AgentService

Indice

Capitolo 1: Introduzione ... 3
1.1 La piattaforma AgentService ... 3
Capitolo 2: Installazione ed esecuzione della piattaforma ... 8
2.1 Il pacchetto AgentService ... 8
2.2 Driver della piattaforma ... 8
2.3 Il processo d'installazione ... 9
2.4 I file di configurazione ... 10
2.5 L'esecuzione della piattaforma ... 11
2.5.1 Prima esecuzione ... 12
2.5.2 Esecuzione standard ... 12
2.5.3 Esecuzione batch-script ... 13
2.6 Il ciclo di vita della piattaforma ... 14
Capitolo 3: Definizione di un agente ... 16
3.1 Definizione di tipi Knowledge ... 17
3.2 Definizione dei Behaviour ... 19
3.3 Definizione di AgentTemplate ... 22
3.4 L'agente – AID (AgentIdentifier) ... 24
3.5 Istanziazione ed esecuzione degli agenti in modalità batch ... 25
3.5.1 La classe ModuleError ... 28
Capitolo 4: Il comportamento a runtime ... 29
4.1 Servizio di scrittura su Console – IConsoleProxy ... 30
4.2 Servizio di messaggistica – IMsgClient ... 30
4.2.1 Struttura di un messaggio – AgentMessage ... 32
4.2.1.1 MessageEnvelope ... 32
4.2.1.2 MessageBody ... 33
4.2.1.3 AgentMessage ... 34
4.2.2 Comunicazione basata su semplice scambio di messaggi ... 36
4.2.3 MessageFilter ... 36
4.2.3.1 Classe MessageFilter ... 37
4.2.3.2 Filtri derivati ... 38
4.2.3.3 Utilizzo pratico dei filtri ... 38
4.2.4 Gestione delle conversazioni ... 39
4.2.4.1 Ciclo di vita di una conversazione ... 40
4.2.4.2 L'interfaccia IConversation ... 41
4.2.4.3 La conversazione ... 44

4.3 Servizio di logging – ILogClient ... 48
4.4 Servizio di pagine bianche – IWhitePagesClient ... 49
4.5 Servizio di pagine gialle – IYellowPagesClient ... 50
4.6 Servizio di persistenza – IPersistenceClient ... 53
4.7 Servizio di controllo dei behaviour – IBehaviourContext ... 53
Capitolo 5: Gestione del MAS ... 55
5.1 La classe PlatformController ... 55
5.1.1 Servizi accessibili attraverso la proprietà Services ... 56
5.1.2 Servizi accessibili attraverso la proprietà Modules ... 57
Capitolo 6: Estendere AgentService ... 58
6.1 Creazione di moduli ... 58
6.1.1 L'interfaccia IPlatformModule ... 59
6.1.2 Le interfacce IPlatformContext e IPlatformContextEx ... 61
6.1.3 L'interfaccia ILoggingModule ... 66
6.1.4 L'interfaccia IMessaggingModule ... 67
6.1.5 L'interfaccia IPersistenceModule ... 69
6.1.6 L'interfaccia IStorageModule ... 70
6.2 Esempio di modulo addizionale – SnoopingModule ... 71
Capitolo 7: Il servizio di supporto delle ontologie ... 79
7.1 Le metaclassi ontologiche ... 79
7.1.1 Attributi e vincoli ... 80
7.1.2 Schemi ... 80
7.1.3 La classe Ontology ... 82
7.1.4 Progettazione di vincoli ... 83
7.1.5 Descrittori astratti e introspector ... 83
7.2 Progettare un'ontologia ... 84
7.2.1 BasicOntology ... 85
7.2.2 Aggiunta degli schemi e degli slot ... 85
7.2.3 Progettazione con Protégé ... 87
7.2.3.1 IRE ... 88
7.2.3.2 Ontology ... 88
7.2.4 Protege2AgentService ... 88
7.3 SL ... 88
7.4 Conversazioni con supporto ontologico ... 89
7.4.1 Come aprire una conversazione ... 90
7.4.2 Conversazioni in ingresso ... 91
La libreria AgentService ... 92
7.5 I namespace ... 92

Capitolo 1: Introduzione

1.1 La piattaforma AgentService

AgentService è una piattaforma di programmazione ad agenti basata sulla Common Language Infrastructure (CLI). Le caratteristiche principali di AgentService sono:

- Il modello di agente che semplifica il design e l'implementazione degli agenti.
- Un'integrazione completa con la CLI.
- L'architettura modulare.
- Il sistema di schedulazione dell'attività degli agenti.

Il modello di Agente:

In AgentService, un agente è un componente software in grado di agire in modo autonomo prelevando informazioni dall'ambiente in cui si trova e agendo secondo la propria base di conoscenza. Un agente può scambiare informazioni con altri agenti o con esseri umani e può prendere l'iniziativa per soddisfare i propri obiettivi: è quindi un'entità autonoma avente un certo grado di "intelligenza", creata per poter adattare i propri comportamenti ai cambiamenti dell'ambiente.
Come indicato dalle specifiche proposte da FIPA (Foundation for Intelligent and Phisical Agent), ogni agente è caratterizzato da un identificatore univoco chiamato AID (Agent IDentifier).
L'agente è composto da due elementi fondamentali: comportamenti e unità di conoscenza.
Un agente gestisce un insieme di dati (*Knowledge*) e utilizza un insieme di comportamenti concorrenti (*Behaviour*) che ne determinano l'attività.
La *Knowledge* è la base di conoscenza dell'agente; essa consiste di tutto ciò che l'agente conosce e deve conoscere per svolgere le sue attività e portare così a termine gli obiettivi che si è prefissato: include le strutture dati in cui l'agente immagazzina le informazioni che occorrono ai behaviour per la loro attività, informazioni che essi stessi accrescono e condividono. Inoltre le Knowledge sono rese persistenti dalla piattaforma in modo tale che possano essere recuperate in caso di crash del sistema: in questa maniera gli agenti sono in grado, se necessario, di ripristinare il loro stato.
I *Behaviour*, costituiscono i task degli agenti e contengono tutta la logica computazionale degli agenti. I Behaviour operano in un contesto concorrente e possono avere dei propri dati privati. A tali dati non è però garantita la persistenza come nel caso degli elementi

della Knowledge, dal momento che, per definizione, non concorrono a formare lo stato dell'agente.

All'interno di AgentService vi è una netta distinzione tra la definizione di Agente e le corrispettive istanze schedulate a runtime. Tale distinzione viene realizzata attraverso l'utilizzo di due classi, rispettivamente la classe *AgentTemplate* e la classe *Agent*. AgentTemplate è la classe che viene utilizzata dal programmatore della piattaforma per definire un nuovo tipo di agente, al contrario la classe Agent rappresenta l'agente vero e proprio. Agent è totalmente trasparente all'utente del framework e si occupa della gestione delle attività e dello stato dell'agente mantenendo un riferimento al corrispondente AgentTemplate.
La relazione tra Agent ed AgentTemplate quindi, è simile alla relazione che c'è nel mondo ad oggetti tra la classe (tipo) e le sue istanze: ciascuna istanza di agente opera secondo quanto definito nel corrispondente AgentTemplate. Questa separazione consente di ottenere molti vantaggi tra i quali il fatto di poter mantenere separate la definizione di agente e la sua istanza, di impedire al programmatore di possedere un reference all'agente vero e proprio, e la possibilità di modificare il supporto runtime e la sua struttura senza influenzare l'interfaccia AgentTemplate, quella cioè che viene definita dal programmatore. Inoltre l'agente definito dall'utente dovrà ereditare le caratteristiche da una classe semplice e molto snella, e non avrà a disposizione i metodi e gli attributi che la piattaforma utilizza per la gestione dell'agente, i quali sono necessariamente implementati ad oggetti e sono contenuti in una differente classe a lui non visibile.

Per chiarire il concetto di agente si può fare un paragone tra programmazione ad agenti e programmazione ad oggetti. Agenti ed oggetti mantengono un certo grado di parallelismo (è bene ricordare che, nel nostro caso, gli agenti sono programmati tramite oggetti), ma hanno delle differenze sostanziali: gli oggetti sono passivi, completamente controllati e caratterizzati da una comunicazione sincrona; gli agenti invece sono attivi, autonomi e caratterizzati da una comunicazione asincrona. Gli agenti quindi possono prendere iniziative, hanno il controllo su come e quando elaborare le richieste esterne e normalmente agiscono in cooperazione o coordinazione con altri agenti, formando delle comunità. Ciascun agente mantiene sempre la propria autonomia: è a conoscenza dei servizi offerti dagli altri agenti, ma non della loro implementazione. Per questo motivo la comunicazione avviene solamente tramite lo scambio di messaggi che vengono affidati al sistema di trasporto infrastrutturale che si occupa di recapitarli al destinatario.

La Common Language Infrastructure:

Un agente sviluppato con AgentService ha accesso completo ai servizi offerti dalla CLI. CLI è uno standard ECMA e ISO-IEC che definisce un ambiente di esecuzione virtuale. Tale infrastruttura ha alcune importanti caratteristiche:

- Interoperabilità tra i linguaggi di programmazione: in questo modo è possibile integrare tra loro moduli scritti in differenti linguaggi senza che sia necessario costruire software ad hoc.

- Application domain: sono l'unità di esecuzione all'interno della CLI. In un singolo processo possono essere presenti molti application domains, ciascuno con permessi diversi.
- Remoting: è una tecnica per la comunicazione tra application domain di computer, sistemi operativi o processori diversi. La tecnica consiste nell'utilizzo di appositi canali di comunicazione e di apposite regole di comportamento per gli agenti che hanno la necessità di comunicare oltre i confini del dominio.

Architettura modulare:

La piattaforma AgentService è divisa in moduli che possono essere configurati in modo da adattare la piattaforma a contesti differenti (alcuni servizi se non sono necessari possono essere disattivati). Tali moduli si distinguono in *core* e *additional*. I primi sono i moduli base che devono essere sempre presenti mentre i secondi sono moduli aggiuntivi che gli utenti della piattaforma possono creare.
I moduli core sono:

- **Logging**: si occupa della creazione di un sistema di log per gli agenti della piattaforma.
- **Messagging**: fornisce agli agenti un canale di comunicazione per lo scambio di messaggi.
- **Persistence**: è responsabile del salvataggio e ripristino della knowledge degli agenti in caso di crash del sistema.
- **Storage**: gestisce l'installazione degli assembly in cui sono definiti gli oggetti agent template, behaviour e knowledge. Ogni volta in cui si realizza un nuovo tipo di agente sulla piattaforma, tutti gli assembly dipendenti dall'agente devono essere posti nello storage. Successivamente quando l'agente verrà istanziato, nello storage si potranno trovare tutte le informazioni necessarie alla sua creazione ed esecuzione.

Schedulazione degli agenti:

La schedulazione è un punto cruciale nella realizzazione della piattaforma poiché agenti e behaviour devono poter operare in condizioni di computazione parallela. Il modello di schedulazione utilizzato da AgentService deve garantire l'autonomia richiesta dagli agenti offrendo un contesto multi-threading ai programmatori.
Il modello viene realizzato con l'utilizzo da parte di AgentService di un *application domain* per ciascun agente in esecuzione sulla piattaforma. Inoltre per ottenere l'attività multi-comportamentale degli agenti si crea un thread per ciascun behaviour usato da un agente. I deadlock e le corse nei behaviour vengono evitati usando strutture di sincronizzazione costruite all'interno di AgentService.
L'utilizzo degli application domain offre numerosi vantaggi: garantisce l'autonomia e l'isolamento di ciascun agente in esecuzione mantenendo una certa facilità di gestione sopratutto in rispetto all'utilizzo di processi del SO. Inoltre data la semplicità di gestione

degli application domain è possibile semplificare il controllo degli agenti da parte dello schedulatore della piattaforma.

Sistema multi-agente:

Gli agenti sono stati creati per operare in una comunità chiamata MAS (Multi-Agent System). I MAS sono sistemi decentralizzati con un controllo distribuito e una computazione asincrona, essi si occupano dell'ambiente runtime e definiscono l'infrastruttura per l'interazione e la comunicazione tra gli agenti. I MAS, generalmente, forniscono un insieme di servizi utili per gli agenti come il servizio di trasporto dei messaggi e il servizio di directory. Una tra le più diffuse specifiche architetturali per sistemi multi-agente è quella proposta dalla Foundation of Intelligent Phisical Agents (FIPA), un'organizzazione internazionale che promuove standard per le tecnologie ad agenti.
FIPA prevede la presenza di alcune componenti che realizzano le funzionalità principali del sistema multi-agente. L'architettura prevede la presenza, oltre che degli agenti, di altri componenti, obbligatori per il corretto svolgimento delle attività del framework. Sono appunto tali componenti a fornire servizi agli agenti. E' prevista anche la possibilità di interazioni con componenti esterni alla piattaforma, in particolare con altre entità software che non siano agenti.

I servizi della piattaforma:

AgentService prevede quindi, seguendo le indicazioni di FIPA, la presenza di componenti che hanno il compito di fornire determinati servizi agli agenti presenti all'interno della piattaforma.
I componenti dell'architettura che forniscono servizi vengono implementati come agenti e sono:

- L'**Agent Management System** (AMS):
  è il supervisore e il controllore del MAS ed è responsabile della schedulazione degli agenti (in una piattaforma può esisterne solo uno).
  L'AMS è il componente che esercita il compito di supervisore, controllando l'accesso e l'uso della piattaforma: è quindi responsabile dell'identificazione e della registrazione degli agenti. Ogni agente infatti deve registrarsi con l'AMS per ottenere un AID riconosciuto; successivamente quando la sua vita termina, il suo AID diviene disponibile per essere assegnato, eventualmente, ad un altro agente.
  L'AMS gestisce l'intero ciclo di vita di ogni agente: può richiedere ad un agente di svolgere uno specifico compito e può forzare l'esecuzione di un particolare task; fornisce inoltre un servizio di "elenco del telefono" (white pages) tenendo aggiornato un indice di tutti gli agenti presenti in ogni istante sulla piattaforma (mantiene una directory degli AID per gli agenti registrati).
  Un agente quindi può richiedere quattro diversi servizi all'AMS:
  Register: realizza le operazioni di registrazione di un agente.

