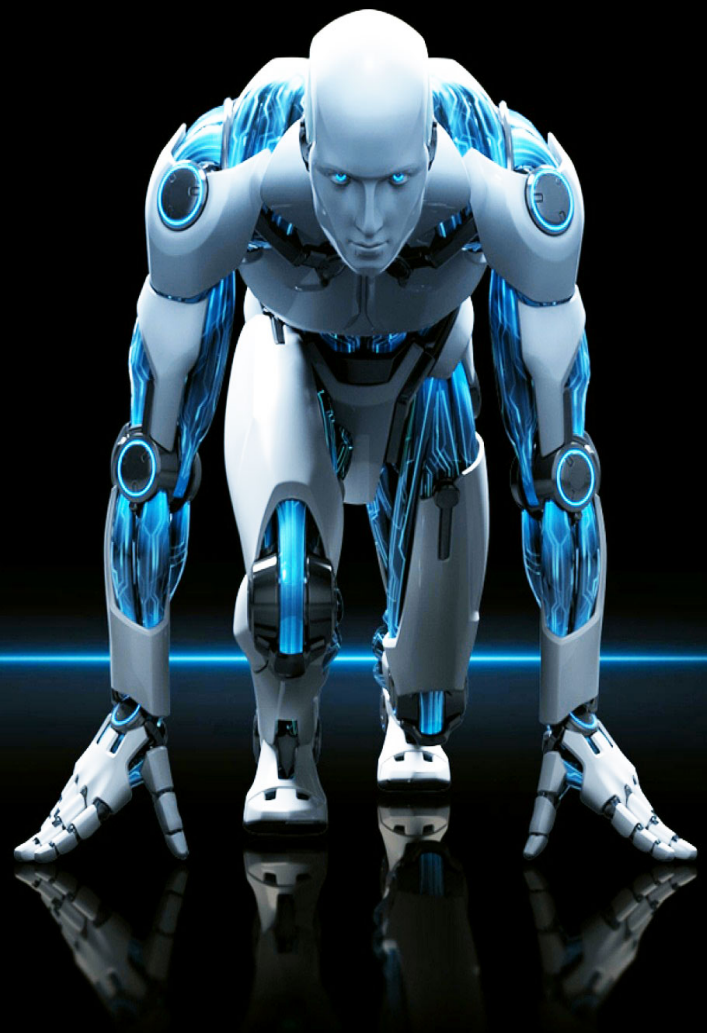
Benvenuti alla lezione di "Programming multi-agent systems in AgentSpeak using Jason"

Cap 7 - User-defined Component



Anna Tamburro

26/05/2016

Jason (User-defined component)

Jason è distribuito con una serie di funzionalità di base.

Gli sviluppatori durante le fasi di sviluppo dei loro sistemi vogliono avere una serie di requisiti (accesso ai database, integrazione dei sistemi legacy, interfacce grafiche e molti altri).

Per soddisfare questi requisiti è stato adottato un approccio basato sull'estendibilità.

Jason è stato sviluppato in java e così anche le sue estensioni e personalizzazioni includeranno alcuni programmi Java.

Gli agenti possono essere sviluppati con:

- Linguaggio dichiarativo
- Paradigma object-oriented.

Jason (User-defined component)

Nuovi comandi possono essere sviluppati usando azioni interne definite dall'utente tramite diversi componenti dell'interprete.

Ci sono 2 ragioni per creare un'azione interna:

- Estendere le capacità dell'agente (questo può creare un miglior livello di astrazione per il task richiesto rispetto a quella fornito da *AgentSpeak*)
- Consentire all'agente di utilizzare il codice legacy già programmato in Java o altri linguaggi.

Esistono anche azioni standard interne che appartengono già alla distribuzione di Jason.

E' possibile, inoltre, anche personalizzare ed estendere diverse componenti dell'interprete.

Jason (User-defined component)

Le azioni interne definite dall'utente dovrebbero essere organizzate in librerie.

L'azione *distance*, per esempio è contenuta nella libreria *math* ed è accessibile sia nel contesto che nel body dei piani di *AgentSpeak*.

Nel codice *AgentSpeak*, un'azione interna è accessibile in questo modo:

```
+event : true <- math.distance(10,10,20,30,D); ...
+event : math.distance(10,10,20,30,D) & D > 30  <-  ...
```

Jason (User-defined component)

Le librerie sono definite come pacchetti Java e ogni azione nella libreria utente dovrebbe essere una classe Java all'interno del pacchetto.

