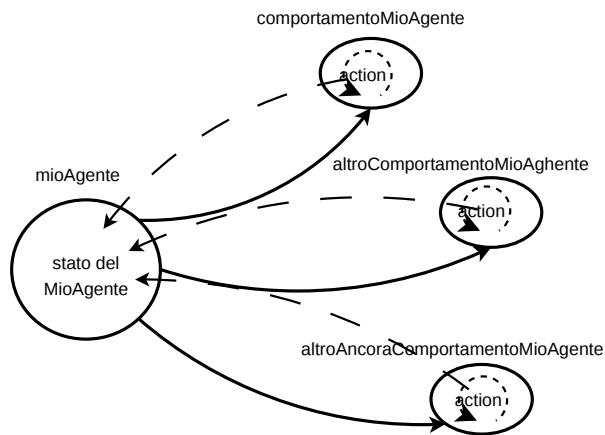


Figure 4: Agente e comportamenti come thread.

```

18
19 class AltroComportamentoMioAgente extends JAMSimpleBehaviour {
20     [ ... ]
21     public AltroComportamentoMioAgente(JAMAgent myAgent) {
22         super(myAgent);
23         [ ... ]
24     }
25     [ ... ]
26     public void setup() {
27         [ ... ]
28     }
29     public void dispose() {
30         [ ... ]
31     }
32     public void action() {
33         [ ... ]
34     }
35 }
36
37 class AltroAncoraComportamentoMioAgente extends JAMWhileBehaviour {
38     [ ... ]
39     public AltroAncoraComportamentoMioAgente(JAMAgent myAgent) {
40         super(myAgent);
41         [ ... ]
42     }
43     [ ... ]
44     public void setup() {
45         [ ... ]
46     }
47     public void dispose() {
48         [ ... ]
49     }
50     public void action() {
51         [ ... ]
52     }
53 }

```

Tali comportamenti possono essere associati alla definizione di un agente nel seguente modo:

```

1 class MioAgente extends JAMAgent {
2     [ ... ]

```

```

3  public MioAgente( ... ) {
4      [ ... ]
5      addBehaviour( new ComportamentoMioAgente(this) );
6      addBehaviour( new AltroComportamentoMioAgente(this) );
7      addBehaviour( new AltroAncoraComportamentoMioAgente(this) );
8  }
9  [ ... ]
10 }

```

L’attivazione di un agente avviene mediante invocazione del metodo *start* (dopo *init* che ha lo scopo di collegarlo alla *RAM*), il quale attiva i propri comportamenti, ognuno su un thread diverso.

```

1  [ ... ]
2  JAMAgent agent = new MioAgente( ... );
3  agent.init();
4  agent.start();
5  [ ... ]

```

Un agente può comunicare con altri agenti (conoscendone il nome (identificatore) o inviando messaggi a tutti gli agenti presenti in un dato momento) attraverso i metodi *send*. Inoltre può leggere i messaggi ricevuti dagli altri agenti mediante i metodi *receive*. Tipicamente tali metodi sono utilizzati all’interno del metodo *action*:

```

1  public void action() {
2      [ ... ]
3      Message request = new Message(Performative.REQUEST);
4      request.setSender(myagent.getMyID());
5      request.setReceiver( ... );
6      request.setContent("Che_ora_e'");
7      myAgent.send(request);
8      [ ... ]
9      Message answerRequest =
10         myAgent.receive(Performative.INFORM, ... );
11      [ ... ]
12  }

```

Dell’effettivo trasferimento dei messaggi via rete si occuperà la sottostante classe *JAMAgent* (la cui realizzazione è uno degli obiettivi del progetto).

2.3 Un esempio dattagliato: "Che ora è?"

Si vuole definire un agente di tipo *TimeProvider* che risponde a messaggi di tipo REQUEST (*performativa*, si veda anche [10, Sezione 8.2.3]) con *contenuto* “Che ora è?” inviando prima un messaggio di tipo AGREE se in grado di inviare l’ora e quindi si mette in attesa di un messaggio di tipo INFORM che confermi di volere ricevere l’ora, se questo messaggio arriva in un tempo finito di secondi, tale agente invia un messaggio di tipo INFORM contenente l’ora. L’ora è calcolata mediante un opportuno comportamento che incrementa ad intervalli di tempo regolari una variabile di tipo intero.

Se il messaggio inviato non è di tipo REQUEST allora l’agente di tipo *TimeProvider* invia come risposta un messaggio di tipo REFUSE. Se il *content* della REQUEST non è “Che ora è?” allora l’agente di tipo *TimeProvider* risponde con un messaggio di tipo NOT-UNDERSTOOD. Il diagramma di sequenza è riportato in Fig 5.

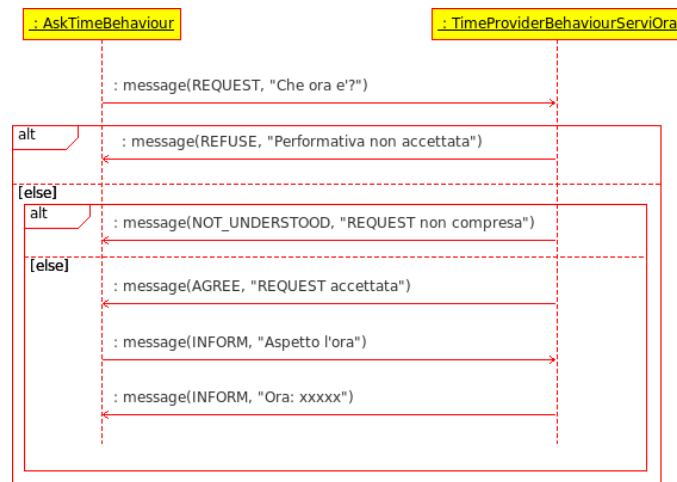


Figure 5: Diagramma di sequenza dell'esempio del time provider.

Il codice in Java con il supporto delle librerie **JAM** che definiremo è riportato di seguito¹. *TimeProviderBehaviourCalcola* è il comportamento che si occupa di calcolare l'ora, incrementando un timer ad intervalli regolari di tempo. Si noti come il codice presentato è completamente trasparente al fatto che si utilizzi RMI, il programmatore e l'utente di queste classi non necessitano di conoscere alcun particolare del funzionamento del *middleware*.

```

1 import JAM.*;
2
3 public class TimeProviderBehaviourCalcola extends JAMWhileBehaviour {
4     public TimeProviderBehaviourCalcola(JAMAgent myAgent) {
5         super(myAgent);
6     }
7     public void setup() throws JAMBehaviourInterruptedException {
8         ((TimeProviderAgent)myAgent).resetOra();
9     }
10    public void dispose() throws JAMBehaviourInterruptedException { }
11    public void action() throws JAMBehaviourInterruptedException {
12        ((TimeProviderAgent)myAgent).incrementaOra();
13        sleep(1000);
14    }
15 }
  
```

Il seguente è, invece, il comportamento che si occupa di rispondere ai clienti che inviano le richieste.

```

1 import JAM.*;
2
3 public class TimeProviderBehaviourServiOra extends JAMWhileBehaviour {
4     private LogFile log;
5     public TimeProviderBehaviourServiOra(JAMAgent myAgent) {
6         super(myAgent);
7         log = new LogFile();
8     }
9     public void setup() throws JAMBehaviourInterruptedException {
  
```

¹Tutto il codice che segue è disponibile presso il supporto on-line al corso.

```

10     try {
11         log.startLog(myAgent.getMyID().toString().trim(),
12             "Log_file_per_" + myAgent.getMyID());
13         log.log("0-Inizializzazione_comportamento.");
14     } catch(JAMIOException err) {
15         System.out.println("Errore:" + err);
16         done();
17     }
18 }
19 public void dispose() throws JAMBehaviourInterruptedException {
20     try {
21         log.log("END-Esecuguita_dispose.");
22         log.endLog();
23     } catch(JAMIOException err) {
24         System.out.println("Errore:" + err);
25         done();
26     }
27 }
28 public void action() throws JAMBehaviourInterruptedException {
29     try {
30         log.log("1-Start_del_comportamento, attesa_messaggio_iniziale.");
31         Message message = myAgent.receive();
32         log.log("2-Ricevuto_messaggio_iniziale, inizio_sua_analisi");
33         if(message.getPerformative() != Performative.REQUEST) {
34             log.log("3-Performativa_diversa_da_REQUEST, preparo_REFUSE.", message);
35             Message request = new Message(
36                 myAgent.getMyID(),
37                 message.getSender(),
38                 Performative.REFUSE,
39                 "Performativa_non_accettata",
40                 message
41             );
42             log.log("4-Invio_REFUSE", request);
43             myAgent.send(request);
44             log.log("5-Inviata_REFUSE", request);
45         } else if(!message.getContent().equals("Che_ora_e'")) {
46             log.log("6-Non_capisco_il_contento, preparo_NOT_UNDERSTOOD.", message);
47             Message not_understood = new Message(
48                 myAgent.getMyID(),
49                 message.getSender(),
50                 Performative.NOT_UNDERSTOOD,
51                 "REQUEST_non_compresa",
52                 message
53             );
54             log.log("7-Invio_NOT_UNDERSTOOD.", not_understood);
55             myAgent.send(not_understood);
56             log.log("8-Inviata_NOT_UNDERSTOOD.", not_understood);
57         } else {
58             log.log("9-REQUEST_accettabile, preparo_AGREE.", message);
59             AgentID agentid = message.getSender();
60             Message agree = new Message(
61                 myAgent.getMyID(),
62                 agentid,
63                 Performative.AGREE,
64                 "REQUEST_accettata",
65                 message
66             );
67             log.log("10-Invio_AGREE.", agree);
68             myAgent.send(agree);
69             log.log("11-Inviata_AGREE.", agree);
70             log.log("12-Mi_metto_in_attesa_conferma_da_" + agentid + ".");
71             sleep(1000);
72             log.log("13-Mi_risveglio_e_verifico_se_arrivata_conferma_da_" +
73                 agentid + ".");
74             if(!myAgent.isThereMessage(agentid, Performative.INFORM)) {
75                 log.log("14-Nessuna_conferma_da_" + agentid + ".");
76             } else {
77                 Message confirm = myAgent.receive(agentid, Performative.INFORM);

```

```

78         log.log("15-␣Arrivata␣conferma␣da␣" + agentid + ".", confirm);
79         if(confirm.getContent().equals("Aspetto␣l'ora.")) {
80             log.log("16-␣Preparo␣messaggio␣con␣ora␣per␣" + agentid + ".");
81             Message inform = new Message(
82                 myAgent.getMyID(),
83                 agentid,
84                 Performative.INFORM,
85                 "Ora:␣" + ((TimeProviderAgent)myAgent).getOra()
86             );
87             log.log("17-␣Invio␣ora␣a␣" + agentid + ".", inform);
88             myAgent.send(inform);
89             log.log("19-␣Inviata␣ora␣a␣" + agentid + ".", inform);
90         }
91     }
92 }
93 } catch(JAMADSLEException err1) {
94     System.out.println("Errore:␣" + err1);
95     done();
96 } catch(JAMIOException err2) {
97     System.out.println("Errore:␣" + err2);
98     done();
99 }
100 }
101 }