Deregister: realizza le operazioni di deregistrazione di un agente.
Search: realizza le operazioni di ricerca di un agente.
Modify: realizza le operazioni di modifica dei parametri descrittivi di un agente.

- Il **Directory Facilitator** (DF): fornisce un servizio di "pagine gialle" (yellow pages) per permettere agli agenti che lo desiderano di far conoscere le proprie funzionalità ad altri agenti.
  Il DF quindi mantiene una lista di agenti dando origine ad un dominio (AD, Agent Domain), cioè ad un insieme di agenti e servizi. L'utilità del DF consiste nella possibilità di informare il sistema, l'AMS ed altri agenti (all'interno o all'esterno della piattaforma) dell'esistenza e delle funzionalità degli agenti iscritti (l'AMS gestisce l'esistenza ma non le funzionalità). Gli agenti quindi possono usare la funzione di ricerca del DF per trovare tutti gli agenti che offrono un particolare servizio.
  In ogni piattaforma deve esistere almeno un agente DF (possono coesistere più DF). Il DF inoltre è abilitato al mantenimento di collegamenti con altri DF situati su altre piattaforme.
  Il DF quindi, oltre ad avere le stesse funzioni dell'AMS (register, deregister, search, modify), consente agli agenti di registrarsi indicando non solo il nome, ma anche i particolari servizi esposti.

- Il **Message Transport System** (MTS): si occupa dello scambio di messaggi all'interno della piattaforma e della comunicazione tra diverse piattaforme.
  L'MTS viene descritto in modo astratto dalle specifiche FIPA come il servizio che consente agli agenti di comunicare tra di loro e con AMS e DF.

Capitolo 2: Installazione ed esecuzione della piattaforma

2.1 Il pacchetto AgentService

Nel pacchetto AgentService le cartelle principali sono:
- Prototype
- Install

La prima contiene il codice sorgente suddiviso in varie directory. In *AgentService 1.0* si può aprire con Microsoft Visual Studio .NET il file *AgentService.csproj*, che carica il progetto dell'intera piattaforma. Nelle altre cartelle sono contenuti i file relativi ai progetti delle singole parti.

La seconda directory contiene i file necessari all'installazione:
- *AgentService.dll* -> Libreria per il funzionamento della piattaforma
- *AgentServiceDefaultModules.dll* -> Libreria in cui sono contenuti i moduli di default
- *xmlconf.dll* -> Libreria per il supporto all'xml
- *MSMQModule.dll* -> Libreria per l'utilizzo di un sistema di trasporto messaggi basato su Microsoft Message Queue, differente quindi da quello implementato di default
- *asdrv.exe* -> Driver della piattaforma, è un programma eseguibile, che può essere lanciato da console, indicando gli opportuni parametri, per l'installazione e l'esecuzione della piattaforma
- *install.xml* -> File in formato xml in cui settare i parametri dell'installazione della piattaforma

2.2 Driver della piattaforma

Il comando *asdrv* (Agent Service Driver), lanciato da Console, permette di effettuare diverse operazioni a seconda dei parametri con cui viene eseguito. Le possibili alternative sono:

- *asdrv –i:<nome_file_d'installazione>* -> Installa la piattaforma con i parametri che gli vengono passati nel file d'installazione. Questo file è scritto in xml e in seguito verrà descritto come editarlo per effettuare l'installazione.
- *asdrv –u:<nome_file_d'installazione>* -> Disinstalla tutti i file relativi alla piattaforma.
- *asdrv –r:<nome_file_di_configurazione>* -> Esegue la piattaforma con i parametri scritti nel file di configurazione .conf creato automaticamente all'atto dell'installazione.
- *asdrv –x:<nome_file_batch-script>* -> esegue il batch-script definito nel file con estensione .abs
- *--text _format* -> è un'opzione che può essere aggiunta agli altri comandi e definisce il formato dei file d'installazione e configurazione come file di testo invece che file xml.
- *asdrv* -> Se chiamato senza parametri, viene lanciato il comando di default *asdrv –r:conf/ platform.conf*

2.3 Il processo d'installazione

Tale processo installa la piattaforma nella directory desiderata (in seguito chiamata directory base), utilizzando le informazioni presenti nel file xml d'installazione.
Per prima cosa è quindi necessario editare il file *install.xml* presente nella cartella *install* per settare i parametri dell'installazione.
Analizziamo il seguente file d'esempio:

```
<?xml version="1.0" encoding="utf-8" ?>
– <installation xmlns="install.xsd">
    <base-dir path="C:\AgentService\Demo" />
    <repository file="install.dat" />
    <description name="AgentService" mobile="false" dynamic="true" transport-
      profile="[none]" />
  – <core>
      <storage full-type-
        name="AgentService.Platform.Modules.Default.DefaultStorageModule"
        assembly-name="AgentServiceDefaultModules.dll" config-file="storage.conf"
        />
      <persistence full-type-
        name="AgentService.Platform.Modules.Default.DefaultPersistenceModule"
        assembly-name="AgentServiceDefaultModules.dll" config-
        file="persistence.conf" />
      <messaging full-type-
        name="AgentService.Platform.Modules.Default.DefaultMessagingModule"
        assembly-name="AgentServiceDefaultModules.dll" config-
        file="messaging.conf" />
      <logging full-type-
        name="AgentService.Platform.Modules.Default.DefaultLoggingModule"
```

```
      assembly-name="AgentServiceDefaultModules.dll" config-file="logging.conf"
      />
  </core>
  <additional />
</installation>
```

- `<installation xmlns="install.xsd">` -> schema xml utilizzato per la validazione del file d'installazione
- `<base-dir path="Nome_path" />` -> permette di settare il path in cui installare la piattaforma
- `<repository file="install.dat" />` -> nome del file repository dei dati dell'installazione
- `<description name="AgentService" mobile="false" dynamic="true" transport-profile="[none]" />` -> tag descrittivo sulle caratteristiche della piattaforma
- `<core>` -> tag contenente l'elenco dei moduli base che si vuole vengano caricati sulla piattaforma. Per il suo corretto funzionamento è necessario che vengano caricati almeno i moduli *storage, persistence, messaging, logging*. La sintassi è la seguente: *nome_classe nome_assembly* (in cui è contenuto il modulo specificato) *nome_file_di_configurazione* (che viene creato durante l'installazione)
- `<additional>` -> tag contenente l'elenco dei moduli addizionali. La sintassi è uguale a quella precedentemente descritta

Dopo aver settato i parametri l'installazione si avvia editando il comando da console: *asdrv –i:<nome_file_d'installazione>*, all'interno della directory Install. Viene quindi creata la directory base (come specificato nel file d'installazione) ed al suo interno vengono generati:

- *asdrv.exe*
- *AgentService.dll, xmlconf.dll*
- *agents* -> cartella che gli agenti possono utilizzare per salvare dei dati
- *conf* -> cartella che contiene i file di configurazione. (Vedi 2.4)
- *modules* -> cartella al cui interno si trovano i moduli core e additional, per ciascuno di essi troviamo la libreria all'interno di *bin* e i file di configurazione all'interno di *conf*.

Il processo d'installazione terminerà la prima volta in cui verrà lanciata la piattaforma (vedi 2.5.1).