I nomi del package e della classe sono i nomi della libreria e dell'azione che saranno poi usati nei programmi *AgentSpeak*.

```
package math;
import jason.*;
import jason.asSyntax.*;
import jason.asSemantics.*;
 public class distance extends DefaultInternalAction {
        @Override public Object execute( TransitionSystem ts, Unifier un, Term[] args ) throws Exception { <
        the code that implements the IA goes here >
        }
 }
```

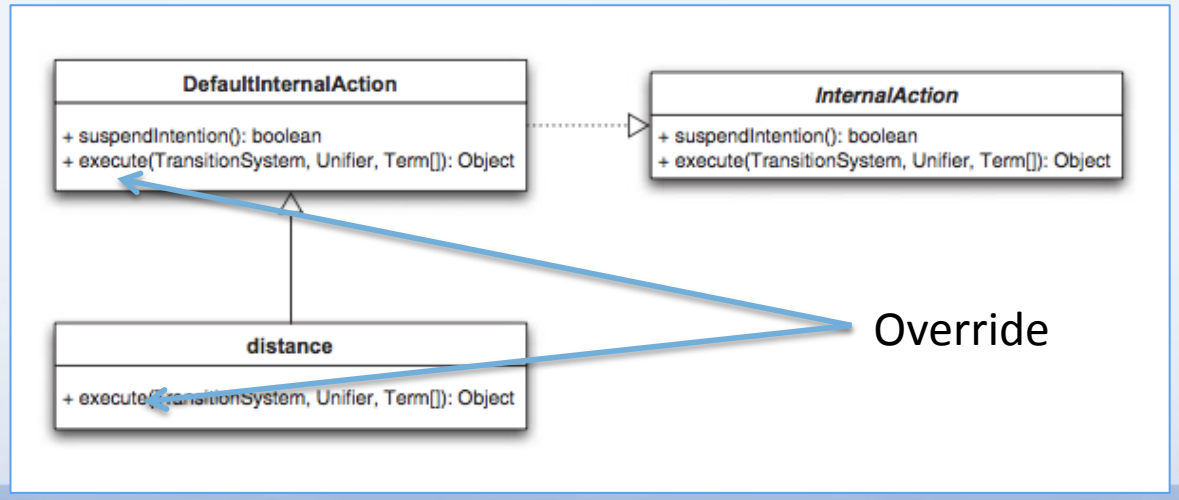
- Gli identificatori che iniziano con una lettera maiuscola in *AgentSpeak* denotano le variabili
- i nomi della libreria e della classe dell'azione interna iniziano con una lettera minuscola.

Jason (User-defined component)

Tutte le classi che definiscono azioni interne devono implementare l'interfaccia *InternalAction*.

Esiste un'implementazione di default **DefaultInternalAction** che dovrebbe semplificare la creazione di una nuova azione interna.

Una nuova classe per definire una nuova azione interna deve estendere la classe **DefaultInternalAction.**



Jason (User-defined component)

Il metodo *execute(TransitionSystem, Unifier, Term[])* viene richiamato dall'agente di *AgentSpeak*

- Il primo argomento del metodo è **TransitionSystem** che contiene le informazioni sullo stato corrente dell'agente
- Il secondo argomento è **Unifier** che determina l'esecuzione del piano in cui è apparsa l'azione interna o il controllo di se il piano è applicabile e questo dipende da se l'azione interna appare nel contesto o nel body del piano.
- Il terzo argomento **Term[]** è un array di argomenti passati alle azioni interne dall'utente nel codice di *AgentSpeak* che ha chiamato l'azione interna.

Il secondo metodo della classe **DefaultInternalAction** è chiamato **suspendIntention**: questo metodo dovrebbe restituire true quando l'azione interna provoca l'intenzione di essere sospesa.

La sua implementazione predefinita nella classe **DefaultInternalAction** restituisce false.