```

La definizione dell'agente di tipo *TimeProviderAgent* si riduce ora a creare i vari comportamenti e l'orologio come parte del proprio stato. Un agente di tipo *TimeProviderAgent* si può ora creare in un main qualsiasi. Nell'esempio che segue sono creati due agenti nel *main* della stessa classe *TimeProviderAgent*.

```

1 import JAM.*;
2
3 public class TimeProviderAgent extends JAMAgent {
4     private int orologio;
5     public TimeProviderAgent(String s) throws JAMADSLEException {
6         super(new PersonalAgentID(s, "Time␣Provider"));
7         addBehaviour(new TimeProviderBehaviourCalcola(this));
8         addBehaviour(new TimeProviderBehaviourServiOra(this));
9     }
10    public void resetOra() {
11        orologio = 0;
12    }
13    public void incrementaOra() {
14        orologio++;
15    }
16    public int getOra() {
17        return orologio;
18    }
19    public static void main(String args[]) throws JAMADSLEException {
20        TimeProviderAgent timeprovideragent =
21            new TimeProviderAgent("Berlino");
22        TimeProviderAgent timeprovideragent1 =
23            new TimeProviderAgent("Parigi");
24        timeprovideragent.init();
25        timeprovideragent.start();
26        timeprovideragent1.init();
27        timeprovideragent1.start();
28    }
29 }

```

Il codice del comportamento di un agente che richiede l'ora può essere il seguente:

```

1 import JAM.*;
2
3 public class AskTimeBehaviour extends JAMWhileBehaviour {

```

```

4  private LogFile log;
5  public AskTimeBehaviour(JAMAgent myAgent) {
6      super(myAgent);
7      log = new LogFile();
8  }
9  public void setup() throws JAMBehaviourInterruptedException {
10     try {
11         log.startLog(myAgent.getMyID().toString().trim(),
12             "Log_file_per_" + myAgent.getMyID());
13         log.log("0-Inizializzazione_comportamento.");
14     } catch(JAMIOException err) {
15         System.out.println("Errore:" + err);
16         done();
17     }
18 }
19 public void dispose() throws JAMBehaviourInterruptedException {
20     try {
21         log.log("END-Esecuguita_dispose.");
22         log.endLog();
23     } catch(JAMIOException err) {
24         System.out.println("Errore:" + err);
25         done();
26     }
27 }
28 public void action() throws JAMBehaviourInterruptedException {
29     try {
30         log.log("1-Start_del_comportamento,attesa_messaggio_iniziale.");
31         CategoryAgentID timeProviderCategoryID = new CategoryAgentID("TimeProvider");
32         log.log("2-Preparo_REQUEST_dell'ora.");
33         Message request = new Message(
34             myAgent.getMyID(),
35             timeProviderCategoryID,
36             Performative.REQUEST,
37             "Che_ora_e'?"
38         );
39         log.log("3-Invio_REQUEST.", request);
40         myAgent.send(request);
41         log.log("4-Inviato_REQUEST.", request);
42         log.log("5-Attendo_risposta.");
43         Message answer = myAgent.receive(timeProviderCategoryID);
44         log.log("6-Trovata_risposta.", answer);
45         if(answer.getPerformative() != Performative.AGREE) {
46             log.log("7-La_risposta_non_e'una_agree_eseguo_una_done.", answer);
47             done();
48         } else {
49             log.log("8-La_risposta_e'una_agree_prepara_la_conferma.", answer);
50             AgentID selectedProvider = answer.getSender();
51             Message confirm = new Message(
52                 myAgent.getMyID(),
53                 selectedProvider,
54                 Performative.INFORM,
55                 "Aspetto_l'ora."
56             );
57             log.log("9-Invio_la_conferma.", confirm);
58             myAgent.send(confirm);
59             log.log("10-Inviata_la_conferma.", confirm);
60             log.log("11-Attendo_il_messaggio_con_l'ora_da" + selectedProvider);
61             sleep(2000);
62             log.log("12-Controllo_se_arrivato_il_messaggio_con_l'ora_da" + selectedProvider);
63             if(myAgent.isThereMessage(selectedProvider, Performative.INFORM)) {
64                 log.log("13-Leggo_il_messaggio_con_l'ora_da" + selectedProvider);
65                 Message inform = myAgent.receive(selectedProvider, Performative.INFORM);
66                 log.log("14-Letto_il_messaggio_con_l'ora_da" + selectedProvider, inform);
67                 System.out.println(inform);
68             } else {
69                 log.log("15-Nessun_messaggio_da" + selectedProvider);
70                 System.out.println("Nessuna_risposta,rinuncio.");
71             }

```

```

72         done();
73     }
74 } catch (JAMADSLEException err1) {
75     System.out.println("Errore:␣" + err1);
76     done();
77 } catch (JAMIOException err2) {
78     System.out.println("Errore:␣" + err2);
79     done();
80 }
81 }
82 }

```

La creazione dell'agente e l'inserimento del comportamento è realizzato nel *main* della seguente classe che definisce anche l'agente stesso.

```

1  import JAM.*;
2
3  public class AskTimeAgent extends JAMAgent {
4      public AskTimeAgent(String category, String name)
5          throws JAMADSLEException {
6          super(new PersonalAgentID(category, name));
7      }
8      public static void main(String[] args) throws JAMADSLEException {
9          AskTimeAgent asktimeagent =
10             new AskTimeAgent("Matteo", "Baldoni");
11          asktimeagent.addBehaviour(new AskTimeBehaviour(asktimeagent));
12          AskTimeAgent asktimeagent1 =
13             new AskTimeAgent("Mario", "Rossi");
14          asktimeagent1.addBehaviour(new AskTimeBehaviourRefuse(asktimeagent1));
15          AskTimeAgent asktimeagent2 =
16             new AskTimeAgent("Mercoledì", "Adams");
17          asktimeagent2.addBehaviour(
18             new AskTimeBehaviourNotUnderstood(asktimeagent2));
19          asktimeagent.init();
20          asktimeagent.start();
21          asktimeagent1.init();
22          asktimeagent1.start();
23          asktimeagent2.init();
24          asktimeagent2.start();
25      }
26 }

```

Il codice di sopra riporta di altri due agenti che presentano un comportamento non corretto, *AskTimeBehaviourRefuse* invia inizialmente una INFORM anziché una REQUEST, mentre *AskTimeBehaviourNotUnderstood* invia correttamente una REQUEST ma con un *content* errato. Il codice è disponibile presso il supporto on-line al corso.

Gli oggetti *log* di tipo *LogFile* sono utilizzati per tracciare su di un file quanto avviene per scopi di debug. Anche il codice di questa classe è disponibile presso il supporto on-line al corso.

3 Descrizione dettagliata delle parti da svolgere

Nelle seguenti sezioni verranno presentate le varie parti del laboratorio da svolgere. È importante notare che i diagrammi di classe presentati nelle figure 6, 7, 10, 18, 20 e 4 sono puramente indicativi e potrebbe essere necessario introdurre ulteriori metodi. Inoltre tali diagrammi appositamente non riportano la visibilità dei metodi e dei campi come non riportano la firma completa dei metodi o il tipo dei campi. I diagrammi che dovranno essere presentati all'esame

dovranno invece essere completi di tali informazioni. I diagrammi presenti in questo documento sono stati realizzati utilizzando *Umbrello UML Modeller* [2], altri prodotti free o opensource sono, ad esempio, *Poseidon for UML* [3] e *ArgoUML* [8].

3.1 Parte I: l'interfaccia *AgentID*, le classi *GenericAgentID*, *CategoryAgentID*, *PersonalAgentID*, *Performative* e *Message*

Definire le classi *GenericAgentID*, *CategoryAgentID* e *PersonalAgentID*, la seconda estenda la prima, la terza estende la seconda e tutte implementano l'interfaccia *AgentID*. L'interfaccia *AgentID* richiede che vengano implementati i seguenti metodi:

- *boolean equals(Object agentID)*: confronta un oggetto della classe che implementa l'interfaccia *AgentID* con un oggetto di tipo *AgentID*. Restituisce *true* se l'identificatore dell'agente passato come parametro ha lo stesso nome e appartiene alla stessa categoria dell'oggetto su cui è invocato il metodo, *false* altrimenti;².
- *String getName()*: restituisce il valore corrente del campo *name* di tipo *String* che rappresenta il nome simbolico dell'agente (es. "orologio_atomico_GFerraris");
- *String getCategory()*: restituisce il valore corrente del campo *category* di tipo *String* che rappresenta la categoria a cui appartiene l'agente (es: "time_provider", "time_requester");
- *String toString()*: che restituisce un oggetto di tipo *String* che rappresenta l'oggetto stesso.

Le classi *CategoryAgentID*, *GenericAgentID* vengono utilizzate per individuare, rispettivamente, tutti gli agenti appartenenti ad una categoria e tutti gli agenti presenti in un dato momento nella piattaforma.

Definire poi un *tipo enumerativo Performative* che elenca le performative disponibili (UNKNOWN, REQUEST, INFORM, REFUSE, QUERY IF, QUERY_REF, AGREE, FAILURE, NOT_UNDERSTOOD, CALL_FOR_PROPOSAL, PROPOSAL, ecc.).³

Definire poi una classe *Message* contenente i seguenti campi:

- *sender*: di tipo *AgentID* che indica il mittente del messaggio;

² Attenzione! Cosa succede quando si confronta un oggetto di tipo *GenericAgentID* e *CategoryAgentID* con un oggetto di tipo *PersonalAgentID*? Ricordarsi di trattare correttamente questi casi.

³ Informazioni sul tipo enumerativo possono essere trovate sul libro di testo [4, pag. 51-, pag. 213-]. Un'altra fonte utile è il tutorial di Java [7].

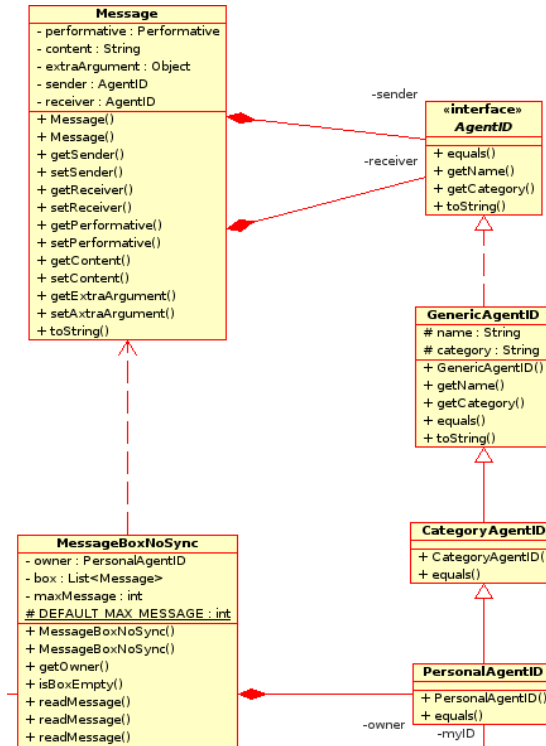


Figure 6: Diagramma delle classi della prima parte del progetto.