2.4 I file di configurazione

Al termine della prima parte del processo di installazione viene generato il file *platform.conf* all'interno della directory *conf* . Il corrispondente file di configurazione del file di installazione precedentemente discusso è mostrato qui di seguito:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
- <platform first-start="false" skip-failed-modules="true">
    <base-dir path="C:\AgentService\Demo" />
    <repository file="install.dat" />
    <description name="AgentService" transport-profile="[none]" mobile="false"
      dynamic="true" />
  - <modules>
    - <core bin-path="modules\core\bin\" conf-path="modules\core\conf\">
        <storage key="1" full-type-
          name="AgentService.Platform.Modules.Default.DefaultStorageModule"
          assembly-name="AgentServiceDefaultModules.dll" config-
          file="storage.conf" />
        <messaging key="3" full-type-
          name="AgentService.Platform.Modules.Default.DefaultMessagingMod
          ule" assembly-name="AgentServiceDefaultModules.dll" config-
          file="messaging.conf" />
        <persistence key="2" full-type-
          name="AgentService.Platform.Modules.Default.DefaultPersistenceMod
          ule" assembly-name="AgentServiceDefaultModules.dll" config-
          file="persistence.conf" />
        <logging key="4" full-type-
          name="AgentService.Platform.Modules.Default.DefaultLoggingModule
          " assembly-name="AgentServiceDefaultModules.dll" config-
          file="logging.conf" />
      </core>
      <additional bin-path="modules\additional\bin\" conf-
        path="modules\additional\bin\" />
    </modules>
  </platform>
```

Come si può vedere è molto simile al file *install.xml* (da cui deriva i parametri) tranne per il campo *key* presente per ogni modulo caricato. Tale campo è inizialmente vuoto; viene poi inizializzato con dei valori interi alla prima esecuzione della piattaforma. Questi sono usati internamente dalla piattaforma per reperire le informazioni di installazione associate ad ogni modulo e non devono essere modificate.
Altri file di configurazione vengono creati per ciascun modulo all'interno delle rispettive directory (generati dopo la prima esecuzione).

2.5 L'esecuzione della piattaforma

L'esecuzione della piattaforma avviene tramite il comando *asdrv* eseguito all'interno della directory di installazione di AgentService (nel caso discusso precedentemente:

C:\AgentService\Demo). L'esecuzione del comando *asdrv* senza alcun parametro aggiuntivo comporta l'attivazione della piattaforma utilizzando come file di configurazione quello associato al percorso relativo *conf\platform.conf,* in alternativa è possibile specificare un altro file di configurazione utilizzando il parametro *–r:config_file*).
E' inoltre possibile attivare la piattaforma in modalità batch con lo switch *-x:script-file* (.abs).
Nel caso in cui sia la prima installazione, vengono effettuate delle operazioni aggiuntive, descritte in 2.5.1, altrimenti viene eseguito subito ciò che è descritto in 2.5.2.

2.5.1 Prima esecuzione

La prima esecuzione della piattaforma comporta il completamento del processo di installazione. In questa fase avviene l'installazione effettiva dei moduli. Nel caso dell'installazione dei moduli di default vengono effettuate le seguenti operazioni:

- generazione dei file di configurazione dei moduli caricati
- creazione della cartella *logs* con all'interno i log della piattaforma e degli agenti e gestito dal *DefaultLoggingModule*
- creazione della cartella *persistence* contenente i dati degli agenti necessari alla loro persistenza e gestito dal *DefaultPersistenceModule*
- creazione della cartella *storage* che conterrà gli assembly in cui sono definiti *AgentTemplate*, *Knowledge* e *Behaviour* (*DefaultStorageModule*)

Al termine dell'installazione dei moduli l'esecuzione prosegue normalmente.

2.5.2 Esecuzione standard

L'esecuzione della piattaforma prevede i seguenti step:

- se non è la prima volta che viene lanciata, vengono recuperate le informazioni dal repository delle informazioni d'installazione
- vengono caricati i vari moduli (prima i core modules, poi gli additional modules)
- vengono attivati i core agents:
  - AMS (Agent Management System)
  - MTS (Message Transport Service)
  - DF (Directory Facilitator)
- vengono attivati gli agenti definiti dall'utente

A questo punto la piattaforma è pronta ed in esecuzione.
Per interrompere l'esecuzione della piattaforma è sufficiente premere il tasto INVIO che attiva il procedimento inverso:

- vengono stoppati gli user agents ed eventualmente persistiti
- vengono stoppati i core agents e persistiti (prima DF, poi MTS e quindi AMS)
- la piattaforma passa in modalità ready
- inizia la procedura di chiusura:
    - vengono scaricati i moduli addizionali
    - vengono scaricati i moduli core
    - viene aggiornato il file di configurazione
- termine della procedura di chiusura

Per una trattazione più esaustiva si rimanda al paragrafo 2.6 in cui viene descritto il ciclo di vita della piattaforma.

2.5.3 Esecuzione batch-script

L'attivazione della piattaforma in modalità batch avviene lanciando il driver con lo switch *–x:* seguito dal percorso che identifica il file contenente lo script batch (*asdrv –x:nome_file.abs*). Lo script batch contiene una serie di direttive e comandi che sono eseguiti automaticamente dalla piattaforma. Attualmente sono attivate le direttive conf, assembly, type e file e non possibile utilizzare alcun comando. Le direttive precedentemente citate permettono però di avviare la piattaforma e cedere il controllo ad un oggetto che interfacciandosi con essa esegue il processo batch richiesto. In effetti, il processo batch che si vuole eseguire è di fatto programmato nel tipo dell'oggetto a cui viene delegato il controllo. Vediamo più in dettaglio il significato di tali direttive:

- *conf=percorso_file_configurazione* -> questa direttiva specifica il percorso del file di configurazione che deve essere utilizzato per avviare la piattaforma
- *assembly=percorso_assembly* -> questa direttiva specifica il percorso all'assembly contenente la definizione del tipo batch che deve essere utilizzato per eseguire la piattaforma
- *type=nome_completo_del_tipo* -> definisce il fully qualified name del tipo che deve essere caricato ed istanziato dal driver della piattaforma per eseguire il processo batch che ci interessa
- *file=percorso_file_configurazione_batch* -> definisce il percorso di un eventuale file di configurazione che deve essere utilizzato per eseguire il processo batch programmato all'interno del tipo di cui sopra

Nel pacchetto AgentService è presente una cartella Apps in cui vi sono alcune demo, la cartella AuctionDemo contiene un'applicazione ad agenti che simula una semplice asta con un banditore e due partecipanti. E' possibile avviare la piattaforma in modalità batch facendo in modo che esegua direttamente questa demo. Il contenuto del file *auction.abs* che avvia tale demo è il seguente:

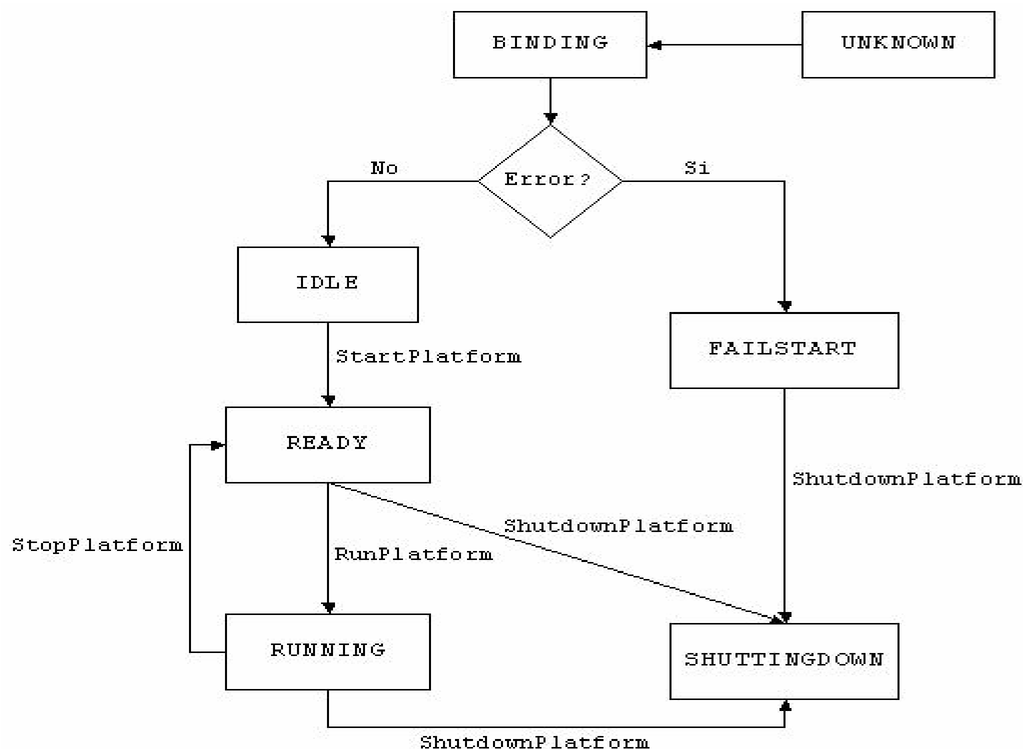
```
conf=conf\platform.conf
assembly=AuctionDemo.dll
type=AuctionDemo.AuctionBatch
```

Osserviamo che i valori dei parametri impostati prevedono che il file auction.abs sia presente nella root directory dell'installazione di AgentService e che in essa vi sia anche il file AuctionDemo.dll.

Una volta letto lo script batch la piattaforma si avvierà normalmente ed una volta terminata l'eventuale l'installazione dei moduli, cederà il controllo allo script batch.

2.6 Il ciclo di vita della piattaforma

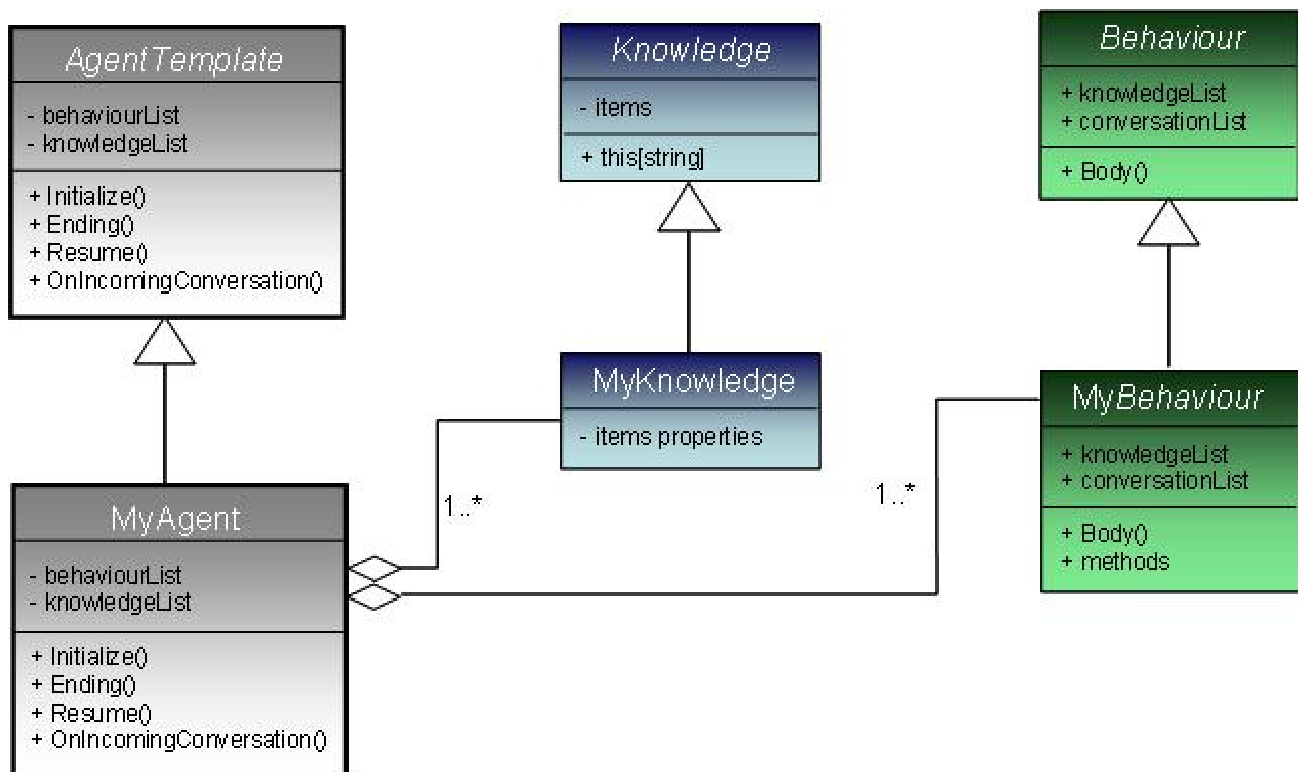
E' importante conoscere il ciclo di vita della piattaforma, in quanto il programmatore dovrà poter essere in grado di controllarlo per i propri scopi. Il ciclo di vita evolve attraverso differenti stati definiti nell'enumerazione *AgentService.Platform.PlatformState* (vedi il file: Prototype\AgentService 1.0\Platform\State.cs). Si riporta prima lo schema del ciclo di vita e si spiegano poi le operazioni effettuate nei vari stati (notare che nello schema non sono presenti tutti gli stati, ma mancano quelli transienti):



- *Unknown* -> E' lo stato iniziale della piattaforma, quando ancora non è stata svolta alcuna operazione.
- *Binding* -> E' lo stato in cui la piattaforma si porta immediatamente dopo *Unknown*: viene letto il file di configurazione della piattaforma, creato il file repository delle informazioni d'installazione e portata a termine l'installazione dei vari moduli (nel caso in cui sia il primo avvio della piattaforma) oppure recuperate le informazioni precedentemente scritte nel repository (nel caso in cui non sia il primo avvio). L'esito di queste operazioni può essere positivo o negativo.
- *FailStart* -> L'esito del binding è negativo, a causa di un file di configurazione corrotto. In questo stato la piattaforma resta in attesa, finchè non riceve il comando *ShutdownPlatform*.
- *Idle* -> Il binding è avvenuto con successo; in questo stato la piattaforma non compie nessuna operazione, attende il comando d'avvio.
- *Starting* -> E' uno stato transiente, collocato tra *Idle* e *Ready*. Viene raggiunto quando la piattaforma riceve il comando *StartPlatform*. In questo stato vengono caricati i moduli della piattaforma nel seguente ordine: per primi i core modules (Storage, Messaging, Persistence, Logging) e per ultimi i moduli addizionali.
- *Ready* -> Questo stato viene raggiunto al termine del caricamento dei vari moduli, oppure in seguito ad una richiesta di fermare la piattaforma dallo stato *Running*. La piattaforma resta in attesa di un comando (*RunPlatform* o *ShutdownPlatform*).
- *Activating* -> E' lo stato transiente tra *Ready* e *Running*. Viene raggiunto in seguito al comando *RunPlatform*. Vengono attivati i core agents (nell'ordine: AMS, MTS, DF) ed in seguito gli agenti definiti dal programmatore.
- *Running* -> La piattaforma giunge in questo stato in seguito al comando *RunPlatform*, è in questo momento che viene attivato il ciclo di vita degli agenti. A questo punto la piattaforma può essere fermata col comando *StopPlatform* ed essere riportata nello stato di *Ready*, oppure può essere chiusa direttamente tramite il comando *ShutdownPlatform*.
- *Stopping* -> E' lo stato in cui la piattaforma si trova quando riceve il comando *StopPlatform*: vengono fermati e viene applicata la persistenza prima agli user agents e poi ai core agents, infine viene fermato l'AMS. Tale stato è raggiunto anche in seguito alla chiamata al comando di *ShutdownPlatform*, in quanto è uno stato che deve essere obbligatoriamente attraversato per la chiusura della piattaforma.
- *ShuttingDown* -> Questo stato viene raggiunto quando viene invocato il metodo *ShutdownPlatform*, dopo che la piattaforma è passata nello stato di *Stopping*. Vengono eseguite tutte le operazioni per la chiusura della piattaforma: vengono scaricati i moduli addizionali, i core modules (in sequenza il logging, persistence, messaging e storage), vengono scritte le informazioni necessarie sul file di configurazione della piattaforma e infine avviene la terminazione della piattaforma.

Capitolo 3: Definizione di un agente

Un agente eredita le sue funzioni da due classi: *Agent* e *AgentTemplate*. Tale implementazione ha lo scopo di tenere separate la parte di gestione dell'attività dell'agente dalla sua definizione vera e propria.
Agent modella l'istanza di un agente in esecuzione, è totalmente trasparente all'utente del framework, viene istanziata a runtime e configurata con un opportuna classe derivante da *AgentTemplate*. *AgentTemplate* invece, rappresenta il tipo di agente che si vuole implementare: essa infatti è la classe base astratta che il programmatore utilizza per costruire il proprio agente. Il programmatore deve quindi definire una classe derivata da *AgentTemplate* (che nel seguito chiameremo *User_AgentTemplate*) che ridefinendo i metodi della classe base crea le knowledge ed i behaviour dell'agente (vedi 4.3). Analogamente le knowledge ed i behaviour di un agente definite dal programmatore devono essere delle classi derivate rispettivamente da *Knowledge* e *Behaviour*. Il class diagram UML che chiarisce le relazioni tra i tipi sopra descritti è il seguente:



In questo modo *Agent* si occuperà della gestione del ciclo di vita dell'agente, schedulandone i behaviour, controllandone l'accesso agli oggetti knowledge e mantenendo i riferimenti alla piattaforma per i servizi dell'AMS, del DF e dell'MTS, mentre *AgentTemplate* conterrà l'informazione necessaria a definire le caratteristiche dell'agente.
Per prima cosa vedremo come creare le classi knowledge (3.1); poi come creare i behaviour (3.2); ed infine come creare lo User_AgentTemplate (3.3), in cui utilizzare le knowledge ed i behaviour precedentemente definiti. Descriveremo poi l'agente vero e proprio (o almeno quella parte che è visibile al programmatore) (3.4) e vedremo infine come utilizzare quello che abbiamo fin qui definito, implementando in modalità batch l'istanziazione e l'esecuzione degli agenti (3.5).

3.1 Definizione di tipi Knowledge

La knowledge costituisce la base di conoscenza a disposizione dell'agente. Questa è composta da vari elementi che sono organizzati in un'apposita struttura dati di tipo hash-table.
Per creare una tipo knowledge è necessario definire una nuova classe che erediti dalla classe base *Knowledge*. Tale classe è contenuta nel namespace `AgentService.Agent.Knowledge` ed è definita come serializzabile, per tal motivo, tutti i membri della classe derivata dovranno essere definiti serializzabili.
La creazione di un tipo knowledge prevede la definizione di due tipi di costruttori: uno di default senza argomenti (deve essere implementato per il corretto funzionamento della piattaforma nella fase di persistenza dell'agente) ed uno con almeno due parametri: il nome della Knowledge e il modello di persistenza. Per aggiungere elementi alla knowledge bisogna utilizzare il metodo *AddItem* definito nella classe base. In aggiunta può essere definito un costruttore che inizializzi anche tutti i membri della knowledge; in tal caso la lista dei parametri specifici deve essere preceduta dai parametri definiti nel precedente costruttore.

Qui di seguito viene riportato un esempio di knowledge chiamata *Budget*, usata nel progetto AuctionDemo:

```
public class Budget:Knowledge
{
   public Budget():base(){}

   public Budget(string name,PersistenceMode pMode,int amount_object,double
     increment_object): base(name, pMode)
   {
            this.AddItem("Amount",typeof(int),amount_object);
            this.AddItem("Increment",typeof(double),increment_object);
   }
}
```

<u>Costruttore di default</u>:

```
public Budget():base(){}
```

<u>Costruttore con parametri</u>:

```
public Budget(string name,PersistenceMode pMode,int amount,double increment):
base(name, pMode)
```

i parametri obbligatori sono:

- una stringa che identifica univocamente il nome della knowledge che verrà istanziata;
- il modo di persistenza: la persistenza è definita nella classe base tramite l'enum *PersistenceMode* che può assumere 3 valori corrispondenti a 3 possibili livelli di persistenza:
  - *NotPersistence* -> le knowledge dell'agente non vengono salvati;
  - *Persistence* -> il salvataggio della base di conoscenza avviene solo su richiesta manuale (scritta esplicitamente nel codice) o in seguito alla terminazione dell'agente;
  - *StrongPersistence* -> il salvataggio avviene ad ogni modifica dei dati.

Anche in questo caso è necessario chiamare il costruttore della classe base a cui bisogna passare i due valori sopra descritti.

E' possibile aggiungere al proprio costruttore ulteriori parametri da utilizzare in fase d'istanziazione per creare knowledge dello stesso tipo, ma con valori diversi, in questo caso sono stati definiti i parametri che permettono di inizializzare l'ammontare e l'incremento del budget di ogni partecipante all'asta (questi sono ovviamente membri della knowledge *Budget*).

<u>Metodo *AddItem*</u>:

```
this.AddItem("Amount",typeof(int),amount_value);
```

Questo metodo aggiunge un elemento alla base di conoscenza dell'agente e vuole come parametri:

- una stringa che costituisce l'identificatore dell'elemento (ricordiamo che esso è memorizzato in una hash table);
- il tipo dell'elemento che intendiamo aggiungere;
- l'istanza dell'oggetto che aggiungiamo alla base di conoscenza, potrebbe anche essere nulla.

3.2 Definizione dei Behaviour

I behaviour rappresentano le differenti funzionalità che caratterizzano un agente. Ogni agente può quindi avere più behaviour, che possono condividere alcune componenti della base di conoscenza dell'agente.
Per creare un behaviour è necessario definire una nuova classe che erediti la classe *Behaviour* presente nel namespace `AgentService.Agent.Behaviour`. E' quindi necessario definire un costruttore con parametri e ridefinire il metodo *Body* della classe base che costituisce l'entry point dell'esecuzione del behaviour.
Di seguito è riportato un codice d'esempio per il behaviour *Bid* definito nel progetto AuctionDemo:

```
public class Bid : Behaviour
{
	private Budget budget;

	public Bid(string name, Budget budget):base(name, budget)
	{
		this.budget = budget;
	}

	public override void Body()
	{
		// Insert here
		// Agent activity algorithm
		...
		this.GetNewBid(price);

	}

	private int GetNewBid(int price)
	{
		this.LockKnowledge(this.budget);
		double increment = (double) this.budget["Increment",this];
		int amount = (int) this.budget["Amount",this];

		this.budget["Increment",this]=5.2;
		this.budget.Update(this);

		this.ReleaseKnowledge(true);

		int bid = (int)(price*increment);

		return bid>amount ? 0 : bid;
	}
}
```

<u>Costruttore:</u>

```
public Bid(string name, Budget budget_name):base(name, budget_name)
```

i parametri che bisogna passare al costruttore sono due:

- una stringa che identifica univocamente il behaviour;
- la lista di knowledge che il behaviour può utilizzare. Nell'esempio riportato si definisce un solo parametro knowledge nel costruttore, ma se ne potrebbero definire più di uno.

Il costruttore deve chiamare il costruttore della classe base passandogli i parametri sopra descritti.

Metodo *Body*:

```
public override void Body()
```

Questo metodo è l'entry point del behaviour, per cui nel suo corpo si deve scrivere il codice che si vuole venga eseguito quando viene lanciato il behaviour. Inoltre tale metodo deve essere ridefinito obbligatoriamente poiché nella classe base è astratto.

Accesso alla knowledge:

I behaviour devono poter accedere alle knowledge. Siccome queste sono condivise tra tutti i behaviour di un agente, deve essere garantita la consistenza dei dati. A questo scopo è implementato un apposito algoritmo: ogni behaviour, prima di poter utilizzare una knowledge, deve farne il lock in maniera tale da bloccarne l'accesso. Successivamente, quando il behaviour termina l'utilizzo di quella knowledge, deve rilasciarla, sbloccandola.
Per implementare l'algoritmo vengono utilizzate due funzioni che devono sempre essere chiamate prima (*LockKnowledge*) e dopo (*ReleaseKnowledge*) l'accesso ad una knowledge:

- `this.LockKnowledge(this.budget_name); ->` la funzione vuole come parametri una lista delle knowledge di cui si vuole acquisire un accesso esclusivo.
- `this.ReleaseKnowledge(true);` -> la funzione vuole un parametro di tipo bool: il programmatore dovrebbe sempre passargli il valore true. Il valore false infatti fa sì che un'eventuale eccezione non venga gestita e quindi non viene normalmente usato.

E' importante sapere che non è possibile effettuare lock annidati, pena la generazione di un'eccezione: bisogna effettuare un unico lock a cui passare tutte le knowledge necessarie.
L'esecuzione del metodo *LockKnowledge* comporta anche un altro effetto: viene creata una copia temporanea delle knowledge acquisite (che chiameremo per comodità temp_knowledge), in questo modo tutti i cambiamenti effettuati dall'utente avvengono su temp_knowledge senza interessare la knowledge originale. Il behaviour quindi mantiene un riferimento alla temp_knowledge e non alla knowledge originale: tale realizzazione ha lo scopo di mantenere il più possibile l'isolamento dei dati.

Successivamente quando il behaviour chiama la funzione *ReleaseKnowledge*, la sua esecuzione provoca l'eliminazione della temp_knowledge, in modo tale da impedire al behaviour di mantenere un riferimento alla knowledge. Questo però rende necessario copiare i dati cambiati in temp_knowledge nella knowledge originale: questa operazione viene effettuata dal metodo *Update.*

`this.budget.Update(this);` -> è un metodo della classe *Knowledge e* vuole come parametro il behaviour che ha richiesto il lock della knowledge. Poiché viene usato all'interno del behaviour che ha lockato la knowledge gli si passa *this*.

Una volta effettuato l'accesso alla knowledge, si presenta un ulteriore problema. Infatti le knowledge di un agente sono formate da vari elementi. Esse vengono memorizzate in un vettore nella classe base *Knowledge* e ciascuna è organizzata come una hash table: accedere ogni volta a tale vettore è molto scomodo. Per ovviare al problema si consiglia, all'atto della creazione del behaviour, di creare un riferimento alle knowledge utilizzate, come mostrato nell'esempio (all'interno del costruttore):

```
private Budget budget;

public Bid(string name, Budget budget_name):base(name, budget_name)
            {
                  this.budget = budget_name;
            }
```

A questo punto per accedere ad un elemento all'interno della knowledge, questa viene chiamata attraverso l'indicizzazione diretta della hash table:

```
double increment = (double) this.budget["Increment",this];
```

Come parametri, vuole l'identificatore dell'elemento e il nome del behaviour che ne fa richiesta (*this*, cioè io, il behaviour). Da notare che deve essere effettuato un cast all'atto dell'estrazione dell'elemento, perchè è memorizzato genericamente come un tipo object.
La sintassi per scrivere un elemento nella knowledge è analoga:

```
this.budget["Increment",this]=5.2;
```

Si ricorda la necessità di chiamare *Update* per rendere effettivi in memoria i cambiamenti effettuati.

<u>L'interfaccia Runtime</u>:

La classe base Behaviour mette a disposizione della classe behaviour definita dal programmatore una serie di metodi per l'accesso a tutti i servizi della piattaforma. Tali metodi sono contenuti nella proprietà Runtime e verranno analizzati in seguito. (Vedi Capitolo 4:).

3.3 Definizione di AgentTemplate

Per definire un agente, il programmatore deve definire una classe derivata dalla classe astratta *AgentTemplate*, a cui l'agente istanziato dalla piattaforma mantiene un riferimento e nella quale vengono descritte tutte le caratteristiche dell'agente stesso. Tale classe è definita nel namespace `AgentService.Agent`.
Questa classe deve ridefinire i metodi astratti della classe base: *Initialize* e *Finalize*. Può inoltre ridefinire (non obbligatoriamente) altri due metodi: *Resume* e *HandleConversation* che di default non svolgono alcuna funzione (sono definiti virtuali). Di seguito è riportato il codice d'esempio per l'AgentTemplate *Bidder* definito in AuctionDemo:

```
public class Bidder:AgentTemplate
{
	public Bidder():base(){}

	public override void Initialize(AgentService.Agent.Runtime.ITypeContext
	typeContext, AgentService.Agent.Runtime.IFactoryContext factoryContext,
	string configFile)
	{
		typeContext.RegisterKnowledgeTypes(typeof(Budget));
		typeContext.RegisterBehaviourTypes(typeof(Bid));

		factoryContext.CreateKnowledge(typeof(Budget),
				"Budget",PersistenceMode.NotPersistent,50,2);
		factoryContext.CreateBehaviour(typeof(Bid),"Bid","Budget");
	}

	public override void Finalize(){}
}
```

<u>Costruttore</u>:

```
public Bidder():base()
```

Il costruttore non contiene normalmente del codice di inizializzazione per il template ma questo è spostato nel metodo *Initialize()*.

<u>Metodo Initialize:</u>

```
public override void Initialize(AgentService.Agent.Runtime.ItypeContext
typeContext, AgentService.Agent.Runtime.IFactoryContext factoryContext,
string configFile)
```

Questo metodo viene chiamato dalla piattaforma all'atto della creazione dell'agente, definisce ed inizializza le knowledge ed i behaviour utilizzati dall'agente. A questo metodo vengono passati:

- un oggetto di tipo *ITypeContext*: è un riferimento che verrà utilizzato nel corpo del metodo per la registrazione dei tipi di knowledge e behaviour;

- un oggetto di tipo *IFactoryContext:* è un riferimento che verrà utilizzato nel corpo del metodo per l'istanziazione di knowledge e behaviour;
- una stringa che identifica il nome di un file di configurazione, incluso il path del file. Il file può essere usato per passare informazioni aggiuntive a *Initialize()*. Non è infatti possibile passare parametri a proprio piacimento. Ovviamente sarà necessario definire anche il modo con cui accedere al file e le operazioni da compiere con le informazioni così recuperate.

E' importante notare che questa è la ridefinizione di un metodo, quindi non bisogna preoccuparsi di passargli gli oggetti appena visti. Questa operazione verrà effettuata quando il metodo sarà invocato dall'infrastruttura della piattaforma all'atto della creazione dell'agente.

Nel corpo del metodo bisogna definire quali saranno le knowledge ed i behaviour utilizzati dall'agente. Per farlo è prima necessario registrare i tipi di behaviour o knowledge che saranno utilizzati, utilizzando i metodi di *ITypeContext*:

```
typeContext.RegisterKnowledgeTypes(typeof(Budget));
typeContext.RegisterBehaviourTypes(typeof(Bid));
```

Il parametro richiesto è il tipo di Knowledge o Behaviour (definiti dall'utente) che si vogliono creare. Tale coppia di metodi registra i tipi passati come parametri come i tipi da cui è possibile istanziare oggetti. Essendo i parametri dei due metodi di tipo *params* è possibile invocarli con una lista parametri variabile.

L'operazione successiva è quella di istanziare le knowledge ed i behaviour precedentemente dichiarati. Per quanto riguarda le knowledge viene utilizzato il seguente metodo di *IFactoryContext*:

```
factoryContext.CreateKnowledge (typeof(Budget), "Budget", PersistenceMode.NotPersistent, 50, 2);
```

CreateKnowledge vuole come parametri il tipo di knowledge che si vuole istanziare, una stringa che rappresenta l'identificativo di tale knowledge, il modo di persistenza (vedi 3.1) ed eventuali ulteriori parametri che si vogliono passare al costruttore della knowledge. Il numero di parametri di quest'ultimo tipo che possono essere passati non è definito a priori, ma viene lasciata libertà al programmatore (a questo scopo la realizzazione è effettuata tramite un *params* di *object*).

Per quanto riguarda i behaviour viene utilizzato il seguente metodo di *IFactoryContext*:

```
factoryContext.CreateBehaviour(typeof(Bid),"Bid","Budget");
```

I parametri necessari sono: il tipo di behaviour che si vuole istanziare, la stringa identificativa del behaviour, i nomi delle knowledge che saranno utilizzate dal behaviour. Quest'ultimo campo è stato realizzato tramite un *params*, in maniera tale può essere

accettato un numero variabile di parametri, a seconda delle knowledge richieste dal behaviour.

Metodo Finalize:

```
public override void Finalize(){}
```

Nel corpo del metodo deve essere contenuto il codice che si vuole venga eseguito prima che l'agente termini la sua attività, ossia venga distrutto dalla piattaforma.

Metodo Resume:

```
public override void Resume(AgentService.Agent.Runtime.ITypeContext typeContext, AgentService.Agent.Runtime.IFactoryContext factoryContext)
```

Tale metodo viene invocato dalla piattaforma al suo riavvio dopo un'eventuale interruzione, che può anche avvenire ad esempio per un crash di sistema. Al riavvio, la piattaforma ricarica tutti gli agenti, successivamente lancia i rispettivi metodi *Resume*. All'interno del corpo di tale metodo va quindi posto il codice che si vuole venga eseguito in tali circostanze, ad esempio per ripristinare una knowledge o riavviare un behaviour.

Metodo HandleConversation:

```
public override void HandleConversation(IAgentContext context,
AgentService.Agent.Runtime.ConversationArgs args)
{
	args.Conversation.Accept();
	context.AddConversation("ManageAuction",args.Conversation);
}
```

Questo metodo viene invocato dalla piattaforma ogni volta in cui l'agente riceve una richiesta di apertura di una conversazione: se non viene ridefinito, il comportamento di default è quello di rifiutare ogni richiesta. Tale metodo verrà analizzato nello specifico in 4.2.4.3.

3.4 L'agente – AID (AgentIdentifier)

Come detto in precedenza, l'agente vero e proprio, rappresentato dalla classe *Agent*, viene gestito dalla piattaforma e risulta completamente trasparente al programmatore, che si occuperà di progettare solamente l'*AgentTemplate*. Facendo un paragone con la programmazione classica ad oggetti, l'*AgentTemplate* sta ad *Agent* come un oggetto sta alla sua istanza (di essa si ha un nome univoco che la caratterizza).
In maniera simile, il programmatore dell'agente si riferisce ad esso solo attraverso il suo identificativo : AID (AgentIdentifier).

L'AID è un'identificatore univoco globale associato ad un dato agente. E' composto da:

- nome simbolico dell'agente
- nome simbolico della piattaforma che ospita l'agente
- nome globale nella forma: *[nome_agente]@[nome_piattaforma]*
- una lista degli indirizzi a cui l'agente può essere contattato
- una lista di indirizzi di resolvers, che sanno come contattare l'agente ed inoltrare ad esso i messaggi che ricevono

L'AID è stato realizzato come una classe, definita nel namespace `AgentService.Agent`, che mette a disposizione una serie di proprietà per l'accesso ai dati membro (accesso solo in lettura e non in scrittura), un certo numero di costruttori ed un metodo per la comparazione di due AID.

Proprietà per l'accesso in lettura ai dati membro:

- `string Name` -> nome dell'agente
- `string GlobalName` -> nome globale
- `string HapName` -> nome della piattaforma proprietaria dell'agente
- `System.Collections.ArrayList Addresses` -> indirizzi ai quali un agente può essere contattato
- `System.Collections.ArrayList Resolvers` -> resolvers di un agente

3.5 Istanziazione ed esecuzione degli agenti in modalità batch

Una volta definiti gli agenti come sopra descritto, l'ultimo passo è quello della loro istanziazione ed esecuzione da parte della piattaforma.
La piattaforma può funzionare in vari modi: ad esempio può comportarsi come un server ed attendere di ricevere dei comandi (utilizzando dei comandi testuali con lo standard input o interfacciandosi attraverso un canale di remoting) oppure può utilizzare la modalità batch. Nel seguito vedremo come realizzare quest'ultima.

Per lanciare la modalità batch si utilizza il comando:

asdrv –x:nome_file.abs

In questa modalità la piattaforma viene lanciata ed esegue uno script (vedere 2.5.3). Come già visto, nel caso del progetto AuctionDemo lo script contiene:

conf=conf\platform.conf -> path file di configurazione
assembly=AuctionDemo.dll -> nome dell'assembly da caricare

type=AuctionDemo.AuctionBatch -> istanza della classe che esegue il batch

E' cura del programmatore definire le azioni che verranno intraprese dalla piattaforma: a questo scopo il programmatore deve creare una classe che implementa l'interfaccia *IPlatformBatch* (nell'esempio AuctionBatch, che è contenuta nella dll AuctionDemo) in cui definire il comportamento voluto della piattaforma.
Vediamo la struttura dell'interfaccia *IPlatformBatch* contenuta nel namespace *AgentService.Platform*:

```
namespace AgentService.Platform
{
	public interface IPlatformBatch
	{
		void Run(PlatformController controller);
		void Run(PlatformController controller, string configFile);
	}
}
```

Come si può vedere questa contiene un solo metodo: *Run* al quale viene passato un oggetto *PlatformController*. Opzionalmente è possibile passargli il path di un file, che può essere utilizzato per trasmettere al metodo parametri addizionali contenuti all'interno del file.
Tramite l'oggetto *PlatformController* (vedi 5.1) è possibile controllare il ciclo di vita della piattaforma (vedi 2.6), istanziare ed attivare gli agenti.
Affinchè la classe AuctionBatch funzioni correttamente è necessario che vengano inclusi almeno i seguenti namespace:

- AgentService.Platform
- AgentService.Platform.Configuration
- AgentService.Platform.Modules

Viene di seguito riportato il codice d'esempio di AuctionBatch:

```
namespace AuctionDemo
{
	public class AuctionBatch: AgentService.Platform.IPlatformBatch
	{
		public void Run(AgentService.Platform.PlatformController controller)
		{
			System.Console.WriteLine("Press [Return] to start...");
			System.Console.ReadLine();
			controller.StartPlatform();

			if (controller.PlatformState == PlatformState.Ready)
			{
				ModuleError error =
					controller.Modules.UpLoadAssembly("AuctionDemo.dll",true
				);
				if (error == null)
				{
					controller.RunPlatform();
					if (controller.PlatformState == PlatformState.Running)
```

```
                        {
controller.Services.CreateAgent("AuctionDemo.Auctioneer","Magalli","",false);
controller.Services.CreateAgent("AuctionDemo.Bidder","Tinto","",false);
controller.Services.CreateAgent("AuctionDemo.Bidder2","Riccardo","",false);
controller.Services.ActivateAllAgents();
                        }
                        else
                        {
                              Console.Out.WriteLine("error");
                        }
                  }
                  else
                  {
                        Console.Out.WriteLine("error: {0}",error.Description);
                  }
            }
            System.Threading.Thread.Sleep(2000);
            Console.ReadLine();
            controller.ShutdownPlatform();
      }

      public void Run(AgentService.Platform.PlatformController controller,
                              string configFile)
      {
            // we do not need config files
            this.Run(controller);
      }
   }
}
```

Quando viene lanciata in modalità batch, la piattaforma, una volta raggiunto lo stato di idle, invoca il metodo *Run* della classe, definita dal programmatore, indicata nel file *.abs*. E' compito del programmatore controllare sempre lo stato in cui si trova la piattaforma e poter quindi richiedere correttamente le operazioni.
Vediamo la sequenza delle principali operazioni normalmente svolte all'interno del *Batch*, come evidenziato dall'esempio sopra indicato:

- Si avvia la piattaforma con il comando S*tartPlatform* (così facendo si giunge nello stato *Ready*)
- Si controlla di essere giunti nello stato *Ready*.
- Se così è, è possibile caricare gli assembly (contenenti i template degli agenti desiderati) nello storage.
- Se così facendo non viene generato alcun errore, si esegue la piattaforma con il comando *RunPlatform*.
- Si controlla di essere nello stato Running.
- A questo punto è possibile creare gli agenti attraverso il metodo *CreateAgent*; con la creazione, gli agenti vengono registrati nell'AMS, nel DF e nell'MTS (creando una coda dei messaggi per loro) ma non sono ancora realmente partiti.
- Si avviano quindi gli agenti con *ActivateAllAgents:* partono quindi tutti i thread associati ai loro behaviour.

- E' possibile quindi chiudere la piattaforma con il metodo *ShutdownPlatform*. Si ricorda che la chiusura della piattaforma può essere eseguita sia che ci si trovi nello stato Ready che in Running.

Notare che devono essere ridefiniti entrambi i metodi Run: nell'esempio non viene utilizzato alcun file di configurazione.

3.5.1 La classe ModuleError

Tale classe viene utilizzata per contenere tutte le informazioni riguardanti un errore generato da un modulo. E' contenuta nel namespace *AgentService.Platform.Modules*. E' composta da:

- un reference al modulo che ha generato l'errore
- l'eccezione che ha generato l'errore
- una descrizione dell'errore (come string)
- un array dei messaggi relativi all'errore

Tali dati membro sono accessibili tramite le rispettive proprietà:

- `IPlatformModule OriginatingModule`
- `Exception Exception`
- `string Description`
- `string[] Messages`

Questo è quanto il programmatore necessita di conoscere di questa classe, in quanto la utilizzerà solo per ottenere informazioni circa la generazione di errori.

Capitolo 4: Il comportamento a runtime

In questo capitolo analizzeremo i servizi che la piattaforma fornisce ai vari agenti in fase di runtime: questi sono contenuti nell'interfaccia *IRuntime* definita nel namespace `AgentService.Agent.Runtime`.
Quando un agente viene creato, viene creato anche un oggetto *Runtime* che implementa l'interfaccia *IRuntime*. Ogni agente possiede quindi un suo oggetto Runtime. In seguito ogni volta in cui viene creato un behaviour, la piattaforma invoca un particolare metodo che crea un clone di IRuntime.
In tal modo dall'interno di ciascun behaviour è possibile accedere ai servizi offerti dalla piattaforma:

- Scrittura su console
- Servizio di messaggistica
- Servizio di logging
- Servizio di pagine bianche
- Servizio di pagine gialle
- Servizio di persistenza
- Servizio di controllo dei behaviour

L'interfaccia *IRuntime* è così definita:

```
namespace AgentService.Agent.Runtime
{
	public interface IRuntime
	{
		// Messaging Service (Sends, retrieves messages and conversation
		// management).
		IMsgClient MessageSvc
		{ get; }

		// Persistence Service (Persists and resumes Knowledge objects).
		IPersistenceClient PersistenceSvc
		{ get; }

		// Logging Service (Gives access to the agent logging subsystem).
		ILogClient LogSvc
		{ get; }

		// Yellow Pages Service (Searches agent's aid by service).
		IYellowPagesClient YPageSvc
		{ get; }
```

```
            // White Pages Service (Searches agent's aid by name).
            IWhitePagesClient WPageSvc
            { get; }

            // Gets a reference to the IBehaviourContext interface
            IBehaviourContext Context
            { get; }

            // Gets a reference to the platform console, used by all
            // the agents to append messages to the platform console
            IConsoleProxy Console
            { get; }
      }
}
```

Come si può vedere l'interfaccia è costituita da una serie di proprietà che permettono di accedere ad altrettante interfacce che forniscono i servizi desiderati. Nel seguito le analizzeremo una ad una.

4.1 Servizio di scrittura su Console – IConsoleProxy

IConsoleProxy offre il servizio di scrittura sulla Console della piattaforma comune a tutti gli agenti.

```
namespace AgentService.Agent.Runtime
{
       public interface IConsoleProxy
       {
             void AppendMessage(string message, params object[] args);
             void AppendMessage(string message);
       }
}
```

Contiene solo il metodo *AppendMessage*, che ha la stessa sintassi utilizzata dal Writeline di .NET.

Tale metodo è quindi utilizzabile all'interno di un behaviour di un agente, un esempio è il seguente:

```
this.Runtime.Console.AppendMessage("waiting for auction data: {0}", auction_data);
```

4.2 Servizio di messaggistica – IMsgClient

La proprietà *MessageSvc* dell'interfaccia *IRuntime* fornisce accesso al servizio di messaggistica della piattaforma. Ogni behaviour, tramite l'interfaccia di accesso al

runtime, dispone di un client del servizio di messaggistica le cui funzionalità sono definite dall'interfaccia *ImsgClient*. Questa è la sua definizione:

```
namespace AgentService.Agent.Runtime
{
	public interface IMsgClient
	{
		AgentDelegate GetStopAgentDelegate();
		event ConversationDelegate OnIncomingConversation;
		IConversation OpenConversation(AID to);
		IOntologicalConversation OpenConversation(AID to, Ontology.Ontology onto);
		bool HasMessages();
		bool SendMessage(AgentMessage message);
		AgentMessage ReceiveMessage();
		AgentMessageCollection ReceiveMessage(MessageFilter filter,bool blocking);
		AgentMessage PeekMessage();
		AgentMessageCollection GetAllMessages(bool bRemove);
		AgentMessage WaitForMessage();
		bool RemoveMessage(AgentMessage message);
	}
}
```

Gli agenti vengono creati in maniera tale da realizzare il più possibile l'isolamento dall'ambiente circostante; per tale motivo, l'unico modo che hanno di comunicare tra loro è scambiandosi messaggi.
I messaggi sono composti da un *body* (il corpo contente il messaggio vero e proprio) e da un'*envelope* (contente le informazioni di servizio per l'invio e la ricezione del messaggio). Ogni messaggio deve sempre contenere il destinatario a cui essere inviato ed ogni agente possiede un buffer nel quale mette in coda tutti i messaggi ricevuti. Esistono inoltre vari metodi che permettono la lettura e la cancellazione dei messaggi.

Una forma più evoluta di messaggistica è la conversazione che implementa un servizio di comunicazione connesso[1] tra due agenti. Una conversazione può essere intesa come "privata" fra due agenti: solo mittente e destinatario possono accedere ai messaggi in essa contenuti. Per questa sua caratteristica una conversazione deve essere instaurata prima di poter essere utilizzata e deve essere rimossa al termine del suo utilizzo. La gestione delle conversazioni e la loro implementazione dipendono dal modulo del servizio di messaggistica il quale si occupa di fornire alla piattaforma l'opportuno oggetto che implementa l'interfaccia *IMsgClient*, da cui si possono aprire delle conversazioni.

Si ricorda che, in base al modello di agente utilizzato in AgentService, la logica funzionale di un agente è definita all'interno dei behaviour che lo costituiscono. Per tal

[1] Con servizio connesso si intende in questo caso un servizio di scambio messaggi in cui vi è una fase iniziale di setup di un canale virtuale all'interno del quale verranno spediti i messaggi relativi alla conversazione. L'invio dei messaggi appartenenti ad una conversazione non richiede di specificare per ogni messaggio mittente e destinatario, ma tale operazione è effettuate durante l'instaurazione del canale virtuale. Possiamo paragonare il servizio di messaggistica senza conversazioni all'utilizzo del protocollo UDP: in tal caso il servizio di messaggistica fornito dalle conversazioni potrebbe essere paragonato al modello offerto dal TCP.

motivo, sebbene gli end-point di una comunicazione siano due agenti in realtà la comunicazione avviene sempre tra due behaviour.

Analizziamo per prima cosa la struttura di un messaggio (4.2.1); vediamo poi come utilizzare IMsgClient per la comunicazione tra gli agenti tramite semplici messaggi (4.2.2) o attraverso una conversazione (4.2.4).

4.2.1 Struttura di un messaggio – AgentMessage

Un messaggio è formato da due parti principali: l'envelope del messaggio ed il corpo (body) del messaggio. Vi è poi un identificativo che indica la conversazione a cui tale messaggio appartiene. Vediamo prima la struttura di *MessageEnvelope*, analizziamo poi *MessageBody* ed infine la classe *AgentMessage* (che comprende ovviamente le prime due). Queste classi sono contenute nel namespace *AgentService.Platform.MTS*.

4.2.1.1 MessageEnvelope

MessageEnvelope definisce la busta di un messaggio. In essa sono contenute le seguenti informazioni:

- destinatari del messaggio
- mittente del messaggio
- eventuale commento al messaggio
- informazioni riguardanti il tipo di rappresentazione sintattica utilizzata nel corpo del messaggio (chiamata ACL)
- la data e l'ora della creazione del messaggio (da parte del mittente)
- informazioni riguardanti come è stato crittografato il corpo del messaggio
- il nome degli agenti a cui deve essere inviato il messaggio
- il tipo di trasporto necessario per il messaggio

Proprietà per l'accesso in lettura dei dati membro:

- `AIDCollection To;`
  restituisce un oggetto *AIDCollection* contenente tutti i destinatari del messaggio. *AIDCollection* è strutturata come una normale collezione di .NET, per cui si considera noto il suo utilizzo.

- `AID From;`
  restituisce l'AID del mittente del messaggio.

- `StringCollection Comments;`
  restituisce la collezione di commenti attaccati al messaggio.

- `string AclRepresentation;`
  restituisce l'ACL del messaggio, cioè il tipo di rappresentazione sintattica utilizzata nel corpo del messaggio.

- `System.DateTime Date;`
  restituisce data e ora della creazione del messaggio.

- `System.Collections.ArrayList Encrypted;`
  restituisce il tipo di codifica utilizzata nel corpo del messaggio.

- `AIDCollection IntendedReceivers;`
  restituisce la collezione di AID indicante i destinatari "ultimi": è infatti possibile mandare un messaggio ad un agente per far sì che questo lo recapiti ad un altro. In realtà, comunque, la piattaforma controlla sempre il dato *To* per vedere a chi inviare i messaggi e gestisce poi in maniera autonoma l'inoltro di messaggi.

- `string TransportBehaviour;`
  restituisce, se presenti, i requisiti di trasporto del messaggio.

Costruttori:

- `public MessageEnvelope()`
  Costruttore di default che costruisce un oggetto MessageEnvelope vuoto, con tutti i campi iniziati a null.

- `public MessageEnvelope(string acl,params AID[] receivers)`
  Costruisce un oggetto MessageEnvelope con i parametri passatigli: l'ACL e i destinatari primari del messaggio (quelli recuperabili con la proprietà *To*).

4.2.1.2 MessageBody

Questa classe definisce il corpo di un messaggio, costituito da una collezione di body_item che rappresenta l'informazione che l'agente vuole trasmettere. Tale collezione è organizzata come una hash table: ne deriva la necessità di specificare sempre un identificativo univoco per ogni body_item.

Costruttore:

- `public MessageBody()`
  Costruisce un oggetto messageBody creando una nuova collezione di body_item.

Metodi:

- `public void Add(string itemId, object instance);`
  Aggiunge un nuovo body_item alla collezione. Come parametri vuole un identificatore dell'item e l'oggetto da aggiungere.

- `public void Remove(string itemId);`
  Rimuove l'elemento caratterizzato dall'identificatore passatogli per parametro dalla collezione.

- `public object this[string itemId];`
  Questo metodo è un indexer: può prelevare un elemento dalla collezione se utilizzato alla destra dell'operatore di assegnazione od impostare il valore di un elemento della collezione se usato alla sinistra dell'operatore di assegnazione.

4.2.1.3 **AgentMessage**

La classe *AgentMessage* definisce un messaggio e si compone di:

- una busta accessibile tramite la proprietà *Envelope*;
- un contenuto accessibile tramite la proprietà *Body*;
- un identificatore di conversazione *conversationId*.

Proprietà per l'accesso ai dati membro:

- `MessageEnvelope Envelope;`
  Restituisce un oggetto *MessageEnvelope* che costituisce l'envelope del messaggio (4.2.1.1).

- `MessageBody Body;`
  Restituisce un oggetto *MessageBody* che costituisce il corpo del messaggio (4.2.1.2).

- `int ConversationId;`
  Restituisce l'identificativo della conversazione a cui il messaggio appartiene. Tale identificativo deve essere un numero positivo se il messaggio appartiene ad una conversazione. Deve valere 0 se è invece un messaggio singolo, -1 se è un messaggio di servizio (tali messaggi vengono usati dalla piattaforma per l'instaurazione delle conversazioni e non devono essere usati dal programmatore). Per facilitare l'utilizzo è stato creato l'enum MessageType:

  ```
  public enum MessageType
  {
  ```

```
        ServiceMessage = -1,
        SimpleMessage = 0
}
```

Costruttori:

- `public AgentMessage()`
  Costruttore senza parametri di *AgentMessage*. Chiama i costruttori di default di *MessageBody* e di *MessageEnvelope*. Imposta il conversationId come *MessageType.SimpleMessage*.

- `public AgentMessage(string acl,MessageBody body,params AID[] receivers)`
  Costruisce un oggetto *AgentMessage* con i parametri specificati. Chiama il costruttore di *MessageEnvelope* al quale passa l'ACL e i receivers. Pone il body dell'*AgentMessage* uguale al corrispondente parametro e inizializza il conversationId come *MessageType.SimpleMessage*.

- `public AgentMessage(MessageEnvelope envelope,MessageBody body,int conversationId)`
  Costruisce un oggetto *AgentMessage*, inizializzando i suoi dati membro con i parametri passati.

Metodi:

- `public AgentMessage Reply(bool copyBody);`
  Crea un messaggio di risposta al mittente. Si usa il parametro true se si vuole copiare il corpo del messaggio originale nella risposta, false altrimenti.

- `public AgentMessage Reply(bool copyBody,AID aid);`
  Crea un messaggio di risposta che invia al mittente e a tutti gli altri destinatari del messaggio. Oltre al parametro booleano che indica se copiare il testo del messaggio originale, l'altro parametro deve essere l'AID dell'agente che sta chiamando il metodo *Reply*: questo perchè viene estrapolata dal messaggio ricevuto la lista dei receivers, dalla quale è necessario togliere il proprio identificativo (per non rimandare il messaggio a se stessi!).

- `public override bool Equals(object obj);`
  Ridefinisce il metodo *Equals* confrontando due istanze di *AgentMessage* sulla base della data e ora presente nell'envelope. Il metodo va quindi a confrontare l'hash code delle due istanze delle envelope.

4.2.2 Comunicazione basata su semplice scambio di messaggi

Le operazioni base per il sistema di messaggistica sono utilizzabili attraverso l'interfaccia *IMsgClient*.

Metodi:

- `bool HasMessages();`
  Restituisce *true* se sono presenti messaggi in coda, false altrimenti.

- `bool SendMessage(AgentMessage message);`
  Manda un messaggio ad un agente. Restituisce *true* se il messaggio è stato inviato correttamente.

- `AgentMessage ReceiveMessage();`
  Restituisce un AgentMessage estraendo il primo messaggio presente in coda.

- `AgentMessageCollection ReceiveMessage(MessageFilter filter, bool blocking);`
  Restituisce una collezione di messaggi che soddisfano i criteri specificati nell'oggetto filter passato come parametro. E' cura del programmatore creare una classe derivata da *MessageFilter* nella quale specificare i criteri di filtraggio da applicare alla coda di messaggi. Il secondo parametro indica se il metodo debba essere bloccante o meno.

- `AgentMessage PeekMessage();`
  Legge il contenuto del primo messaggio in coda senza rimuoverlo dalla stessa.

- `AgentMessageCollection GetAllMessages(bool bRemove);`
  Restituisce una collezione di AgentMessage contenente tutti i messaggi presenti in coda. Se il parametro bRemove è posto a *true* i messaggi vengono anche rimossi dalla coda.

- `AgentMessage WaitForMessage();`
  Attende che un nuovo messaggio giunga nella coda dell'agente e lo estrae.

- `bool RemoveMessage(AgentMessage message);`
  Rimuove il messaggio specificato tramite parametro dalla coda dell'agente. Restituisce *true* se la rimozione è andata a buon fine.

4.2.3 MessageFilter

L'utilizzo dei messaggi da parte degli agenti è di fondamentale importanza per la comunicazione con i loro simili; può ad esempio esistere l'eventualità che un agente non voglia ricevere i messaggi spediti da un certo mittente, oppure che sia interessato a ricevere messaggi spediti solamente in un determinato intervallo di tempo e così via. Questi filtri possono essere sviluppati poiché ogni messaggio contiene nell'envelope una serie di dati (tra i quali il mittente e la data) sulla base dei quali può avvenire una selezione.

E' nella fase di ricezione dei messaggi che entra in gioco l'utilità dei filtri: in particolare utilizzando la funzione *ReceiveMessage(MessageFilter filter, bool blocking)* viene restituita in fase di ricezione una lista di messaggi che soddisfano la condizione espressa nel filtro *filter* passato appunto come parametro alla funzione stessa.

4.2.3.1 Classe MessageFilter

La classe base MessageFilter fornisce le principali funzionalità comuni a tutti i filtri; in particolare riportiamo di seguito i metodi in essa contenuti con una breve descrizione di come vengano utilizzati:

- ```
  public delegate bool FilterDelegate(AgentMessage message)
  ```
  Non esistendo i puntatori a funzione nel linguaggio C# viene utilizzato il meccanismo dei *delegate*; questo meccanismo permette di associare ad un oggetto una funzione con una signatura specifica: in questa maniera è possibile creare funzioni diverse associabili allo stesso oggetto. *FilterDelegate* prende come parametro un messaggio e restituisce vero se il messaggio rispetta i criteri del filtro.

- ```
  private FilterDelegate filter;
  ```
  Il delegate filter è il membro della classe che conterrà il metodo che esegue il filtraggio dei messaggi.

- ```
  public MessageFilter()
  {
        filter = new FilterDelegate(this.FilterTrue);
  }
  ```
  È il costruttore di default della classe base all'interno del quale viene inizializzato l'oggetto filter con nuovo oggetto di tipo delegate che punta alla funzione d'appoggio *FilterTrue*. Un'istanza creata col costruttore di base lascerà passare tutti i messaggi.

- ```
  private bool FilterTrue(AgentMessage message)
  {
        return true;
  }
  ```
  Delegate che non effettua alcun filtraggio impostato come valore di default del membro filter. La classe *MessageFilter* non effettua alcun filtraggio per cui è

necessario reimplementare nelle classi derivate i suoi metodi in modo tale da realizzare le modalità specifiche del filtro desiderato.

4.2.3.2 Filtri derivati

Per fornire alcune funzionalità di base sono stati implementati una serie di filtri per la ricezione di utilizzo comune; questi filtri sono contenuti nel progetto *MessageFilterLibrary* e sono i seguenti:

- *Filtro* `MessageFilter_From(AID from)`
  Permette di ricevere solamente i messaggi spediti da un determinato mittente specificato dall'AID from.

- *Filtro* `MessageFilter_BlackList()`
  Consente di specificare una lista di mittenti "indesiderati" dai quali non si vogliono ricevere messaggi; inoltre questo filtro contiene due metodi

  ```
  void PutInBlackList(AID from)
  void RemoveFromBlackList(AID from)
  ```

  per l'inserimento e la rimozione degli AID dalla BlackList.

- *Filtro* `MessageFilter_Date_Before(DateTime Prima_Data_Utile)`
  Scarta i messaggi spediti dopo una certa data (Prima_Data_Utile).

- *Filtro* `MessageFilter_Date_After(DateTime Prima_Data_Utile)`
  Scarta i messaggi spediti prima di una certa data (Prima_Data_Utile).

- *Filtro* `MessageFilter_Compound(MessageFilter a, MessageFilter b,bool PredicateType)`
  Consente di sviluppare filtri più complessi utilizzando i filtri già implementati. Il meccanismo di base accetta come parametri due filtri e l'operazione logica che si vuole effettuare su di essi. Un messaggio che soddisfi la condizione logica (AND, OR) viene accettato.

4.2.3.3 Utilizzo pratico dei filtri

Viene di seguito riportato un esempio di utilizzo dei filtri per la ricezione di messaggi: in questo caso viene usato il filtro *MessageFilter_From* che permette la ricezione solamente di quei messaggi spediti dall'agente identificato con l'AID "DataAfter". Per istanziare il filtro è necessario conoscere l'AID dell'unico mittente dal quale si vuole ricevere.

```
AID from = new AID("DateAfter",this.Owner.HapName);
```

```
filtro = new MessageFilter_From(from);
```

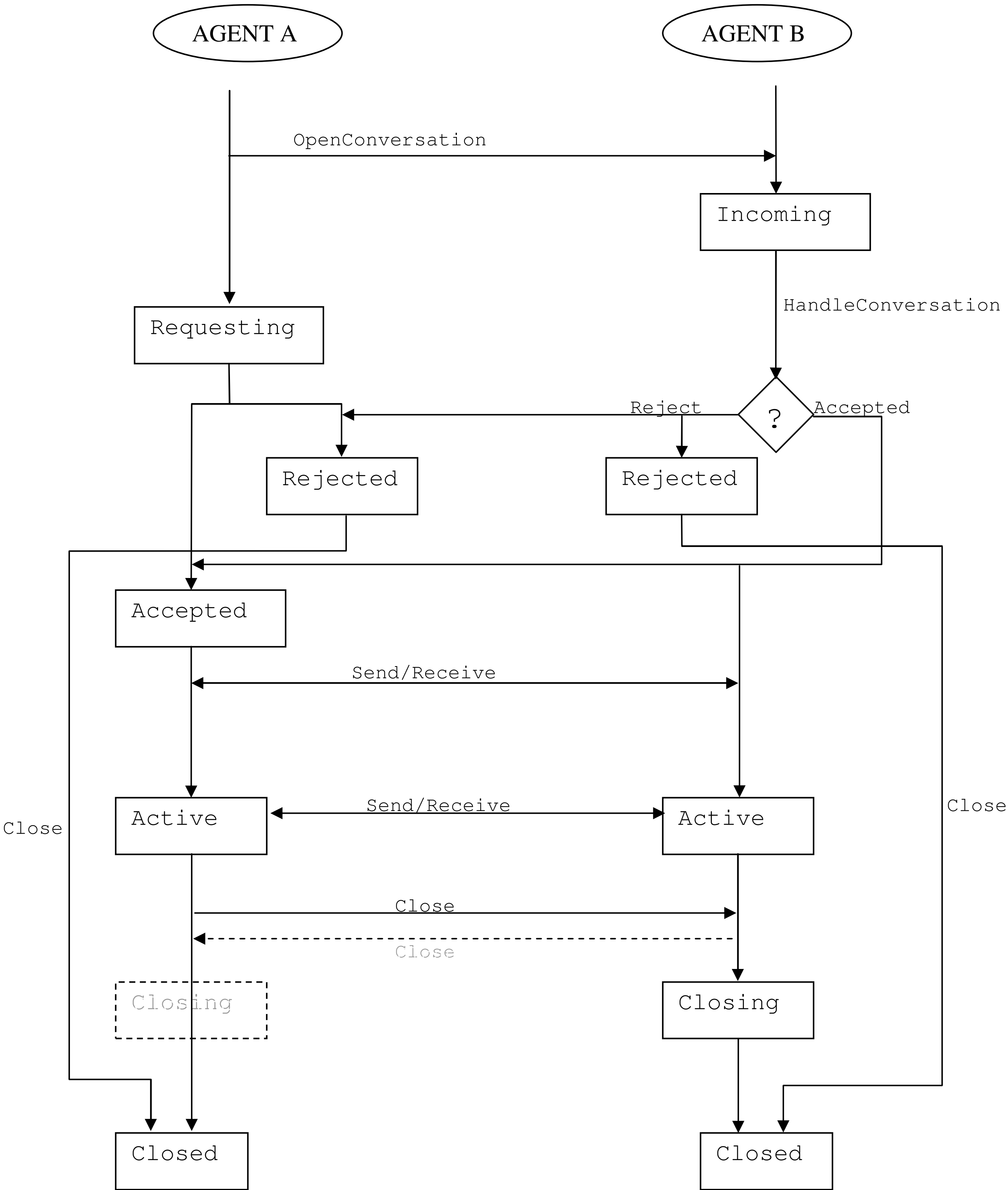
Una volta creato il filtro esso viene passato come parametro alla ReceiveMessage che utilizzerà l'oggetto delegate per decidere quali messaggi filtrare.