Jason (User-defined component)

Il codice Java nel metodo di esecuzione per l'azione interna math.distance ha tre fasi:

```
try {
        NumberTerm p1x = (NumberTerm)args[0];
        NumberTerm p1y = (NumberTerm)args[1];
        NumberTerm p2x = (NumberTerm)args[2];
        NumberTerm p2y = (NumberTerm)args[3];

        double r = Math.abs(p1x.solve()-p2x.solve()) +
        Math.abs(p1y.solve()-p2y.solve());

        NumberTerm result = new NumberTermImpl(r);
        return un.unifies(result,args[4]);
}
```

1. Si ottengono i riferimenti per i 4 numeri passati come parametri alla funzioni e per i quali sarà calcolata la distanza. **NumberTerm** è un'interfaccia implementata da **NumberTermImpl**. **VarTerm** è una variabile che può essere associata ad un numero o un'espressione aritmetica ***ArithExpr***.
2. Si calcola il valore della distanza. Il metodo solve() viene usato per ottenere il valore numerico di NumberTerm in modo che anche le espressioni possano essere usate come parametri, per esempio: math.distance (X1 + 2, Y1 / 5-1,20,20, D)
3. Viene creato un oggetto NumberTerm che rappresenta il risultato. Questo oggetto viene unificato con il quinto argomento. Tale unificazione potrebbe non riuscire o avere successo.

Jason (User-defined component)

Il codice Java nel metodo di esecuzione per l'azione interna math.distance ha tre fasi:

```
try {
        NumberTerm p1x = (NumberTerm)args[0];
        NumberTerm p1y = (NumberTerm)args[1];
        NumberTerm p2x = (NumberTerm)args[2];
        NumberTerm p2y = (NumberTerm)args[3];
        // 2. calculates the distance
        double r = Math.abs(p1x.solve()-p2x.solve()) +
        Math.abs(p1y.solve()-p2y.solve());
        // 3. creates the term with the result and
        // unifies the result with the 5th argument
        NumberTerm result = new NumberTermImpl(r);
        return un.unifies(result,args[4]);
}
```

```
catch (ArrayIndexOutOfBoundsException e) {

        throw new JasonException("The internal action
        'distance'"+
        "has not received five arguments!");
        }
 catch (ClassCastException e) {
         throw new JasonException("The internal action
        'distance'"+ "has received arguments that are not
        numbers!");
}
catch (Exception e) {
        throw new JasonException("Error in 'distance'");
 }
```

Jason (User-defined component)

Il risultato dell'unificazione è anche il risultato dell'esecuzione dell'azione interna. Nel caso in cui l'azione interna viene utilizzata nel contesto di un piano e ritorna *false* significa che il piano non è applicabile. Quando viene utilizzato in un body del piano, il return *false* farà fallirà il piano che ha chiamato l'azione interna.

Un'azione interna può restituire:

- un valore booleano: indica se l'esecuzione dell'azione interna ha avuto successo o meno
- un iteratore di unificatori: viene utilizzata per analizzare soluzioni alternative quando l'azione interna viene utilizzata nel contesto dei piani.

Personalizzazione della classe Agent

Dal punto di vista di un interprete *AgentSpeak* esteso, un agente è:

- un insieme di belief
- una serie di piani
- funzioni di selezione definite dall'utente
- funzione di fiducia (un rapporto socialmente accettabile per i messaggi ricevuti)
- Belief Update Function
- Belief Revision Function
- circumstance: comprende gli eventi in corso, le intenzioni e varie altre strutture che sono necessarie durante l'interpretazione di un agente *AgentSpeak*

L'implementazione predefinita di queste funzioni è codificata in una classe chiamata **Agent**, che può essere personalizzata dagli sviluppatori al fine di estendere le funzionalità di base.

Personalizzazione della classe Agent

La classe Agent comprende una serie di metodi su cui viene effettuato override:

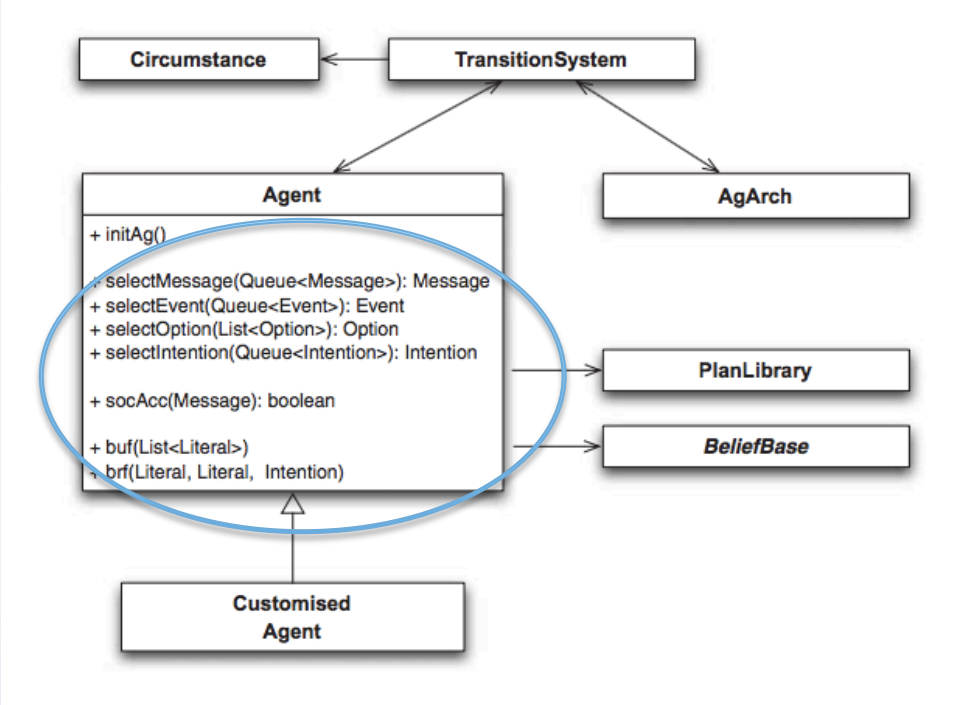


Diagramma delle classi: Agent

Personalizzazione della classe Agent

Alcuni dei metodi della classe Agent che vengono sottoposti ad override sono i seguenti:

- **selectMessage**(Queue<Message> mailbox): seleziona il messaggio che sarà gestito nel ciclo di ragionamento corrente. L'implementazione di default rimuove e restituisce il primo messaggio della posta dell'agente
- **selectEvent**(Queue<Event> events): seleziona l'evento che verrà gestito nel ciclo di ragionamento corrente. L'implementazione di default rimuove e restituisce il primo evento della coda
- **selectOption**(List<Option> options) : questo metodo viene utilizzato per selezionare una tra diverse opzioni. L'implementazione di default rimuove e restituisce la prima opzione secondo l'ordine in cui i piani sono stati scritti nel codice.

Personalizzazione della classe Agent

- **selectIntention**(Queue<Intention>intentions): seleziona l'intenzione di essere ulteriormente eseguito nel ciclo di ragionamento corrente. L'implementazione di default rimuove e restistuisce la prima intenzione in coda e dopo l'esecuzione l'intenzione viene inserita alla fine della coda. Le intenzioni vengono eseguite secondo l'algoritmo di scheduling Round Robin
- **socAcc** (Message m): restituisce true se il messaggio m è socialmente accettabile. L'implementazione di default restituisce true per tutti i messaggi. Nelle applicazioni in cui la sicurezza è un problema, è necessario sovrascriverlo perché l'agente potrebbe risultare suscettibile ad attacchi
- **buf**(List<Literal> percepts): aggiorna la base di belief con i precetti dati e aggiunge tutti i cambiamenti che sono stati effettivamente eseguiti come nuovi eventi nel set di eventi.
- **brf** (Literal add, Literal rem, Intention i): rivede la belief permettendo l'aggiunta e la rimonzione (se presente) di un un letterale. Il terzo parametro è la struttura dell' intenzione che ha richiesto il cambiamento di belief. Lo scopo di questo metodo è essere sottoposto ad override, in quanto l'implementazione di default non fa nulla.

Personalizzazione della classe Agent

La funzione di Belief Update aggiorna semplicemente la belief e genera gli eventi esterni, ma non ne garantisce la consistenza.

Spetta allo sviluppatore del modello ambiente fare in modo che le contraddizioni non si verifichino.

L'implementazione di default della Belief Update function funziona come segue:

for all $b \in B$
  if $b \notin P$
    then *delete b from B*
      add $\langle -b, \top \rangle$ *to E*
for all $p \in P$
  If $p \notin B$
    then *add p to B*
    add $\langle +p[source(percept)], \top \rangle$ *to E*

B: Insieme di tutti i litterali nella belief
P: lista dei percetti attuali
E: insieme degli eventi

Personalizzazione della classe Agent

Come esempio di personalizzazione di un agente usiamo il robot aspirapolvere.
Esempio: cambiamo *event selection function* del robot in modo da dare priorità agli eventi creati quando viene percepito lo sporco.

```
import jason.asSemantics.*;
import jason.asSyntax.*;
import java.util.*;
public class DirtyFocusAgent extends Agent {
	static Trigger focus = Trigger.parseTrigger( "+dirty[source(percept)]");
	@Override
	public Event selectEvent(Queue<Event> events) {
		Iterator<Event> i = events.iterator();
		while (i.hasNext()) {
			Event e = i.next();
			if (e.getTrigger().equals(focus)) {
				i.remove();
				return e;
			}
		}
		return super.selectEvent(events);
	}
}
```