- *receiver*: di tipo “scegliere” che indica il destinatario del messaggio, il destinatario può essere un agente specifico, una categoria o tutti gli agenti presenti nella piattaforma in quel dato momento;
- *performative*: di tipo *Performative* che specifica la performativa utilizzata;
- *content*: di tipo *String* che memorizza il contenuto del messaggio;
- *extraArgument*: di tipo *Object* che permette di memorizzare eventuali ulteriori informazioni in forma di oggetto.

Per ogni classe si definiscano gli opportuni costruttori. Si definiscano poi i necessari metodi di tipo “*get*” ed uno di tipo “*set*”, per i campi presenti nelle vari classi che ne necessitino. Ad esempio, per il campo *sender* della classe *Message*:

- *public AgentID getSender()*: restituisce l’oggetto di tipo *AgentID* rappresentante il mittente del messaggio contenuto nel campo *sender*;
- *public void setSender(AgentID sender)*: inizializza il campo *sender* con il valore contenuto nel parametro *sender*.

Si definisca ancora un metodo

```
public String toString()
```

che restituisce una stringa che rappresenta l'oggetto di tipo *Message* in formato stampabile su console. Ad esempio, se *messaggio* è una variabile che contiene un riferimento ad un oggetto di tipo *Message*, allora *messaggio.toString()* restituisce una stringa che stampata dovrebbe risultare qualcosa del tipo:

Performativa: REQUEST

Sender: (orologio_atomico_GFerraris, time_provider)

Receiver: (Aldo, time_requester)

Content:

Che ora e'?

ExtraArgument:

21:35

Si ricordi di scegliere (sapendo giustificare il motivo) il tipo di visibilità (*private?* *protected?* *public?* *default?*) per ogni campo, metodo e classe.

3.1.1 Test

La prima parte del progetto deve eseguire correttamente (*senza apportare alcuna modifica!*) il test presente nel file *TestParteI.java* presente nel supporto on-line al corso per l'a.a. 2009/2010, con risultati analoghi a quelli descritti dal file *TestParteI.log*. Si commentino i risultati ottenuti.

3.2 Parte II: la classe *MessageBoxNoSync*

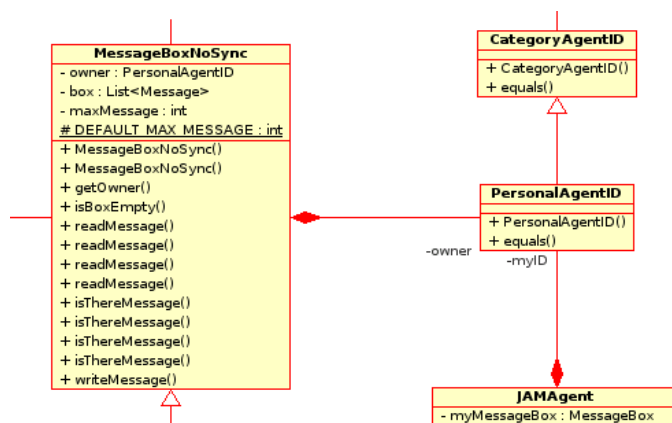


Figure 7: Diagramma delle classi della seconda parte del progetto.

Si definisca una classe *MessageBoxNoSync* come in Figura 7 che contenga i campi:

- *owner*: di tipo *PersonalAgentID* che specifica il proprietario della casella postale (si noti che il proprietario può essere solo un agente concreto);
- *box*: di tipo *java.util.List<E>* (si scelga l'implementazione preferita, *ArrayList<E>*, *LinkedList<E>*, *Vector<E>* ma il campo *box* deve essere di tipo *List<E>*) che rappresenta la *coda* dei messaggi ricevuti dall'utente proprietario della casella;
- *maxMessaggi*: di tipo intero (*positivo, maggiore di zero*), che indica il numero massimo di messaggi che possono essere inseriti nella lista *box*, tale valore è inizializzato e fissato durante la creazione dell'oggetto di tipo *MessageBox* e non può essere successivamente modificato. È possibile definire un valore di *default* nel caso questo non sia specificato nel costruttore.

ed i metodi:

- “*set*” e “*get*” per i campi sopra introdotti che lo necessitano;
- *boolean isEmpty()*: restituisce *true* se *box* è vuota,
- *Message readMessage(...)*: restituisce, togliendolo dalla coda di messaggi, il primo oggetto di tipo *Message* in coda su *box* che soddisfa i requisiti specificati dai suoi parametri. Se il messaggio non è presente nel *box* viene lanciata una eccezione di tipo *JAMMessageBoxException* (si veda anche la Sezione 3.8), in nessun caso deve essere restituito *null* come valore di ritorno.

I requisiti di possibile lettura da prevedersi sono:

- lettura del primo messaggio in coda (il più vecchio in attesa);
- lettura del primo messaggio in coda inviato da un certo agente;
- lettura del primo messaggio in coda inviato da un agente appartenente ad una certa categoria di agenti;
- lettura del primo messaggio in coda corrispondente ad una certa performativa;
- lettura del primo messaggio in coda inviato da un certo agente e corrispondente ad una certa performativa;
- lettura del primo messaggio in coda inviato da un agente appartenente ad una certa categoria di agenti e corrispondente ad una certa performativa.
- *boolean isThereMessage(...)*: restituisce **true** se è presente un messaggio in coda su *box* che soddisfa i requisiti specificati dai suoi parametri (gli stessi per *readMessage*), *false* altrimenti;
- *void writeMessage(Message message)*: inserisce in coda alla casella il messaggio passato come parametro.

3.2.1 Test

Si definisca una classe *ProvaMessageBoxNoSync* contenente un metodo *main* che effettui il test delle classi precedentemente definite.

Si provi a cambiare il tipo di implementazione per il campo *box* (ad esempio, *ArrayList* $\langle E \rangle$ anzichè *Vector* $\langle E \rangle$) e si verifichi che ancora tutto funzioni.

3.3 Parte III: gestire gli accessi remoti e sincronizzati in una *MessageBox*

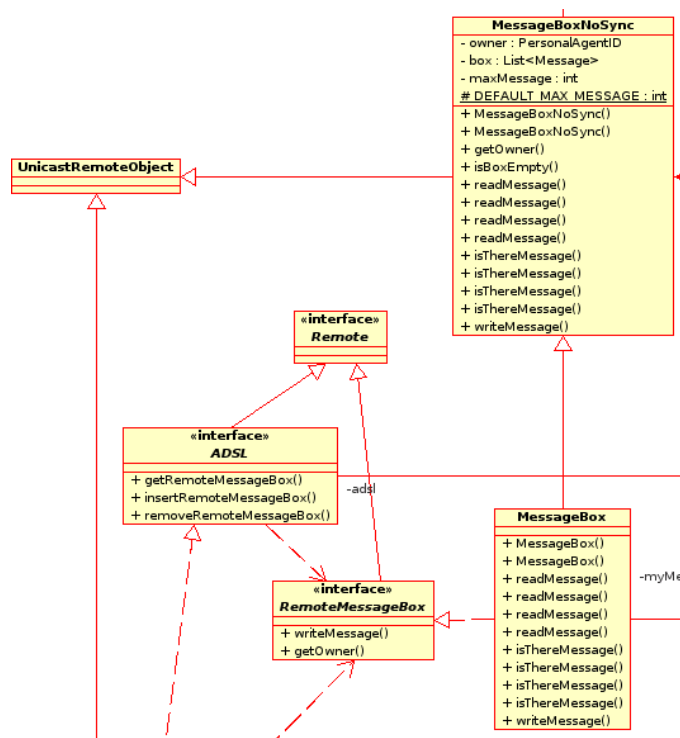


Figure 8: Diagramma delle classi della terza parte del progetto.

Si definisca una classe *MessageBox* come estensione della classe *MessageBoxNoSync* (quindi recuperando pi codice possibile) come in Figura 8 come estensione della classe *MessageBoxNoSync* in modo tale che:

- si garantisca l'accesso alla struttura dati *box* definita in *MessageBoxNoSync* in mutua esclusione tra diversi *thread*;
- un oggetto di tipo *MessageBox* sia “remoto” [5, Cap. 5, pag. 269].

Ogni agente della piattaforma **JAM** possiede una propria coda di messaggi *MessageBox*; in tale coda gli altri agenti gli possono far recapitare i messaggi

(si veda la Figura 9). La lettura dei messaggi avviene esclusivamente da parte

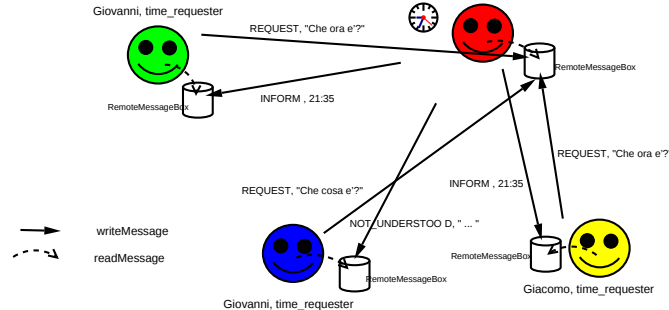


Figure 9: Scambio di messaggi e uso del *RemoteMessageBox*.

dell'agente proprietario della coda mentre la scrittura può avvenire da parte di un qualsiasi altro agente. La coda di messaggi è quindi una risorsa condivisa a cui è necessario garantire l'accesso in mutua esclusione ed accessibile remotamente (per la scrittura) in quanto i vari agenti possono risiedere su java virtual machine diverse.

3.3.1 L'interfaccia *RemoteMessageBox*

Al fine di permettere l'accesso in maniera remota in scrittura alla coda di messaggi di un agente della piattaforma si introduca l'interfaccia "remota" (cioè questa estende l'interfaccia *java.rmi.Remote*) contenente i metodi che è necessario rendere disponibili remotamente e che la classe *MessageBox* implementa (quali? *readMessage?* *writeMessage?* *isThereMessage?* *getOwner?*).

Si noti che, poichè in Java per una classe non è possibile estendere più di un'altra classe e la classe *MessageBox* già estende la classe *MessageBoxNoSync*, non è possibile per *MessageBox* estendere anche la classe *UnicastRemoteObject* come richiesto [5, Cap. 5, pag. 269]. Per questa ragione nella Figura 9 trovate che è la classe *MessageBoxNoSync* ad estendere *UnicastRemoteObject*.