```
AgentMessageCollection messaggi = this.Runtime.MessageSvc.ReceiveMessage(filtro,false);
```

4.2.4 Gestione delle conversazioni

Come già anticipato, una conversazione è una sorta di canale virtuale che viene instaurato tra due agenti. In quanto tale, prima che i due agenti si possano scambiare messaggi all'interno di una conversazione, è necessario che il canale venga creato. In questa sede non verrà trattato il modo in cui tale conversazione viene creata (è un'operazione effettuata autonomamente dalla piattaforma), ma solamente come utilizzarla.
Si ricorda che una conversazione può avvenire solo tra due agenti alla volta, mentre un agente può avere più conversazioni attive contemporaneamente (sono possibili anche più conversazioni differenti tra gli stessi due agenti e perfino tra due stessi behaviour).
Il ciclo di vita di una conversazione è definito da una successione di stati, la cui evoluzione è gestita in parte dal client del servizio di messaggistica, in parte dal programmatore. Più precisamente ciascuno dei due agenti partecipanti alla conversazione mantiene un proprio stato in cui si trova la conversazione e l'evoluzione degli stati può quindi essere controllata (come vedremo in seguito) per gestire meglio la conversazione stessa.
Per una maggiore chiarezza, di seguito viene rappresentato graficamente lo svolgimento di una conversazione tra due agenti.
Verrà poi discusso per prima cosa in via teorica il ciclo di vita di una conversazione (4.2.4.1), e si vedrà quindi in seguito come realizzare praticamente la conversazione. Per fare questo viene prima descritta l'interfaccia *IConversation* che fornisce tutti gli strumenti per gestire la conversazione (4.2.4.2) ed infine come realizzare il restante codice per l'effettivo funzionamento della conversazione (4.2.4.3).

4.2.4.1 Ciclo di vita di una conversazione



Nel diagramma sopra mostrato l'agente A è quello che richiede all'agente B di aprire una conversazione.
Quando A vuole aprire una conversazione con B, manda a quest'ultimo una richiesta con il comando *OpenConversation*. L'oggetto conversazione che viene restituito da tale metodo si pone nello stato *Requesting* e resta in attesa di una risposta da B (può essere positiva o negativa). B non appena riceve la richiesta di conversazione costruisce un oggetto conversazione e lo pone nello stato *Incoming*. A questo punto la piattaforma invoca il metodo *HandleConversation* di B. (Si ricorda che tale metodo era stato definito in AgentTemplate, e verrà nuovamente trattato in seguito). B può quindi accettare o rifiutare la conversazione.

Caso 1: *Reject*. B rifiuta la conversazione. Entrambi gli agenti vanno in stato *Rejected* e restano in attesa del comando *Close* per chiudere la conversazione (è cura del programmatore chiuderla). Ricevuti i rispettivi comandi entrambi vanno nello stato *Closed*. La conversazione è terminata ed il canale viene distrutto.

Caso 2: *Accept*. B accetta la conversazione: viene creato il canale virtuale e l'agente A va nello stato *Accepted*. Nel momento in cui il primo messaggio viene spedito tramite la conversazione (*Send/Receive*) i due oggetti conversazione di A e B transiscono nello stato *Active*. I due agenti possono continuare a scambiarsi messaggi finchè uno dei due non manda all'altro una richiesta di chiusura della conversazione invocando il metodo *Close*. Nella figura di esempio A invoca *Close*: di conseguenza l'oggetto conversazione di A va nello stato *Closed* (chiusura attiva), viene generato automaticamente un messaggio di chiusura della conversazione e lo manda a B. B riceve il messaggio indicante la chiusura della conversazione e va nello stato *Closing* (chiusura passiva). In questo stato l'agente può solo leggere i messaggi della conversazione che sono presenti in coda, e non può più comunicare con l'altro agente. B deve quindi chiamare *Close* per chiudere completamente la conversazione ed andare nello stato *Closed*. La conversazione è terminata e il canale virtuale che costituiva la conversazione viene distrutto.
In figura è rappresentato tratteggiato il caso contrario: infatti la chiusura della conversazione può essere richiesta da uno qualunque dei due agenti.

Esiste un ulteriore stato: *Invalid*. Questo stato è raggiunto ogni qualvolta viene effettuata un'operazione non consentita (ad esempio se si cerca di inviare un messaggio quando si è in uno stato diverso da *Accepted* o *Active*).

4.2.4.2 L'interfaccia IConversation

L'interfaccia *IConversation* offre tutti gli strumenti necessari per la gestione di una conversazione. Una conversazione è creata tramite l'interfaccia *IMsgClient*, quando un behaviour di un'agente chiama il metodo OpenConversation() che torna un oggetto IConversation.
Tramite tale interfaccia è possibile:

- ricavare l'identificativo di una conversazione (si ricorda che ciascuna conversazione è identificata univocamente da un intero);
- ricavare lo stato della conversazione (vedi 4.2.4.1);
- ricavare l'AID dell'interlocutore;
- mandare, ricevere, aspettare messaggi;
- controllare la propria coda di messaggi;
- chiudere la conversazione.

```
namespace AgentService.Agent.Runtime
{
	public interface IConversation
	{
		int ConvId { get; }
		ConversationState State { get; }
		AID Peer { get; }
		bool WaitForAck(System.TimeSpan timeout);
		void Accept();
		void Reject();
		bool HasMessages();
		bool SendMessage(MessageBody content);
		AgentMessage ReceiveMessage();
		AgentMessage PeekMessage();
		AgentMessage WaitForMessage();
		void Close();
	}
}
```

<u>Proprietà</u>:

- `int ConvId;`
  Restituisce l'identificatore della conversazione.

- `ConversationState State;`
  Restituisce lo stato della conversazione. Tale proprietà può essere usata dal programmatore per controllare lo stato in cui la piattaforma si trova e gestirla di conseguenza. Gli stati della conversazione sono definiti dall'enum *ConversationState* (nel namespace `AgentService.Agent.Runtime`):

```
public enum ConversationState
{
	//A request of conversation has been submitted.
	//The initiator is waiting for a response.
	Requesting,

	//A conversation request has arrived.
	Incoming,

	//The conversation has been accepted by the peer.
	Accepted,
```

```
            //The conversation has been reject by the peer.
            Rejected,

            //The conversation is active and at least one message has been
            //sent by one of the two agents that own the conversation.
            Active,

            //The conversation has been closed no further operations
            //are possible on the conversation object.
            Closed,

            //The conversation has reached an invalid state, no further
            //Operations are possible on the conversation object.
            Invalid
      }
```

- `AID Peer;`
  Restituisce l'AID dell'interlocutore.

Metodi:

- `bool WaitForAck(System.TimeSpan timeout);`
  Aspetta una risposta alla richiesta di conversazione. Con timeout si stabilisce il periodo di tempo entro cui deve arrivare l'acknowledge: il metodo restituisce *true* se l'ack viene ricevuto entro tale timeout, *false* altrimenti. Se si pone timeout a zero, il metodo attende finchè la conversazione non viene accettata o rifiutata.

- `void Accept();`
  Accetta la conversazione rappresentata dall'istanza tramite cui il metodo viene chiamato. L'agente che invoca l'Accept pone il proprio oggetto conversazione nello stato *Active*. La piattaforma invia un messaggio con tale risposta all'altro agente che evolve nello stato *Accepted*.

- `void Reject();`
  Rifiuta la conversazione rappresentata dall'istanza tramite cui il metodo viene chiamato. Viene mandato un messaggio con tale risposta all'altro agente. Entrambi gli oggetti conversazione degli agenti evolvono nello stato *Rejected*.

- `bool HasMessages();`
  Controlla se la coda dell'agente ha messaggi appartenenti alla conversazione. Restituisce *true* in caso positivo.

- `bool SendMessage(MessageBody content);`
  Spedisce un messaggio all'interno della conversazione. Il metodo vuole il corpo del messaggio da inviare come parametro. Restituisce *true* se l'operazione è avvenuta con successo.

- `AgentMessage ReceiveMessage();`

Riceve un messaggio della conversazione: restituisce il primo *AgentMessage* presente in coda, rimuovendolo dalla coda.

- `AgentMessage PeekMessage();`
  Restituisce il primo messaggio presente nella coda dell'agente e appartenente alla conversazione. Tale messaggio non viene rimosso dalla coda.

- `AgentMessage WaitForMessage();`
  Attende che un nuovo messaggio appartenente alla conversazione giunga nella coda dell'agente e lo estrae.

- `void Close();`
  Chiude la conversazione. L'agente che chiama il metodo pone il proprio oggetto conversazione nello stato *Closed*. Viene generato inoltre un messaggio di chiusura ed inviato all'altro agente il quale porrà il proprio oggetto conversazione nello stato *Closing*.

4.2.4.3 <u>La conversazione</u>

Come già visto nel paragrafo 4.2.4.1, si può supporre che A sia l'agente che richiede a B di aprire una conversazione. A quindi, invierà una richiesta a B tramite il comando *OpenConversation*[2]. Tale comando si trova nell'interfaccia *IMsgClient* (vedi 4.2):

```
IConversation OpenConversation(AID to);
```

A *OpenConversation* è necessario passare l'AID dell'agente con cui si vuole realizzare la conversazione. Il comando poi, restituisce un oggetto *IConversation* contenente anche l'identificatore della conversazione che si sta cercando d'instaurare.

Nel frattempo, non appena B riceve la richiesta va nello stato *Incoming* e la piattaforma invoca il metodo *HandleConversation* di B. Tale metodo (già accennato in 3.3) è definito in AgentTemplate:

```
public virtual void HandleConversation(IAgentContext context, ConversationArgs args)
{
      args.Conversation.Reject();
      args.Conversation.Close();
}
```

Questo metodo ha due parametri:

[2] Vi è anche un metodo *OpenConversation* che ritorna una *IOntologicalConversation*. Per i dettagli si raccomanda la lettura del Capitolo 7: ed in particolare del paragrafo 7.4.

- sender è un *IAgentContext* (analizzato in seguito) e rappresenta l'agente che ha ricevuto la richiesta di apertura conversazione ed ha quindi lanciato l'evento relativo;
- *ConversationArgs* è un oggetto contenente al suo interno un oggetto *IConversation*, tramite cui si può gestire la conversazione. Vi si può accedere tramite la proprietà *Conversation* di *ConversationArgs*.

Il codice sopra riportato è quello presente nella classe base *AgentTemplate* e di default rifiuta ogni conversazione. Come già detto il programmatore può ridefinire tale metodo nella rispettiva classe derivata (User_AgentTemplate) e decidere di conseguenza quale azione intraprendere alla ricezione di una richiesta di conversazione. Per fare questo il programmatore può accettare o rifiutare la conversazione tramite l'oggetto *ConversationArgs*.
Nel caso in cui si decida di rifiutare una conversazione è anche necessario chiamare il metodo *Close*() per chiuderla.
Invece se si accetta la conversazione, bisogna aggiungerla alla coda di conversazioni di un qualche behaviour (si ricorda che ogni conversazione deve sempre essere associata ad un behaviour che può essere creato ad hoc oppure preesistente) con il metodo *AddConversation* chiamato tramite un oggetto che implementa l'interfaccia *IAgentContext*.

IAgentContext

Questa interfaccia è contenuta nel namespace `AgentService.Agent.Runtime` e fornisce dei metodi utilizzabili all'interno di *HandleConversation*.

```
public interface IAgentContext : IFactoryContext
{
      string[] BehaviourNames
      void AddConversation(string behaviourName, IConversation conversation);
      void StartBehaviour(string behaviourName);
}
```

Proprietà per l'accesso in lettura:

- `string[] BehaviourNames;`
  Restituisce il nome dei behaviour appartenenti all'agente.

Metodi:

- `void AddConversation(string behaviourName, IConversation conversation);`
  Aggiunge la conversazione alla coda delle conversazioni del behaviour. Vuole come parametri: una stringa con il nome del behaviour e un reference all'oggetto *IConversation* che si vuole aggiungere.
  Ogni behaviour ha una coda delle conversazioni (con lo stesso agente o con agenti diversi). Il behaviour infatti mantiene informazioni sull'agente che richiede la conversazione e non è a conoscenza del behaviour che la richiede.

Tale coda di conversazioni è memorizzata nella classe base *Behaviour* in una *IConversationCollection* che è organizzata come una collezione. E' possibile accedervi tramite la proprietà:

```
protected IConversationCollection IncomingConvs
```

Notare che tale proprietà è dichiarata protected. Il programmatore infatti la utilizzerà all'interno dei behaviour degli agenti che crea.

- `void StartBehaviour(string behaviourName);`
  Avvia il behaviour specificato come parametro. Questo perchè un behaviour può essere stato creato precedentemente senza essere stato mandato in esecuzione. Ad esempio può essere attivato quando si riceve una richiesta di conversazione affidandola a tale behaviour.

Si è quindi definito come l'agente B risponde alla richiesta di apertura di conversazione di A. I due agenti possono cominciare a scambiarsi messaggi all'interno della conversazione appena creata. Il codice che gestisce lo scambio di messaggi deve essere posto all'interno del metodo *body()* del behaviour interessato (o comunque deve essere chiamato dall'interno di tale metodo).
Si riporta di seguito un esempio di conversazione tra due agenti (viene riportato solo parte del codice presente nel metodo body dei due behaviour).

Il primo agente possiede un behaviour che cerca d'instaurare una conversazione con l'agente *Magalli*, aspetta di ricevere un messaggio, risponde e poi chiude la conversazione.