STEP:

1. La coda di questo agente può avere eventi del tipo: dirty, pos(r), and pos(l) e in aggiunta gli eventi interni creati dai suoi propositi
2. Il metodo selectEvent controlla tutti gli elementi della coda e controlla se il trigger è uguale a dirty[source(percept)]
3. Se esiste un tale evento, viene rimosso dalla coda e restituito per essere gestito dal ciclo di ragionamento corrente
4. Altrimenti, chiama l'implementazione predefinita del metodo restituendo l'elemento all'inizio della coda.

Personalizzazione della classe Agent

Esempio: il codice seguente implementa una *intention selection function* nel codice di *AgentSpeak* per un qualche Agente.

```
import jason.asSemantics.*;
import jason.asSyntax.*;
import java.util.*;
public class IdleAgent extends Agent {
        static Term idle = DefaultTerm.parse("idle");
        @Override
       public Intention selectIntention(Queue<Intention> intentions) {
                Iterator<Intention> ii = intentions.iterator();
                while (ii.hasNext()) {
                        Intention i = ii.next();
                        if (isIdle(i)) {
                                if (intentions.size() == 1) {
                                        ii.remove();
                                        return i;
                                }
                        } else {  //si riferisce al primo if
                                ii.remove();
                        return i;
                        } //end else
                }// end while
                return null;
            }
```

- L'intenzione, chiamiamola “intenzione di inattività”, deve essere selezionata solo quando non c'è niente altro da fare
- Finchè l'agente ha molte intenzioni, non dovrebbe mai essere selezionata l'intenzione di inattività

Personalizzazione della classe Agent

Esempio: il codice seguente implementa una *intention selection function* nel codice di *AgentSpeak* per un qualche Agente.

```
private boolean isIdle(Intention i) {
        for (IntendedMeans im : i.getIMs())
{
                Pred label =
                im.getPlan().getLabel();
                if (label.hasAnnot(idle)) {
                        return true;
                }
        }
        return false;
}
}
```

- Per contrassegnare l'intenzione come 'inattivo', il programmatore aggiunge un'annotazione di inattività nell'etichetta del piano che formerà l'intenzione.
- Questo può comportare problemi di concorrenza quando nuove funzioni di selezione vengono introdotte senza troppa attenzione.
- Nel caso in cui l'ambiente sia troppo dinamico, invece, risulta utile mettere da parte qualche evento.

Personalizzazione dell'architettura

Per un agente è essenziale lavorare efficacemente in un sistema multi-agente e per fare questo deve:

- interagire con l'ambiente
- Interagire con altri agenti

L'interprete *AgentSpeak* è solo il modulo ragionamento all'interno dell'architettura di un agente complessiva che si interfaccia con il mondo esterno.

Il termine *architettura* sta ad identificare l'architettura complessiva dell'agente.

L'architettura fornisce:

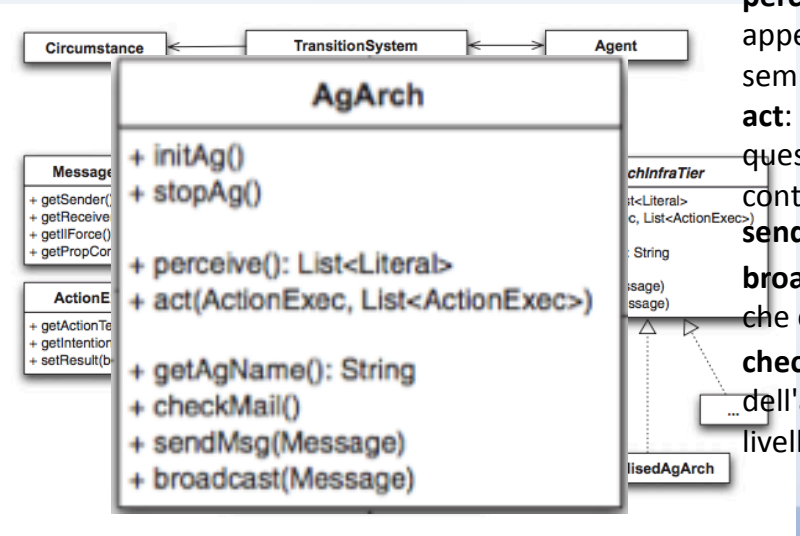
- percezione (tramite i sensori)
- Azione (modellazione dell'agente tramite gli effectors),
- il modo in cui l'agente riceve messaggi da altri agenti.