3.3.2 Sincronizzare gli accessi alla coda di messaggi

Diversi sono gli aspetti da trattare. In primo luogo è necessario garantire l'accesso ad una coda di messaggi in mutua esclusione per le operazioni di lettura (*readMessage*) e scrittura (*writeMessage*) contemporanee (cioè se eseguite su thread diversi) alla stessa coda di messaggi.

In secondo luogo, si desidera che le operazioni di lettura di messaggi siano *bloccanti*. Questo significa che quando un agente effettua una operazione di lettura e non è presente alcun messaggio nella propria coda l'esecuzione si arresta per attendere che un altro agente ne recapiti uno. Il thread contenente l'invocazione della lettura deve quindi essere posto in *wait*, liberare la risorsa ed essere risvegliato quando qualcuno effettuerà una scrittura di un messaggio.

Analogamente, anche le operazioni di scrittura sono *bloccanti*, cioè se la coda ha raggiunto il suo massimo valore di messaggi che possono essere contenuti, il thread contenente la scrittura deve essere posto in *wait* finché un altro thread non effettuerà una lettura di un messaggio.

Si noti che esistono diversi tipi di letture di messaggi (*readMessage*), quella generica che legge il primo messaggio in coda e quella che legge dalla coda un messaggio con specifiche caratteristiche (*sender* e/o *tipo di messaggio*). Anche in questi casi la lettura è *bloccante*, ossia il thread contenente la lettura deve essere posto in *wait* finché un messaggio con le caratteristiche richieste non viene trovato indipendentemente dal fatto che nella coda siano presenti altri messaggi.

3.3.3 ConcurrentModificationException

Sebbene il metodo `isThereMessage` della classe `MessageBox` effettua una semplice lettura si potrebbe comunque andare incontro ad un problema se questo non è dichiarato *synchronized*. Il fatto è che la lettura viene effettuata su di una struttura dati che potrebbe essere modificata nel momento in cui viene interrogata. In questo caso verrebbe lanciata l'eccezione

```
java.util.ConcurrentModificationException
```

Se si desidera effettuare l'interrogazione del campo `box` nella maniera più indipendente possibile dalla sua modifica ma nello stesso tempo senza incorrere nell'eccezione suddetta, una soluzione potrebbe essere quella di effettuare una copia della struttura stessa. Ad esempio:

```
1 public boolean isThereMessage( ... ) {  
2     Message[] tempBox;  
3     synchronized(this) {  
4         tempBox = (java.util. ... ) box.toArray(tempBox);  
5     }  
6     // interrogazione di tempBox;  
7 }
```

In questo modo si limita la mutua esclusione alla sola copiatura della struttura. Il rovescio della medaglia è l'utilizzo dello spazio per la copia.

3.4 Parte IV: l'Agent Directory Service Layer

L'*ADSL* ha lo scopo di “pubblicare” i *RemoteMessageBox* degli agenti presenti nella piattaforma in un dato momento. Un agente, quando creato, si connette con una istanza di *ADSL* (tramite l'indirizzo al *rmiregistry* in cui il suo stub è registrato) e quindi iscrive la propria *RemoteMessageBox*. Questa azione è analoga alla registrazione di un oggetto presso un *rmiregistry* locale, solo che questo può essere fatto anche quando l'*ADSL* risiede in un'altra macchina (si veda anche i commenti nella Sez. 3.4.3). Quando un agente *A* vuole inviare un messaggio ad un agente *B* si rivolge all'*ADSL* specificando l'identificatore dell'agente *B*, l'*ADSL* restituisce il *RemoteMessageBox* di *B*, tramite questo riferimento *A* può invocare la *writeMessage*.

Si definisca una interfaccia “remota” *ADSL* contenente i metodi:

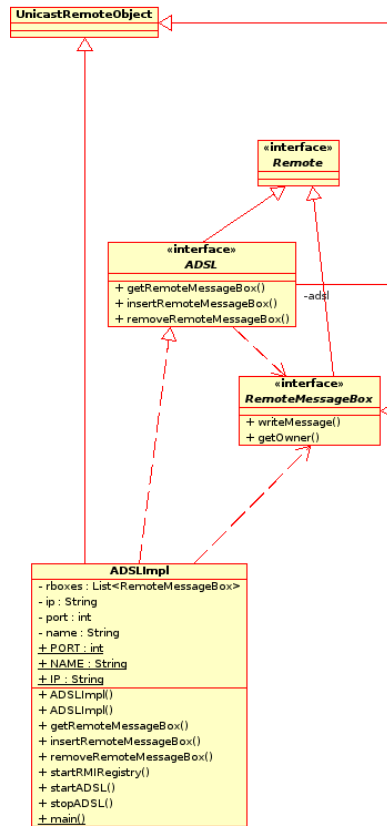


Figure 10: Diagramma delle classi della quarta parte del progetto.

- *List<RemoteMessageBox> getRemoteMessageBox(AgentID agentID) throw ...*, che restituisce una lista di riferimenti ad oggetti (remoti) di tipo *RemoteMessageBox* i cui proprietari sono uguali a *agentID*. Ovvero, nel caso in cui *agentID* contenga un riferimento ad un oggetto di tipo *PersonalAgentID* allora la lista sarà composta dal singolo riferimento al *RemoteMessageBox* dell'agente di ID specificato. Nel caso contenga un riferimento ad una istanza di *CategoryAgentID* allora la lista sarà composta da tutti i riferimenti ai *RemoteMessageBox* i cui proprietari hanno stessa categoria di quella specificata mediante il parametro *agentID*. Infine, nel caso che *agentID* contenga un riferimento ad una istanza di *GenericAgentID* allora la lista restituita contiene in pratica tutti i riferimenti ai *RemoteMessageBox* presenti in quel dato momento.⁴
- *void insertRemoteMessageBox(RemoteMessageBox messageBox) throw ...*,

⁴ Non si deve in alcuna parte del codice del laboratorio fare uso di *instanceof*.

che richiede l'inserimento di *messageBox* presso l'*ADSL*, se l'elemento è già presente non viene effettuata alcuna operazione e viene lanciata un'opportuna eccezione;

- *void removeRemoteMessageBox(AgentID agentID) throw ...*, che richiede la cancellazione dell'oggetto *RemoteMessageBox* presente presso l'*ADSL* di proprietà dell'agente *agentID*. Se l'elemento non è presente non viene effettuata alcuna operazione e viene lanciata un'opportuna eccezione.

Si definisca quindi la classe *ADSLImpl* che implementa l'interfaccia remota *ADSL* e contenente il campo:

- *messageBoxes*: contenente gli oggetti remoti di tipo *RemoteMessageBox*.

Si scelga l'implementazione più opportuna per *messageBoxes*.

3.4.1 *messageBoxes* è una risorsa condivisa

Si noti che *messageBoxes* è una risorsa condivisa, più agenti possono richiedere inserimenti/cancellazioni/interrogazioni contemporaneamente. Si realizzino le implementazioni dei metodi *get*, *insert* e *remove* in maniera opportuna.

3.4.2 Test

Si crei un oggetto di tipo *ADSLImpl* e si effettui il *bind* (o il *rebind*) dell'oggetto presso un *rmiregistry* attivo (si veda la Figura 11). Tutto questo lo si realizzi,

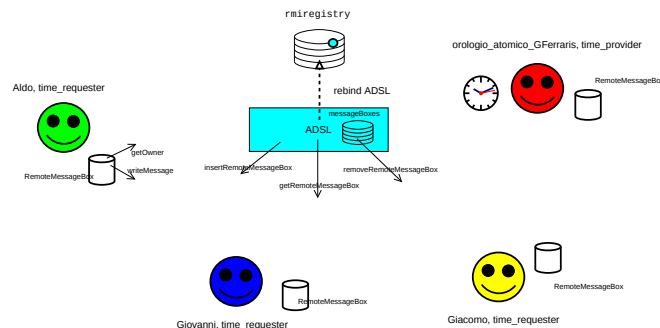


Figure 11: Creazione di un oggetto di tipo *ADSLImpl* e iscrizione presso l'*rmiregistry*.

ad esempio, mediante il *main* di una classe di nome *ProvaADSL* del tipo:

```
1 public class ProvaADSL {
2     public static void main(String[] args) {
3         ...
4         ADSLImpl adsl = new ADSLImpl();
```

⁴ Si noti che in questo modo è possibile eliminare anche caselle di altri agenti ma non è necessario preoccuparsi di tale aspetto.

```

5      try {
6          // la riga seguente e' in alternativa con l'attivazione
7          // a prompt del rmiregistry
8          java.rmi.registry.LocateRegistry.createRegistry(2000);
9          Naming.rebind("rmi://127.0.0.1:2000/ADSL", adsl);
10     }
11     catch (Exception e) {
12         System.err.println("Failed to bind to RMI Registry");
13         System.exit(1);
14     }
15     ...
16 }
17 }

```

A questo punto, in un'altra o più classi, ad esempio *ProvaRemoteMessageBoxUno*, *ProvaRemoteMessageBoxDue*, si effettui il *lookup* presso l'*rmiregistry* dell'oggetto *ADSL* creato in *ProvaADSL*, si veda la Figura 12, quindi si crei un

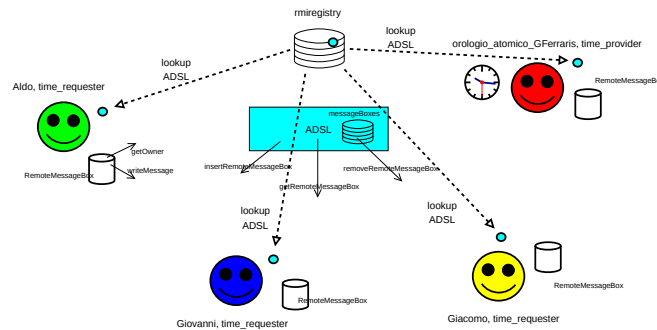


Figure 12: Lookup dell'oggetto di tipo *ADSL* presso *rmiregistry*.

oggetto di tipo *MessageBox* (che è anche un oggetto di tipo *RemoteMessageBox*) e lo si iscriva presso l'*ADSL* attraverso l'invocazione del metodo remoto *insertRemoteMessageBox*, si veda la Figura 13) Il codice potrebbe essere del

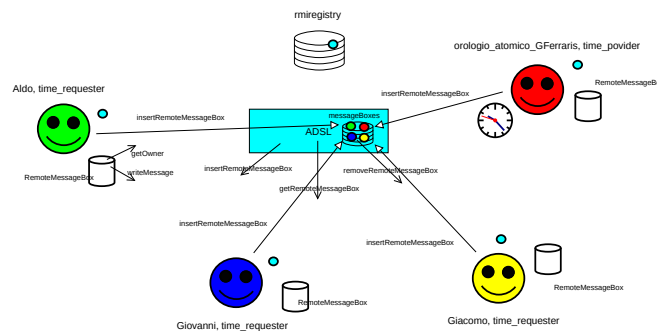


Figure 13: Iscrizione di un oggetto di tipo *RemoteMessageBox* presso l'*ADSL*.

tipo:

```

1 public class ProvaRemoteMessageBoxUno {
2     public static void main(String [] args) {
3         ...
4         ADSL adsl;
5         AgentID myID;
6         MessageBox box1;
7         try {
8             myID = new AgentID();
9             ...
10            adsl = (ADSL)Naming.lookup("rmi://127.0.0.1:2000/ADSL");
11            box1 = new MessageBox();
12            box1.setOwner(myID);
13            ...
14            adsl.insertRemoteMessageBox(box1);
15        }
16        catch(Exception e) {
17            System.out.println("Failed_rmi" + e);
18        }
19        ...
20    }
21 }

```

Ora è possibile da uno dei vari programmi richiedere mediante il metodo *getRemoteMessageBoxes* uno o più oggetti di tipo *RemoteMessageBox* disponibili remotamente (si veda la Figura 14), quindi invocare su di esso il metodo

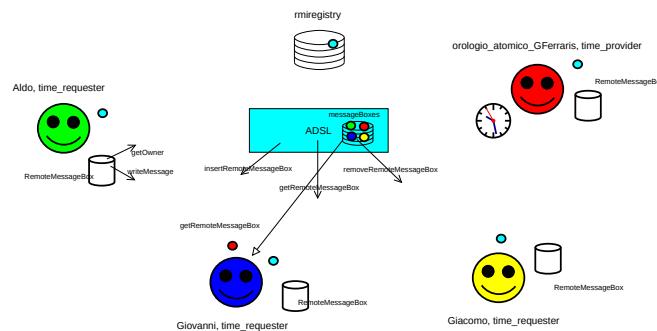


Figure 14: Richiesta di un *RemoteMessageBox* mediante l'*ADSL*.

writeMessage per inserire un oggetto di tipo *Message* sulla casella remota (si veda la Figura 15). Si legga quindi la propria casella per verificare se sono arrivati messaggi inviati da altri programmi (esempio, *ProvaRemoteMessageBoxDue*). Infine si cancelli dal *ADSL* l'oggetto *MessageBox* iscritto precedentemente (questo per simulare il momento che l'agente viene tolto dalla piattaforma), si veda la Figura 16.

Il codice dovrebbe assomigliare al seguente:

```

1 public class ProvaRemoteMessageBoxUno {
2     public static void main(String [] args) {
3         ...
4         ADSL adsl;
5         AgentID myID;
6         MessageBox box1;
7         try {
8             myID = new AgentID();
9             ...

```

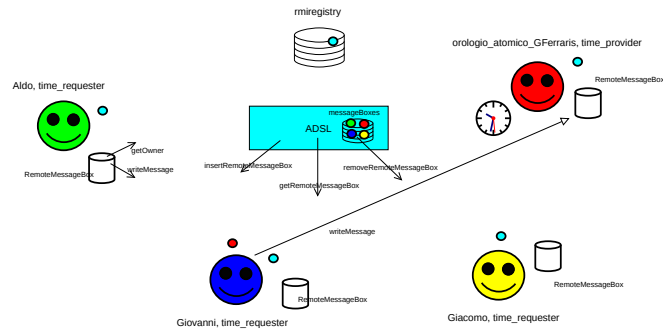


Figure 15: Scrivere un messaggio su un *MessageBox* remoto.

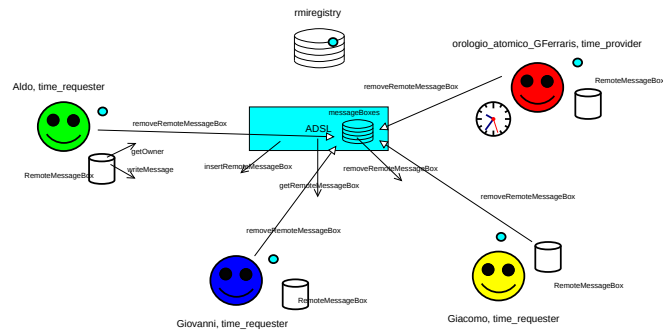


Figure 16: Rimozione della *MessageBox* dalla *ADSL*.

```

10  adsl = (ADSL)Naming.lookup("rmi://127.0.0.1:2000/ADSL");
11  box1 = new MessageBox();
12  box1.setOwner(myID);
13  ...
14  adsl.insertRemoteMessageBox(box1);
15  ...
16  Message msg = new Message();
17  ...
18  AgentID agentRemoteID = new AgentID();
19  ...
20  List<RemoteMessageBox> remoteBox2 = adsl.getRemoteMessageBox(agentRemoteID);
21  for (RemoteMessageBox rmb : remoteBox2)
22      rmb.writeMessage(msg);
23  ...
24  Message myMsg = box1.readMessage();
25  System.out.println(myMsg);
26  ...
27  adsl.removeRemoteMessageBox(myID);
28  ...
29  }
30  catch (Exception e) {
31      System.out.println("Failed_rmi" + e);
32  }
33  ...
34  }
35  }

```

A questo punto si proceda con la compilazione di tutti i file sorgenti

```
> javac *.java
```

la creazione dei file stub per `MessageBox` e `ADSLImpl` (non necessaria con Java2 5.0!)

```
> rmic -v1.2 MessageBox
```

```
> rmic -v1.2 ADSLImpl
```

E si proceda con l'esecuzione, si avvii lo *rmiregistry* (non necessario se si è inclusa l'istruzione *java.rmi.registry.LocateRegistry(2000)* prima di effettuare il binding) *ProvaADSL*

```
> rmiregistry 2000 &
```

```
> java ProvaADSL
```

(il comando "&" è per UNIX, per Windows si usino più finestre DOS diverse), entrambi i comandi dati attiveranno un processo che terminerà solo mediante un *CTRL-C* esplicito. Quindi si lancino da altri due finestre di comando i programmi *ProvaRemoteMessageBoxUno* e *ProvaMessageBoxDue*

```
> java ProvaRemoteMessageBoxUno
```

3.4.3 Osservazioni importanti

Lo scopo dell'*ADSL* è quello di evitare di effettuare presso ogni agente la *lookup* delle caselle di ogni altro agente presente nel sistema. La *lookup* sarà quindi utilizzata per il solo oggetto di tipo *ADSL*. Al fine di ottenere un corretto funzionamento della piattaforma è comunque il fatto che gli oggetti *MessageBox* siano dichiarati anche remoti e iscritti presso un *rmiregistry*. Infatti, se gli oggetti *MessageBox* non fossero remoti, l'effetto di passarli come parametro del metodo *insertRemoteMessageBox* o come valore di ritorno del metodo *getRemoteMessageBoxes* creerebbe una copia presso l'*ADSL* anziché un riferimento all'oggetto stesso e quindi non si avrebbe l'effetto desiderato (si veda anche [5, pag. 285-, pag. 295-]). Si osservi come l'oggetto di tipo *ADSL* funga da vero e proprio *rmiregistry*. Infatti non è necessario effettuare alcuna *rebind* o *lookup* degli oggetti di tipo *MessageBox* per dividerne i riferimenti remoti (ma questi comunque debbono essere dichiarati remoti per poter passare il solo riferimento remoto come parametro o valore restituito della *insertRemoteMessageBox* e *getRemoteMessageBox*, rispettivamente, e non come copia).

3.4.4 Un'interfaccia grafica per l'ADSL

Si realizzi un'interfaccia grafica del tipo mostrato in Figura 17 che permette di gestire l'ADSL e visualizzare i messaggi di *log*. In particolare, si modifichi la classe *ADSLImpl* aggiungendo i campi *port* (di tipo *int*) e *name* (di tipo *String*) che contengono rispettivamente la porta in cui viene avviato lo *rmiregistry* in cui l'oggetto di tipo *ADSLImpl* viene iscritto. Si introducano poi i tre metodi seguenti:

- *void startRMIRegistry()*, che avvia un processo *rmiregistry* sulla porta *port* della macchina su cui l'applicativo è eseguito;

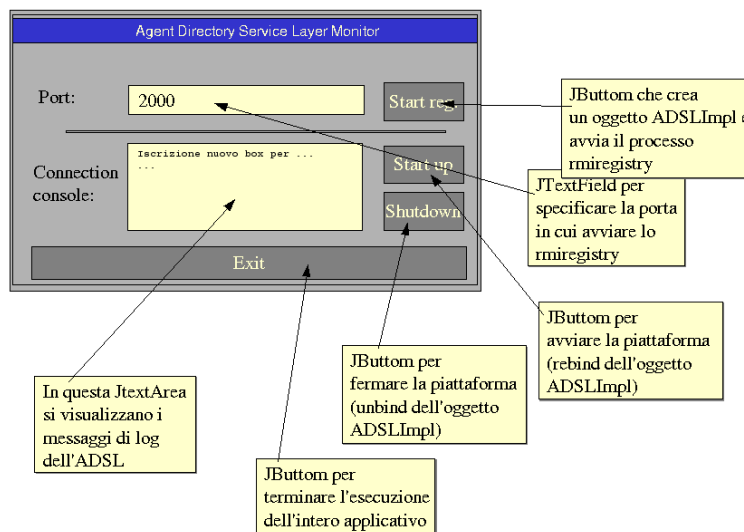


Figure 17: L'ADSL monitor.

- *void startADSL()*, che iscrive (*rebind*) l'oggetto *ADSLImpl* corrente sullo *rmiregistry* con nome *ADSL*;
- *void stopADSL()*, che cancella (*unbind*) l'oggetto *ADSLImpl* corrente dallo *rmiregistry*.

I bottoni “Start reg.”, “Start up” e “Shutdown” quando premuti richiedono l'esecuzione del corrispondente metodo sopra descritto.

ADSLMonitor* osservatore di *ADSLImpl

La seguente parte è facoltativa per la prima sessione d'esame

La classe *ADSLMonitor* che realizza l'interfaccia grafica dell'*ADSL* deve visualizzare una stringa di *log* per ogni operazione effettuata e visualizzare questa presso un opportuno *JTextArea* (si veda la Figura 17). Per realizzare questo si faccia in modo che *ADSLMonitor* risulti un osservatore dell'oggetto *ADSLImpl* creato con la pressione del tasto “Start reg.” e quindi si modifichi ulteriormente la classe *ADSLImpl* in modo da rendere gli oggetti creati *osservabili*. In particolare, devono essere osservate ogni richiesta di esecuzione dei metodi *getRemoteMessageBoxes*, *insertRemoteMessageBox* e *removeRemoteMessageBox*. Queste richieste di esecuzione devono far comparire un'opportuno messaggio sulla “Connection console”, ad esempio:

```
Iscrizione nuovo box per (Aldo, time_requester)
Iscrizione nuovo box per (Giovanni, time_requester)
Iscrizione nuovo box per (orologio_atomico_GFerraris, time_provider)
Richiesto box (Corologio_atomico_GFerraris, time_provider)
```

Richiesto box (Aldo, time_requester)
 Cancellato box (Giovanni, time_requester)

Si noti che poichè la classe *ADSLImpl* estende già la classe *UnicastRemoteObject* non è possibile che questa estenda anche la classe *Observable*, è necessario quindi realizzare in proprio le funzionalità messe a disposizione da questa.