Questi aspetti possono anche essere personalizzati per ogni agente singolarmente.

Personalizzazione dell'architettura

L'implementazione dell'architettura di default si trova nella classe *AgArch* che l'utente può estendere quando risulta necessario.

Questa classe è un bridge per l'infrastruttura sottostante multi-agente in modo che il programmatore possa personalizzare, senza preoccuparsi di come viene implementata concretamente la percezione e della comunicazione che vengono gestite a livello architetturale.



perceive: restituisce un elenco di litterali che rappresentano quanto appena percepito dall'agente. L'implementazione predefinita ottiene semplicemente le percezioni inviati dall'ambiente e lo restituisce.

act: quando l'esecuzione di qualche intenzione contiene un'azione, questo metodo viene chiamato per eseguire l'azione. Il primo argomento contiene i dettagli dell'azione

sendMsg: invia un messaggio a un altro agente

broadcast: questo metodo invia un messaggio a tutti gli agenti noti che condividono l'ambiente

checkMail: questo metodo legge tutti i messaggi nella casella di posta dell'agente a livello infrastruttura e li aggiunge nella casella di posta a livello di interprete come istanze del messaggio della classe Jason.

Personalizzazione dell'architettura

Esempio: rimuovere ogni percezione relativa al posizionamento per l'agente aspirapolvere:

```
import jason.asSyntax.*;
import jason.architecture.*;
import java.util.*;
public class VCArch extends AgArch {
	@Override
	public List<Literal> perceive() {
		// utilizza la percezione di default
		List<Literal> per = super.perceive();
		// se nulla è cambianto nell'ambiente
		if (per != null) {
			Iterator<Literal> ip = per.iterator();
			while (ip.hasNext()) {
				Literal l = ip.next();
				if (l.getFunctor().equals("pos")) {
					ip.remove();
				}
			}
		}
		return per;
	}
}
```

Quando si personalizzano alcuni di questi metodi, è spesso opportuno utilizzare l'implementazione predefinita in *AgArch* e solo dopo che ha terminato la sua esecuzione si passa a processare il valore di ritorno di questi metodi.
Questa classe viene assegnata ad un agente nel file di configurazione del progetto usando la parola chiave *agentArchClass.*

```
MAS vacuum_cleaning {
	environment: VCWorld
	agents:
	vc agentArchClass VCArch;
}
```

Personalizzazione dell'architettura

Esempio: supponiamo di voler controllare, a livello di architettura, azioni per agenti aspirapolvere in modo che possa compiere solo azioni a destra e a sinistra. Le azioni che il robot può scegliere di fare non saranno eseguite, anche se l'interprete le considera eseguite con successo. Questo tipo di personalizzazione dell'architettura viene usata per evitare che gli agenti compiano qualcosa di proibito o pericoloso, dato che essi sono autonomi ed è quindi difficile garantire che non tenterà di fare certe cose.

```
@Override
public void act(ActionExec action, List<ActionExec> feedback) {
	String afunctor = action.getActionTerm().getFunctor();
	if (afunctor.equals("left") || afunctor.equals("right")) {
		// finge che l'azione sia eseguita con successo e la imposta a true
		action.setResult(true);
		feedback.add(action);
	} else {
		// richiama l'implementazione di default
		super.act(action,feedback);
	}
}
```