3.5 Parte V: la classe *JAMAgent*

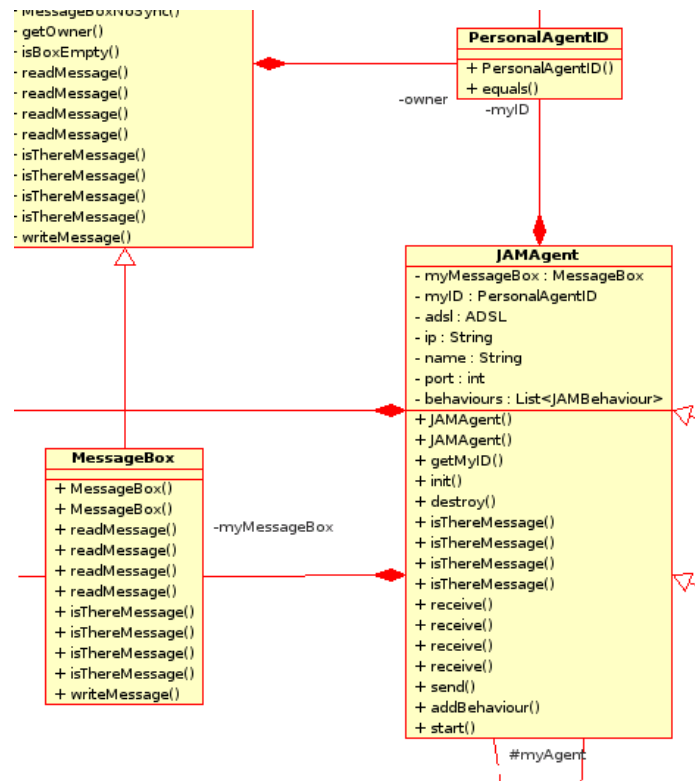


Figure 18: Diagramma delle classi della quinta parte del progetto.

In questa parte del laboratorio incominceremo a realizzare la classe *JAMAgent* utilizzata nell'esempio dell'asta per definire la classe *Banditore* e *ClientAgent*.

Si definisca la classe *JAMAgent* contenente i campi:

- *myMessageBox*: di tipo *MessageBox*;
- *myID*: di tipo *PersonalAgentID*;
- *adsl*: di tipo *ADSL*;

- *name*: di tipo *String*, contenente l'indirizzo il nome dell'ADSL nello *rmiregistry*;
- *ip*: di tipo *String*, contenente l'indirizzo IP dello *rmiregistry*;
- *port*: di tipo *int*, contiene il numero della porta su cui è disponibile lo *rmiregistry* all'indirizzo *ip*.

Si scelgano gli opportuni qualificatori di visibilità e si definiscano, se lo si ritiene necessario, i metodi *get* e *set*. Si definiscano inoltre i seguenti metodi (scegliendo l'opportuno qualificatore di visibilità):

- *void init()*: effettua la *lookup* presso lo *rmiregistry* all'indirizzo *ip/port* dell'oggetto *ADSL* di nome *ADSL* e memorizza tale riferimento remoto in *adsl*, quindi, se tutto è andato bene, iscrive presso l'*ADSL* l'oggetto di tipo *MessageBox myMessageBox*.
- *void destroy()*: effettua la rimozione della casella *myMessageBox* dall'*ADSL*;
- *boolean isThereMessage(...)*: restituisce *true* se vi sono messaggi in *myMessageBox*, *false* altrimenti;
- *void send(...)*: richiede all'*ADSL* mediante il metodo *getRemoteMessageBoxes* gli oggetti di tipo *RemoteMessageBox* il cui proprietario è specificato dal receiver del messaggio *message*, su tali oggetti invoca la *writeMessage* di *message*;
- *Message receive(...)*: legge dalla propria casella *myMessageBox* un messaggio mediante il metodo *readMessage()* con le caratteristiche specificate dai parametri (nessuna, tipo messaggio, sender, categoria, sender e tipo, categoria e tipo).

Si definiscano quindi i seguenti costruttori:

- *public JAMAgent(AgentID agentID, String ip, String name, int port)*: dove *agentID* è il nome dell'agente e il suo ruolo, *ip* contiene l'indirizzo IP dello *rmiregistry* e *port* la porta dello *rmiregistry*, *name* il nome dell'ADSL presso il registry. Il costruttore inizializza gli opportuni campi creando l'oggetto *myMessageBox*;
- *public JAMAgent(AgentID agentID)*: come sopra ma si assume per l'IP il valore 127.0.0.1 e 1099 per la porta;

3.6 Parte VI: un'interfaccia per monitorare l'agente

Si realizzi un'interfaccia grafica del tipo mostrato in Figura 19 che permette di monitorare l'attività di un agente visualizzando i messaggi inseriti nella *MessageBox* locale e quelli successivamente letti. La classe *JAMAgentMonitor* contiene un campo *agent* inizializzato mediante costruttore il riferimento all'oggetto *JAMAgent* di cui la corrente istanza di interfaccia grafica è monitor. Si introducano poi i metodi seguenti:

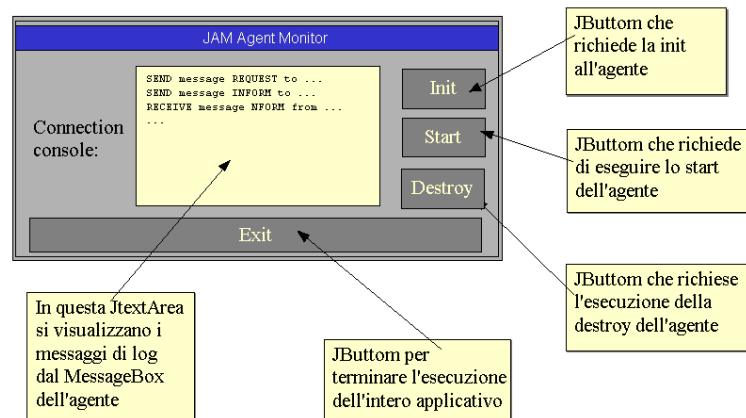


Figure 19: Lo JAMAgent monitor.

- *void initJAMAgent()*, che invoca il metodo *init* sull'oggetto *agent*;
- *void startJAMAgent()*, che invoca il metodo *start* (che verrà introdotto nella Sezione 3.7) sull'oggetto *agent*.
- *void destroyJAMAgent()*, che invoca il metodo *destroy* sull'oggetto *agent*;

I bottoni “Init”, “Start” e “Destroy” quando premuti richiedono l'esecuzione del corrispondente metodo sopra descritto.

Ad esempio, per avviare l'interfaccia grafica sull'agente banditore del nostro esempi iniziale si potrà procedere come segue:

```

1 class TimeProvider extend JAMAgent {
2     [...]
3     public TimeProvider (...) {
4         addBehaviour(new TimeProviderCalcola(this));
5         addBehaviour(new TimeProviderRispondi(this));
6         JAMAgentMonitor monitorTimeProvider =
7             new JAMAgentMonitor(this);
8     }
9     [...]
10 }

```

Per effettuare un test si può per ora utilizzare un agente “generico” *JAMAgent* (che presto dovrà però diventare una classe astratta!).

JAMAgentMonitor* osservatore di *JAMAgent

La classe *JAMAgentMonitor* che realizza l'interfaccia grafica per le classi che estenderanno *JAMAgent* deve visualizzare una stringa di *log* per ogni operazione effettuata e che si desidera monitorare presso una opportuna *JTextArea* (devono essere monitorate almeno le esecuzioni delle *send* e delle *receive*), si veda la Figura 19.

Per realizzare questo si faccia in modo che *JAMAgentMonitor* risulti un osservatore dell'oggetto di tipo *JAMAgent* contenuto nel campo *agent* di *JAMAgentMonitor* passato come parametro e quindi si modifichi la classe *JAMAgent* in modo da rendere gli oggetti creati *osservabili*. In particolare, devono essere osservate almeno ogni richiesta di esecuzione dei metodi *send* e *receive* (ogni *receive*!). Queste richieste di esecuzione devono far comparire un'opportuno messaggio sulla “*Connection console*”, ad esempio:

```
SEND message REQUEST to ...
SEND message INFORM to ...
RECEIVE message INFORM from ...
```

Per realizzare questo l'interfaccia *JAMAgentMonitor* deve implementare la classe *Observer* e quindi registrare le proprie istanze come osservatori dell'oggetto *agent*, un semplice modo è farlo nel costruttore stesso di *JAMAgentMonitor*:

```
1 class JAMAgentMonitor extend ... implements Observer {
2     [...]
3     private JAMAgent agent;
4     [...]
5     public JAMAgentMonitor(JAMAgent ag) {
6         [...]
7         agent = ag;
8         agent.addObserver(this);
9     }
10    [...]
11 }
```

e quindi la classe *JAMAgentMonitor* dovrà implementare il metodo

public void update(Observable ob, Object extra_arg)

specificato nell'interfaccia *Observer*. Il metodo *update* di occuperà di aggiornare la *Connection console* dove la stringa di *log* sarà ricevuta attraverso *extra_arg*.

La classe *JAMAgent* deve essere modificata in modo da estendere la classe *Observable*. Quindi ogni metodo che si desidera monitorare nella *Console connection* dovrà essere modificato con l'aggiunta delle invocazioni dei metodi *setChanged* e *notifyObservers*

```
1 class JAMAgent extend Observable {
2     [...]
3     public void send(Message message) {
4         [...]
5         String log = ...;
6         setChanged();
7         notifyObservers(log);
8     }
9     [...]
10 }
```

3.7 Parte VII: definire il comportamento i un agente

Obiettivo di questa parte del corso è quella di definire le classi che permettono di specificare il/i compartimento/i di un agente. Ogni comportamento di un agente è eseguito all'interno di un thread ad esso dedicato. In JAM abbiamo due tipi di comportamento: uno “*ciclico*” ed uno “*semplice*”. Il comportamento ciclico consiste nell'eseguire il codice specificato in un determinato metodo (*action*) in modo ripetitivo finchè non si ritiene il compito completato (specificato

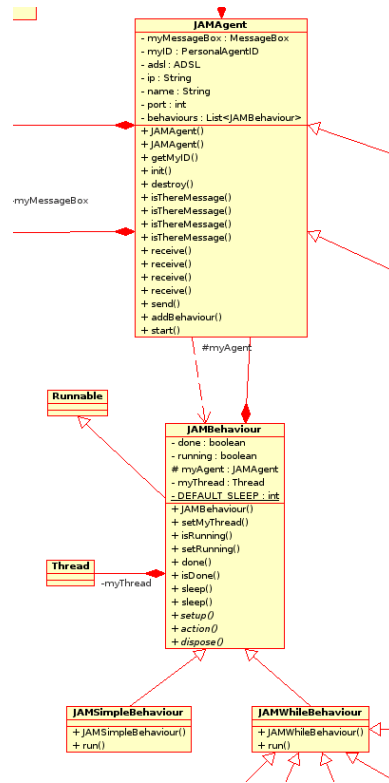


Figure 20: Diagramma delle classi della settima parte del progetto.

invocando un altro metodo, *done*). Il comportamento semplice consiste invece nell'eseguire il codice specificato in un determinato metodo (*action*) una volta sola.

Si definisca la classe *astratta* *JAMBehaviour* in modo che implementi l'interfaccia *Runnable*) e contenga i seguenti campi:

- *done*, di tipo *boolean*, che indica se il comportamento è stato eseguito completamente;
- *running*, di tipo *boolean*, che indica se il comportamento è in esecuzione;
- *myThread*, di tipo *Thread*, che il riferimento all'effettivo thread che esegue il comportamento;
- *myAgent*, di tipo *JAMAgent*, che indica l'agente possessore di tale comportamento.

Si definiscano, inoltre, i seguenti metodi:

- *void done()*: che imposta la variabile booleana *done* a *true*;

- *boolean isDone()*: che restituisce il valore corrente della variabile booleana *done*;
- *boolean isRunning()*: che restituisce il valore corrente della variabile booleana *running*;
- *void setRunning(boolean running)*: che imposta il valore della variabile booleana *running*;
- *void setMyThread(Thread myThread)*: che imposta il valore della variabile *myThread*;
- *void sleep(int ms)*: che invoca la *Thread.sleep* sul thread corrente per *ms* millisecondi;
- *void action() throws JAMBehaviourInterruptedException*: metodo astratto che definisce il codice da eseguire una o più volte.
- *void setup() throws JAMBehaviourInterruptedException*: metodo astratto che definisce il codice da eseguire prima di lanciare il metodo *action*.
- *void dispose() throws JAMBehaviourInterruptedException*: metodo astratto che definisce il codice da eseguire prima di terminare l'esecuzione del comportamento.

Il costruttore della classe *JAMBehavior* inizializza il campo *myAgent* (prendendo l'opportuno valore da un parametro), imposterà *done* a *false*.

Si definisca ora la classe astratta *JAMWhileBehaviour*. Questa classe estende la classe *JAMBehaviour* e ridefinisce il metodo *run* di *Thread* (si veda anche la Figura 20). Poichè in questo caso il comportamento specificato è di tipo ciclico il contenuto di *action* deve essere eseguito sino a quando non si ritiene il comportamento completato. Il metodo *run* è qualcosa del tipo:

```

1  public void run() {
2      try {
3          setup();
4          while (!isDone())
5              action();
6      } catch (JAMBehaviourInterruptedException err) {
7          if (isDone()) return;
8          System.out.println(err);
9      } finally {
10         try {
11             dispose();
12         } catch (JAMBehaviourInterruptedException err) {
13             System.out.println(err);
14         }
15     }
16 }
```

Il metodo *action* può contenere delle invocazioni dei metodi *receive* e *send* di *JAMAgent*. Questi metodi possono far sì che il thread sia messo in stato di *wait* in attesa di un messaggio o che la risorsa *MessageBox* si liberi. Se si invoca il metodo *done* con l'intenzione di far terminare l'esecuzione del thread (ad esempio, in seguito alla pressione del bottone *destroy*) in questi casi non ha

alcun effetto, in quanto il thread non sta eseguendo alcun codice e quindi non può accorgersi se è stata richiesta la terminazione. A questo scopo si modifichi il metodo *done* appena definito in modo che oltre ad impostare il valore della variabile *done* a *true* invochi il metodo *interrupt* della classe *Thread* sul thread che esegue tale comportamento

```

1 public void done() {
2     [...]
3     ... .interrupt();
4 }

```

in modo da permette di far uscire dallo stato di wait il thread corrente. La necessità di catturare l'eccezione di tipo *JAMBehaviourInterruptedException* è data dalla presenza di tale *interrupt* che fa uscire dalla wait il thread facendo lanciare alla wait dell'eccezione *InterruptedException* che viene catturata nei metodi *send/receive* e rilanciata in forma di *JAMBehaviourInterruptedException*. È importante non catturare l'eccezione *InterruptedException* e lasciarla emergere sino al metodo *run* (eventualmente incapsulata di un'altra eccezione come *JAMBehaviourInterruptedException*) poichè l'obiettivo di tale eccezione quando lanciata dall'interno del metodo *done* è quello di interrompere il metodo *run*.

Si noti che nella piattaforma Java 2 i metodi *stop*, *suspend* e *resume* sono stati dichiarati *deprecated*. Lo schema della classe *JAMWhileBehaviour* è ispirata a quanto mostrato in [5, pag. 46-] e presenta una soluzione per permettere ad un thread di interrompere un altro thread senza ricorso al metodo *stop*.

Si definisca ora la classe astratta *JAMSimpleBehaviour* che ridefinisce il metodo *run* di *Thread* in modo che questo invochi i metodi *setup*, *action* una sola volta, *dispose* e quindi il metodo *done*. Si noti che anche qui è necessario includere le due chiamate all'interno del costrutto *try ... catch* come per il caso della *run* di *JAMWhileBehaviour*.

Modifiche da apportare alla classe *JAMAgent*

Si modifichi la classe *JAMAgent* in modo da accogliere la definizione di comportamenti. Prima di tutto si definisca la classe *JAMAgent* astratta:

```
public abstract class JAMAgent ... ..
```

Quindi si aggiunga ai suoi campi il campo *myBehaviours* di tipo *java.util.List* (si scelga un'implementazione) e si introducano i seguenti metodi:

- *void addBehaviour(JAMBehaviour behaviour)*, che aggiunge il comportamento *behaviour* alla lista *myBehaviours*;
- *void start()*, che crea un oggetto di tipo *thread* per ogni comportamento di tipo *JAMBehaviour* non già in esecuzione presente in *myBehaviours* e invoca su di esso il metodo *start* (si ricordi di completare l'interfaccia grafica della Sezione 3.6 in modo che venga eseguito questo metodo in corrispondenza della pressione del tasto "Start").

Si modifichi, infine, il metodo *destroy* in modo che oltre ad effettuare i compiti già descritti nella Sezione 3.5, invochi anche il metodo *done* su ogni comportamento presente in *myBehaviour* in esecuzione.

3.7.1 Package

Tutte le classi create sino a questo punto devono essere incluse in un package di nome *JAM* e quindi rese disponibili mediante un file di tipo *jar*. Si faccia riferimento ai lucidi [1] presentati durante il laboratorio e disponibile presso il supporto on-line al corso.

3.7.2 Test Finale I

La prima parte del progetto deve eseguire correttamente (*senza apportare alcuna modifica!*) il test presentato nella Sez. 2.3 e disponibile nella cartella *Test Finale I* presente nel supporto on-line al corso per l'a.a. 2008/2009, con risultati analoghi a quelli descritti nei file contenuti nella cartella *Test Finale I Log File*. Si commentino i risultati ottenuti.

3.7.3 Test Finale II

Si realizzi uno a scelta dei due seguenti esempi.

Let us share our files! Si realizzi un sistema ad agenti che permetta di condividere e scambiare file appartenenti ad un certo direttorio. Il sistema dovrà contenere agenti di diverso tre diverso tipo:

- *agenti fornitori*: sono agenti che quando eseguiti in un certo direttorio eseguono la scansione dei file presenti nel direttorio stesso e li offrono a chi ne fa richiesta;
- *agenti ricercatori*: sono agenti che effettuano la ricerca di un insieme di file specificati inizialmente dall'utente;
- *agenti fornitori e ricercatori*: sono agenti che esibiscono entrambi i comportamenti delle precedenti due categorie.

Il comportamento di un agente fornitore può essere immaginato come un ciclo che analizza le richieste che pervengono. Queste possono essere di due tipi:

- **QUERY_IF** con content il nome del file cercato. Se l'agente dispone del file cercato risponde con un messaggio del tipo **INFORM** e content i dati del file. Se l'agente non dispone del file risponde con un messaggio di tipo **REFUSE** e le informazioni inviate dal richiedente;
- **REQUEST** con content il nome del file desiderato. Se l'agente dispone del file cercato risponde con un messaggio del tipo **INFORM** e content i dati del file e come argomento supplementare il file richiesto. Se l'agente non dispone del file risponde con un messaggio di tipo **REFUSE** e le informazioni inviate dal richiedente.

Negli altri casi l'agente fornitore risponde con un REFUSE o NOT_UNDERSTOOD a seconda dei casi (si veda anche quanto fatto nell'esempio "Che ora è?").

```

1 import JAM.*;
2
3 public class FileSharingFornitoreBehaviour extends JAMWhileBehaviour {
4     [...]
5     public void setup() throws JAMBehaviourInterruptedException {
6         // scansiono il direttorio corrente e
7         // creo una lista dei file disponibili,
8         // il nome, la dimensione, la data, ecc.;
9     }
10    public void dispose() throws JAMBehaviourInterruptedException { }
11    public void action() throws JAMBehaviourInterruptedException {
12        // leggo un messaggio e lo analizzo
13        // rispondo a seconda del messaggio letto
14    }
15 }

```

Un agente ricercatore analizza una lista di file da cercare e invia per ognuno di essi, una alla volta, una richiesta a tutti gli agenti di tipo fornitore con una QUERY_IF e le informazioni del file cercato (il nome). Dopo aver atteso qualche secondo legge le risposte ricevute dagli agenti fornitori. Se tra queste vi è più di un agente che ha tale file, ne sceglie uno in base a un criterio definito ed effettua la richiesta. Se nessuno dispone del file, passa al successivo file (si decida se si desidera tornarci successivamente).

```

1 import JAM.*;
2
3 public class FileSharingRicercaBehaviour extends JAMWhileBehaviour {
4     [...]
5     public void setup() throws JAMBehaviourInterruptedException {
6         // preparo la lista di file da cercare, leggendo da
7         // file o chiedendola in input all'utente
8     }
9     public void dispose() throws JAMBehaviourInterruptedException {
10        // stampo le statistiche di ricerca (file trovati, file non
11        // trovati, ecc.)
12    }
13    public void action() throws JAMBehaviourInterruptedException {
14        // leggo un file da cercare dalla mia lista
15        // invio una QUERY_IF a tutti i fornitori
16        // attendo qualche secondo
17        // leggo le risposte ricevute
18        // scelgo tra le risposte positive un agente
19        // invio a tale agente una REQUEST del file
20        // leggo la risposta e salvo il file sul direttorio
21        // se non ho altre ricerche da effettuare ho completato
22        // il mio lavoro
23    }
24 }

```

Per scandire un direttorio si faccia riferimento alla classe *File*. Per caricare e salvare un file ci si può ispirare al seguente codice:

```

1 public class FileUtility {
2     public static void loadFile(String nomeFile) {
3         [...]
4         byte[] data;
5         InputStream in = new FileInputStream(nomeFile);
6         int bytesAvailable = in.available();
7         if (bytesAvailable > 0) {
8             data = new byte[bytesAvailable];
9             in.read(data);

```

```

10     }
11     [...]
12 }
13 public static void saveFile(String nomeFile, byte[] data) {
14     [...]
15     OutputStream out = new FileOutputStream(nomeFile);
16     out.write(data);
17     [...]
18 }
19 [...]
20 }

```

Asta On Line Si realizzi un sistema ad agenti che simuli un'asta. In tale sistema alcuni clienti si contengono un oggetto effettuando le proprie offerte ad un agente banditore, il cliente che alla fine avrà effettuato l'offerta più alta si aggiudica l'oggetto in asta. Il sistema ad agenti dovrà contenere due tipi di agenti:

- *banditori*: che valutano le offerte ricevute e rispondono se l'offerta è accettata o respinta perchè superata da altra offerta;
- *cliente*: che dopo aver valutato l'offerta corrente e le proprie disponibilità effettuano un rilancio per conquistare l'oggetto in palio.