Personalizzazione dell'architettura

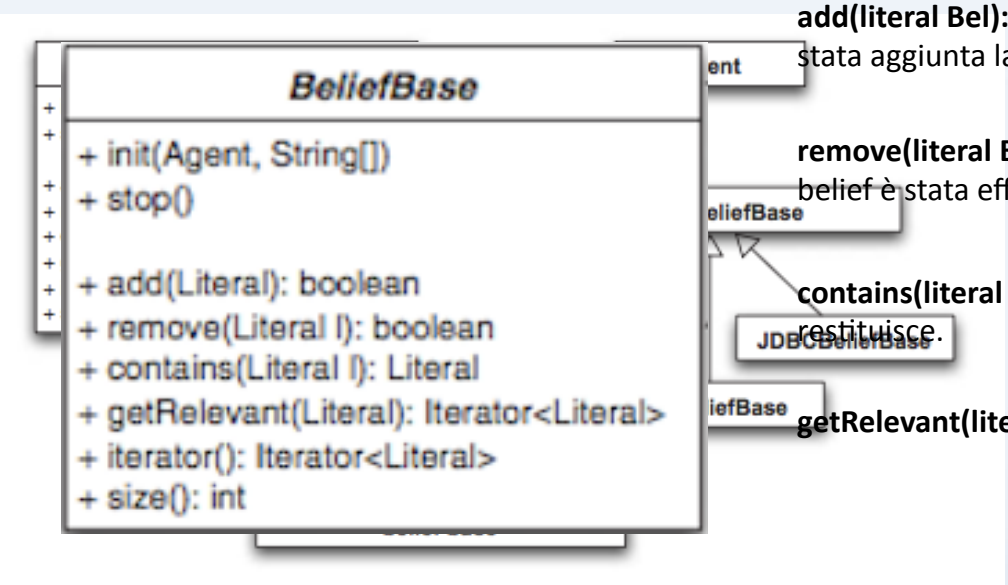
Esempio: torniamo all'esempio robot domestico e supponiamo che l'agente supermarket deve ignorare i messaggi inviati dagli agenti owner.

Si cambia il metodo di posta checkMail() in modo da rimuovere tali messaggi a livello di architettura.

```
@Override public void checkMail() {
        //richiama l'implementazione predefinitva per
        //rimuovere i messaggi con condizione
        mailbox. super.checkMail();
        //rimuove tutti i messaggi inviati dagli agenti owner
        Iterator im = getTS().getC().getMailBox().iterator();
        while (im.hasNext()) {
                Message m = (Message) im.next();
                if (m.getSender().equals("owner")) {
                        im.remove();
                        //invia un messaggio all'agente owner per
                        //informarlo che i suoi messaggi verranno ignorati
                        Message r = new Message( "tell",
                        getAgName(),
                        m.getSender(),
                        "msg(\"You are not allowed to order anything, "+
                        "only your robot can do that!\")" );
                        sendMsg(r);
                } //end if
        } //end while
} //end checkMail
```

Personalizzazione della Belief base

E' possibile personalizzare anche la classe Belief Base che è l'interfaccia implementata dalla classe *DefaultBeliefBase (*implementazione di default).



add(literal Bel): aggiunge un letterale nella BB. Restituisce *true* se è stata aggiunta la belief

remove(literal Bel): rimuove il letterale dalla BB. Restituisce *true* se la belief è stata effettivamente rimossa

contains(literal Bel): controlla se un letterale si trova bella BB e lo restituisce.

getRelevant(literal Bel): restituisce un iteratore per tutte le belief

Personalizzazione della Belief base

Ci sono due personalizzazioni della BB disponibili con la distribuzione Jason:

1. memorizza le belief in un file di testo (in modo da mantenere lo stato della BB di un agente)
2. memorizza alcune delle belief in un database relazionale (può essere usata per l'accesso a qualsiasi db relazionale).

Personalizzazione della Belief base

Memorizzazione delle belief in un file di testo

Per configurare l'agente ad utilizzare la BB persistente in un unico file si usa questa configurazione:

```
MAS custBB {
    agents:
        a beliefBaseClass jason.bb.TextPersistentBB;
}
```

Ogni volta che si esegue questo agente, le sue belief vengono caricate da un file (denominato <nome dell'agente> .bb) e, prima di terminare, le sue belief sono memorizzate nello stesso file.

Si noti che il codice *AgentSpeak* rimane lo stesso indipendentemente dalla personalizzazione della BB. L'attuazione di questa BB sostituisce semplicemente l'init e ferma i metodi per caricare e salvare le belief.

Personalizzazione della Belief base

Memorizzazione delle belief in un database relazionale

- Per utilizzare la persistenza in un database, (Java DataBase Connectivity) si usa la feature Java JDBC
- Non vengono memorizzate tutte le belief in un database, ma solo quelle esplicitamente elencate
- La connessione personalizzata richiede 5 parametri:
  - Driver jdbc
  - URL per la connessione
  - Nome utente
  - Password
  - Una lista di belief mappate nella tabella *tablece*

```
MAS custBB {
    agents:
    a beliefBaseClass jason.bb.JDBCPersistentBB(
    "org.hsqldb.jdbcDriver", // driver for HSQLDB
    "jdbc:hsqldb:bookstore", // URL connection
    "sa", // user
    "", // password
    "[count_exec(1,tablece)]");

}
```

Personalizzazione della Belief base

Memorizzazione delle belief in un database relazionale

- Se la tabella *tablece* non esiste, viene creata
- Con questa configurazione, sia la lettura che la modifica alle belief, vengono mappate con comandi SQL.

```
MAS custBB {
        agents:
        a beliefBaseClass jason.bb.JDBCPersistentBB(
        "org.hsqldb.jdbcDriver", // driver for HSQLDB
        "jdbc:hsqldb:bookstore", // URL connection
        "sa”, // user
        "", // password
        "[count_exec(1,tablece)]");

}
```

Pre-processing Directives

Il linguaggio *AgentSpeak* interpretato da Jason accetta direttive del compilatore.