Un agente di tipo *banditore* quindi legge i vari messaggi ricevuti, questi possono essere di due tipi:

- QUERY_IF con content "Valore corrente?". Il banditore risponde a questo con REFUSE se l'oggetto è già stato aggiudicato, con INFORM e con content il valore della migliore offerta sino a quel momento e il nome dell'offerente se l'oggetto non è stato ancora aggiudicato;
- REQUEST con content l'offerta. Il banditore verifica che questa sia superiore a quella migliore registrata sino a quel momento, se è così aggiorna i dati della migliore offerta e comunica mediante una INFORM che questa è stata accettata. Se invece l'offerta ricevuta non è superiore alla migliore offerta corrente risponde con un messaggio di tipo REFUSE.

Negli altri casi l'agente fornitore risponde con un REFUSE o NOT_UNDERSTOOD a seconda dei casi (si veda anche quanto fatto nell'esempio "Che ora è?")

```

1 import JAM.*;
2
3 public class AstaBanditoreBehaviour extends JAMWhileBehaviour {
4     [...]
5     public void setup() throws JAMBehaviourInterruptedException {
6         // inizializzo al valore di base l'oggetto in asta
7     }
8     public void dispose() throws JAMBehaviourInterruptedException {
9         // stampo statistiche su come e' andata l'asta, chi
10        // si e' aggiudicato l'oggetto e a quanto
11    }
12    public void action() throws JAMBehaviourInterruptedException {
13        // leggo un messaggio e lo analizzo
14        // rispondo a seconda del messaggio letto
15    }
16 }

```

Un agente di tipo *cliente* si comporta nel seguente modo

```
1 import JAM.*;
2
3 public class AstaClienteBehaviour extends JAMWhileBehaviour {
4     [...]
5     public void setup() throws JAMBehaviourInterruptedException {
6         // inizializzo il mio portafoglio con una dotazione
7         // di euro a caso compreso in un certo range di valori
8     }
9     public void dispose() throws JAMBehaviourInterruptedException {
10        // stampo il risultato dell'asta, se ho mi sono
11        // aggiudicato o no l'oggetto e nel caso quanto ho
12        // speso
13    }
14    public void action() throws JAMBehaviourInterruptedException {
15        // se ho soldi nel portafoglio chiedo il valore corrente
16        // dell'oggetto in asta
17        // se ancora in asta e non sono io il migliore offerente
18        // effettuo una offerta superiore a quella migliore
19        // (es. incrementando di un valore random la migliore
20        // offerta corrente)
21        // invio la mia offerta e attendo la risposta
22        // se la mia offerta e' accettata, mi riposo qualche
23        // secondo e poi vado nuovamente a verificare se sono
24        // rimasto io il migliore offerente
25        // se la mia offerta non e' accettata ricomincio da
26        // capo
27        // se non ho soldi sufficienti per rilanciare mi ritiro
28        // rispondo a seconda del messaggio letto
29    }
30 }
```

3.7.4 Test Finale III

La seguente parte è facoltativa per la prima sessione d'esame

Si realizzi l'esempio non scelto nella Sez. 3.7.3.

3.8 Trattamento delle eccezioni

Nello svolgimento del laboratorio, ed in particolare dei Test richiesti in Sezione 3.7.3 e 3.7.4 è necessario definire un insieme di classi di eccezioni. Un possibile schema UML di questo lo si trova in Figura reffig:UML-Eccezioni.

References

- [1] M. Baldoni. Package e Documentazione in Java. Lucidi disponibili presso il supporto on-line al corso presso <http://informatica.i-learn.unito.it/> per l'a.a. 2008/2009.
- [2] KDE Community. Umbrello UML Modeller. <http://uml.sourceforge.net/>.
- [3] Gentleware. Poseidon for UML. <http://www.gentleware.com/>.

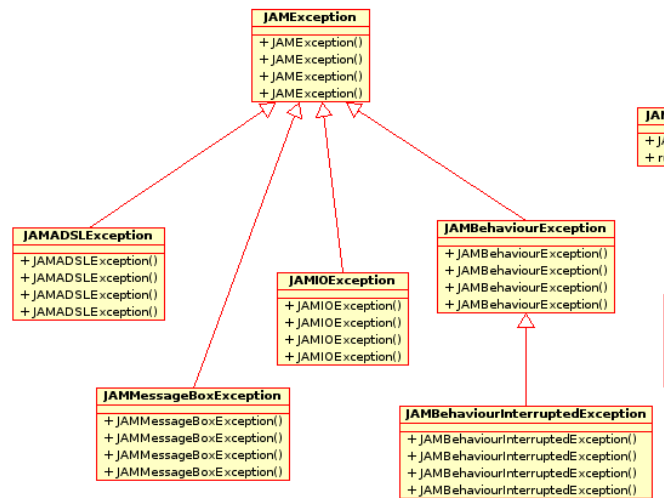


Figure 21: Diagramma delle classi che trattano le eccezioni.

- [4] C. S. Horstmann and G. Cornell. *Java 2: Fondamenti*. Pearson, Prentice Hall, 2005.
- [5] C. S. Horstmann and G. Cornell. *Java 2: Tecniche Avanzate*. Pearson, Prentice Hall, 2005.
- [6] A. Martelli. Agenti e sistemi multi-agente. <http://www.di.unito.it/~baldoni/didattica/a304/ProgInRete/Agenti.pdf>.
- [7] Sum Microsystems. The java tutorials. <http://java.sun.com/docs/books/tutorial/java/javaOO/enum.html>.
- [8] Open Source Software Engineering Tools Tigris.org. ArgoUML. <http://argouml.tigris.org/>.
- [9] TILab S.p.A. Java Agent DEvelopment frameork. <http://jade.cselt.it>.
- [10] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002. Disponibile (ma escluso dal prestito) presso la biblioteca del Dipartimento di Informatica di Torino.

A Diagramma completo delle classi

Nella Figura 22 è visualizzato il diagramma completo delle calssi da sviluppare (a parte quelle che realizzano le interfacce grafiche).

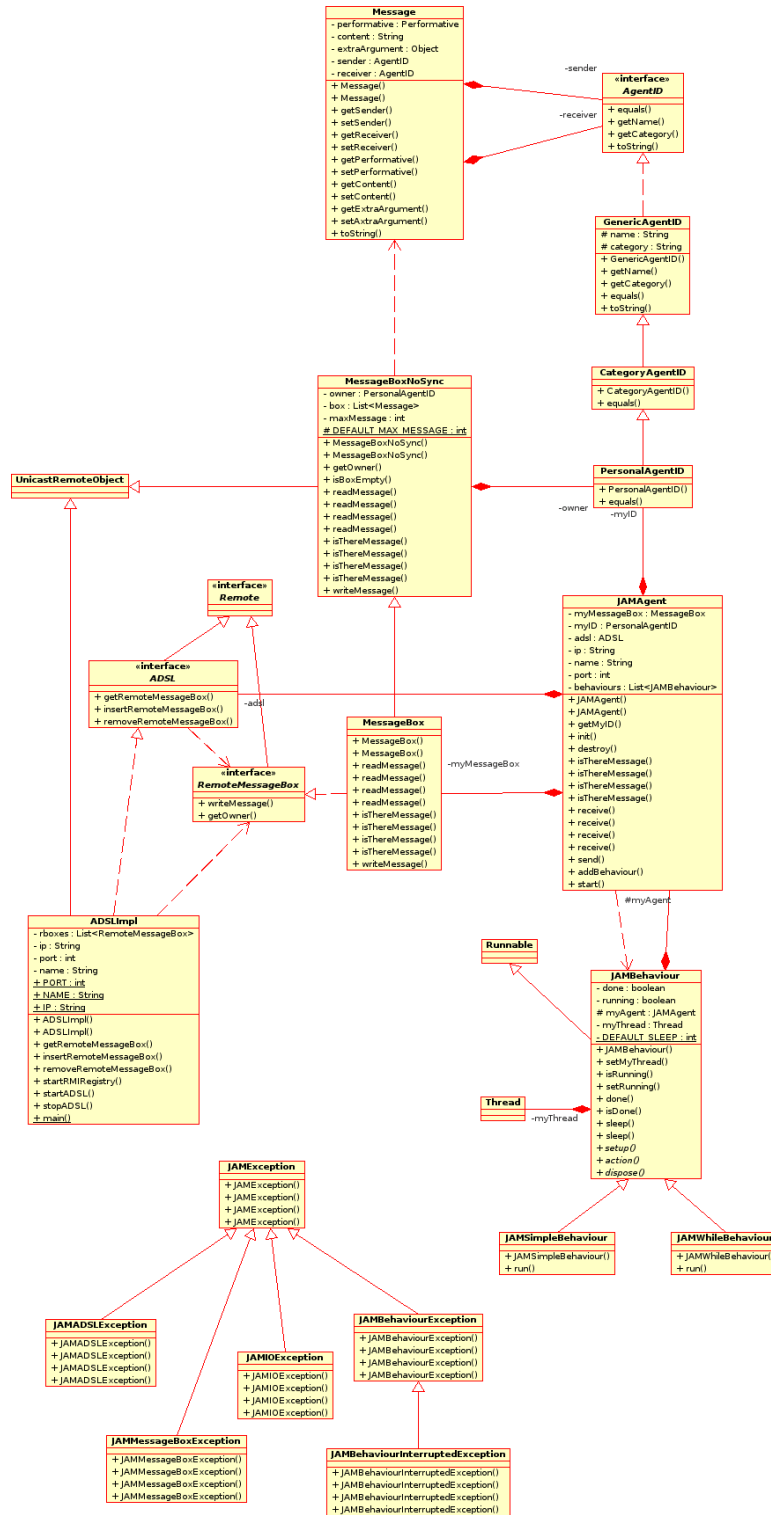


Figure 22: Diagramma delle classi (a parte le classi per la gestione delle interfacce grafiche).