Le direttive vengono utilizzate per passare alcune istruzioni all'interprete non legate alla semantica del linguaggio, ma semplicemente sintattiche.

Esistono due tipi di direttive che possono essere inserite in qualsiasi punto del codice *AgentSpeak*:

- single-command directives
- scope directives

Esempio di Single-command directives :

"{" <directive-name> "(" <parameters> ")" "}"

Esempio di Scope directives:

"{" "begin" <directive-name> "(" <parameters> ")" "}"
<agentspeak-program>
"{" "end" "}"

Pre-processing Directives

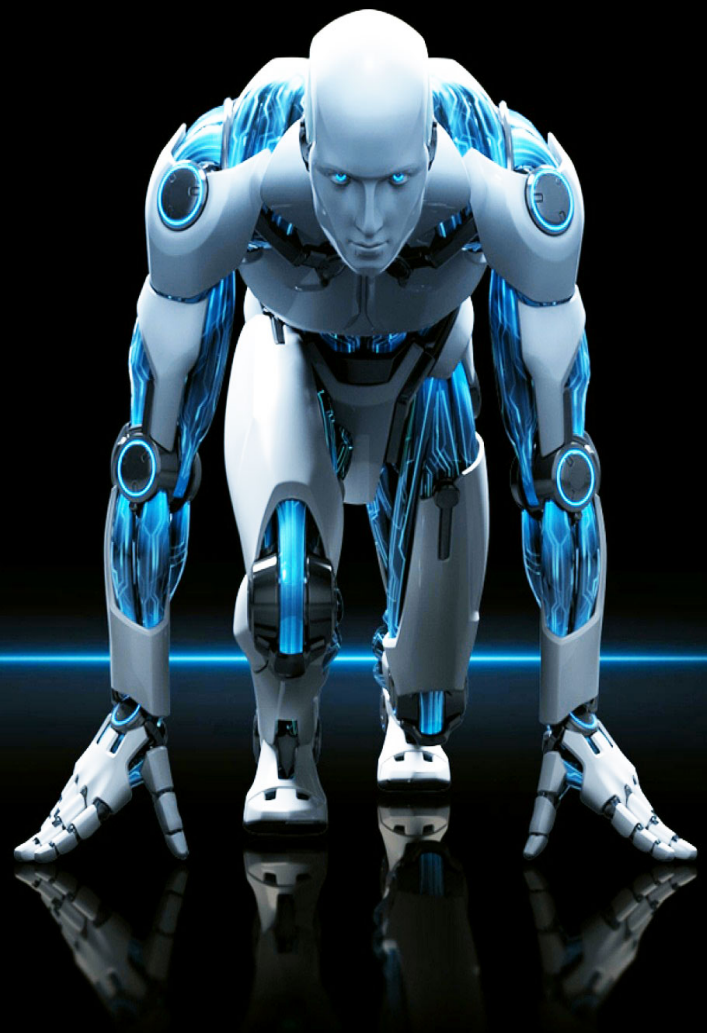
Una caratteristica interessante di Jason è che l'insieme di direttive del compilatore può essere facilmente esteso dal programmatore.

Ogni nuova direttiva viene programmata in una classe che implementa l'interfaccia *Directive*. Questa interfaccia ha solo un metodo:

- Agent process(Pred directive, Agent outterContent, Agent innerContent):
  - il parametro predicato rappresenta la direttiva
  - outterContent rappresenta il punto in cui deve essere usata questa direttiva
  - innerContent indica il contenuto racchiuso dalla direttiva.
  - Il contenuto interno ha una serie di piani e belief che devono essere modificati dalla direttiva
  - Il metodo deve restituire una nuova istanza dell'agente in cui sono contenuti i piani e le belief modificati.

FINE

Spero si sia capito qualcosa :)



Anna Tamburro
26/05/2016