```
public override void Body()
{
	AgentService.Agent.Runtime.IConversation conv =
	this.Runtime.MessageSvc.OpenConversation(new AID("Magalli",this.Owner.HapName));

	while((conv.State !=AgentService.Agent.Runtime.ConversationState.Accepted) &&
		     (conv.State !=AgentService.Agent.Runtime.ConversationState.Active))
	{
		System.Threading.Thread.Sleep(10);
	}
	AgentService.Platform.MTS.AgentMessage recMsg = null;

	this.Runtime.Console.AppendMessage("waiting for data..");

	recMsg = conv.WaitForMessage();
	int price = (int)recMsg.Body["StartPrice"];
	int newPrice = price+5;
	this.Runtime.Console.AppendMessage("..ok : {0},{1}",price,newPrice);

	MessageBody body = new MessageBody();
	body.Add("OriginalPrice",(object)price);
	body.Add("NewPrice",(object)newPrice);
```

```
        AgentMessage msg = new AgentMessage("acl:Auction",body,conv.Peer);
        conv.SendMessage(body);
        this.Runtime.Console.AppendMessage("NewPrice sent");
        System.Threading.Thread.Sleep(100);

        conv.Close();
}
```

Il secondo agente, Magalli, alla ricezione della richiesta di conversazione, accetta la richiesta ed aggiunge la conversazione alla coda di conversazioni del behaviour ManageAuction. Questo è specificato nel metodo *HandleConversation* della classe derivante *AgentTemplate* che caratterizza l'agente creato l'agente Magalli:

```
public override void HandleConversation(AgentService.Agent.Runtime.IAgentContext
sender, AgentService.Agent.Runtime.ConversationArgs args)
{
        args.Conversation.Accept();
        sender.AddConversation("ManageAuction",args.Conversation);
}
```

Alla ricezione di un messaggio indicante la richiesta di apertura di conversazione (inviato dalla piattaforma dopo che Magalli ha effettuato una chiamata al metodo *OpenConversation*), la conversazione viene aggiunta alla coda di conversazioni del behaviour specificato.
A questo punto la conversazione potrà essere gestita (come previsto dal codice scritto dal programmatore) dall'interno del behaviour: nel nostro caso si tratta di ManageAuction, del quale nel seguito viene riportata la parte di body() che gestisce la comunicazione.

Magalli manda all'agente con cui ha instaurato la comunicazione il prezzo iniziale e attende che gli torni in risposta il prezzo finale.

```
public override void Body()
{
            int startPrice = 5;
            MessageBody msgBody = new MessageBody();
            msgBody.Add("StartPrice",((object)startPrice));
            this.Runtime.Console.AppendMessage("Auction: {0}",startPrice);

            IConversation conv1 = this.IncomingConvs[0];

            conv1.SendMessage(msgBody);
            System.Threading.Thread.Sleep(1200);

            this.Runtime.Console.AppendMessage("data sent");

            AgentMessage recMsg1 = null;
            recMsg1 = conv1.WaitForMessage();

            int finalPrice= (int)recMsg1.Body["NewPrice"];
            this.Runtime.Console.AppendMessage("final price: {0}",finalPrice);
```

```
            conv1.Close();
}
```

4.3 Servizio di logging – ILogClient

L'interfaccia *ILogClient* dà all'agente l'accesso al sottosistema di logging. L'oggetto che a tempo di esecuzione verrà restituito come attraverso la proprietà *LogSvc* del runtime, viene creato dal modulo di logging.

```
namespace AgentService.Agent.Runtime
{
      public interface ILogClient
      {
            void Append(LoggingLevel level, string message);
            void CloseSession(int token);
            void OpenSession(int token);
      }
}
```

Ogni volta che un agente viene attivato, la piattaforma invoca il metodo *OpenSession* e quando l'agente viene stoppato *CloseSession*. Questi due metodi non possono essere invocati dal programmatore, pena la generazione di errore.
Il metodo che il programmatore può usare è *Append*, che vuole come parametri il livello di logging e la stringa contente il messaggio di log che si vuole scrivere.
Il *LoggingLevel* è un enum che rappresenta il livello d'importanza che possiede il log che si vuole effettuare, ed è definito nel namespace `AgentService.Platform.Modules`:

```
      public enum LoggingLevel
      {
            // The message appended to the log is informative.
            Info,
            // The message warns about some potential erroneous conditions.
            Warning,
            // The message appended to the log is about an erroneous but not critical
            // condition.
            Alert,
            // The message appended to the log indicates that a severe error has
            // occurred.
            Critical,
            // The system is in an unrecoverable state, crash is likely to happen.
            Panic
      }
```

Il modulo imposta un livello di logging e tale livello viene utilizzato per filtrare i messaggi di log.
Il metodo *Append* passa il messaggio di log al sottosistema di logging solo se il suo *LoggingLevel* è uguale o superiore a quello della piattaforma.

Il programmatore deve quindi assegnare un *LoggingLevel* al suo log in base all'importanza che egli pensa che abbia. Esso sarà poi effettivamente scritto solo in base a come viene settata la piattaforma.

4.4 Servizio di pagine bianche – IWhitePagesClient

L'interfaccia IWhitePagesClient definisce le funzionalità offerte dal servizio di pagine bianche (elenco del telefono) della piattaforma. A tempo di esecuzione, quando viene attivato un agente è compito dell'AMS fornire un opportuno oggetto che offra tali servizi ad ogni agente.
L'AMS indicizza automaticamente tutti gli agenti presenti nel MAS, e fornisce nel contempo un servizio di ricerca per nome degli agenti. Tale servizio restituisce al behaviour che ne ha fatto richiesta una descrizione degli agenti che combaciano con i criteri della ricerca.

```
namespace AgentService.Agent.Runtime
{
	public interface IWhitePagesClient
	{
		AgentDescription[] Search(string pattern);
		AgentDescription SearchExact(string name);
	}
}
```

Il metodo *Search* ricerca gli agenti tramite l'espressione regolare che gli viene passata per parametro (una stringa che descrive la struttura che deve avere il nome dell'agente ricercato). Viene restituita, tramite un array di *AgentDescription*, la lista degli agenti i cui nomi corrispondono ai criteri di ricerca inseriti.

Il metodo *SearchExact* restituisce invece l'*AgentDescription* dell'agente il cui nome corrisponde esattamente alla stringa fornita per parametro.

L'*AgentDescription* è una classe che costituisce la descrizione di un agente. E' organizzata così come prevede lo standard FIPA e contiene i seguenti dati:

- l'AID dell'agente
- il flag che stabilisce la persistenza o meno dell'agente
- una stringa indicante il proprietario dell'agente
- lo stato in cui si trova l'agente

Sono anche definite le proprietà per l'accesso in lettura a tali dati (esclusa la persistenza) che possono essere usate dal programmatore:

- `AgentService.Agent.AID AID`
- `string Ownership`
- `AgentService.Agent.AgentState State`

4.5 Servizio di pagine gialle – IYellowPagesClient

Questa interfaccia offre il servizio di pagine gialle (ricerca di un agente per servizi offerti). A differenza del servizio di pagine bianche offerto dall'AMS, l'iscrizione di un agente a questo servizio non avviene in maniera automatica, ma l'agente deve fare dichiarazione esplicita al DF (è quindi cura del programmatore farlo, se ritiene utile il servizio).

```
public interface IYellowPagesClient
{
      DFAgentDescription[] SearchAgent(ServiceDescription sd);
      DFAgentDescription[] SearchAgent(ServiceDescription sd,
                                       SearchConstraints constraint);
      void Register(DFAgentDescription description);
      void Edit(DFAgentDescription description);
      void Deregister();
}
```

Descriviamo per prima cosa le classi *DFAgentDescription* e *ServiceDescription*, e spieghiamo poi i metodi dell'interfaccia.

DFAgentDescription:

Descrive i servizi offerti da un agente. La classe è conforme allo standard FIPA. E' composta da:

- l'AID dell'agente
- una collezione dei servizi offerti dall'agente
- una lista dei protocolli d'interazione supportati dall'agente
- una lista delle ontologie conosciute dall'agente
- una lista dei linguaggi conosciuti dall'agente

La classe possiede inoltre le proprietà per l'accesso in lettura ai rispettivi dati membro:

- `AgentService.Agent.AID Aid`
- `ServiceDescriptionCollection Services`
- `System.Collections.ArrayList Protocols`
- `System.Collections.ArrayList Ontologies`
- `System.Collections.ArrayList Languages`

ServiceDescription:

Contiene la descrizione di un servizio. E' conforme allo standard FIPA. I suoi dati membro sono:

- il nome del servizio
- il tipo del servizio
- il sottotipo del servizio
- l'AID del proprietario del servizio
- la lista dei protocolli d'interazione supportati dal servizio
- la lista delle ontologie supportate dal servizio
- la lista dei linguaggi supportati dal servizio
- la lista delle proprietà che discriminano il servizio
- un flag indicante se il servizio è pubblicato dalla piattaforma, ossia se è conosciuto anche all'esterno della piattaforma

La classe possiede inoltre le proprietà per l'accesso in lettura ai rispettivi dati membro:

- `string Name`
- `string Type`
- `int SubType`
- `AID Owner`
- `System.Collections.ArrayList Protocols`
- `System.Collections.ArrayList Ontologies`
- `System.Collections.ArrayList Languages`
- `System.Collections.ArrayList Properties`
- `bool AllowPublishing`

Metodi dell'interfaccia _IYellowPagesClient_:

- `DFAgentDescription[] SearchAgent(ServiceDescription sd);`
  Ricerca gli agenti che offrono il servizio le cui caratteristiche coincidono con quelle del _ServiceDescription_ passato per parametro. Il metodo ritorna un array di _DFAgentDescription_.

- `DFAgentDescription[] SearchAgent(ServiceDescription sd, SearchConstraints constraint);`
  Opera come il metodo precedente, ma la ricerca tiene conto delle restrizioni imposte dall'oggetto _SearchConstraints_, analizzato alla fine di questo paragrafo.

- `void Register(DFAgentDescription description);`
  Registra l'agente che invoca il metodo al servizio di Directory Facilitator. Il metodo vuole come parametro la descrizione dei servizi implementati dell'agente tramite un oggetto _DFAgentDescription_.

- `void Edit(DFAgentDescription description);`
  Modifica la descrizione di un agente precedentemente registrato al servizio di Directory Facilitator con i valori contenuti nell'oggetto passato per parametro.

- `void Deregister();`
  Cancella la registrazione dell'agente dal servizio di Directory Facilitator.

SearchConstraints:

Questa classe definisce le restrizioni che saranno utilizzate dal Directory Facilitator per effettuare la ricerca. E' definita nel namespace `AgentService.Platform.DF` e contiene le seguenti informazioni:

- il metodo di ricerca da utilizzare;
- la massima profondità di propagazione della ricerca nelle directory. Questo valore non deve essere negativo;
- il massimo numero di risultati che la ricerca deve ritornare. Questo valore non deve essere negativo;

Il programmatore deve creare un oggetto SearchConstraints da passare al metodo *SearchAgent* sopra descritto; deve quindi utilizzare il seguente costruttore:

```
public SearchConstraints(int depth, int maxResult, SearchMethod sm)
```

I parametri necessari per il costruttore sono la profondità della ricerca, il numero massimo di risultati ed il metodo di ricerca. Quest'ultimo parametro può essere passato come un enum che assume i seguenti valori (definito nel namespace `AgentService.Platform.DF`):

```
public enum SearchMethod
{
      //Search only in the agents of the same platform
      SamePlatform = 1,

      //Search only in the agents of other platforms
      OtherPlatform = 2,

      //Performs the search everywhere
       Global = SamePlatform + OtherPlatform
}
```

La classe SearchConstraints offre anche delle proprietà per l'accesso in lettura ai dati membro:

- `SearchMethod Method`
- `int MaxDepth`

- `int MaxResults`

4.6 Servizio di persistenza – IPersistenceClient

Quest'interfaccia offre dei metodi per la persistenza delle knowledge. E' possibile forzare la persistenza di una knowledge o di un suo item. Con la chiamata a questi metodi il corrispondente oggetto viene passato al sottosistema di persistenza per essere salvato.

```
namespace AgentService.Agent.Runtime
{
	public interface IPersistenceClient
	{
		void PersistKnowledge(AgentService.Agent.Knowledge.Knowledge knowledge);
		bool PersistKnowledgeItem(string kName, string iName, object iValue);
	}
}
```

- *PersistKnowledge*: persiste la knowledge il cui riferimento le è passato per parametro. La sua implementazione in un behaviour può avvenire solo dopo che si è avuto accesso esclusivo alla knowledge (con LockKnowledge). Il metodo lancia un'eccezione se fallisce l'operazione di persistenza.

- *PersistKnowledgeItem*: persiste l'item specificato di una knowledge. Come parametri vuole: il nome dell'istanza knowledge che contiene l'item desiderato, il nome dell'item di cui si desidera effettuare la persistenza, il valore dell'item. Il metodo ritorna *true* se la persistenza dell'item è andata a buon fine, *false* altrimenti.

4.7 Servizio di controllo dei behaviour – IBehaviourContext

Quest'interfaccia fornisce l'accesso ai behaviour dell'agente. Offre dei metodi che permettono di gestire la sincronizzazione dei behaviour. I metodi presentati hanno tutti la funzione di sospendere il behaviour corrente finchè non terminano il o i behaviour che vengono passati per parametro.

```
namespace AgentService.Agent.Runtime
{
	public interface IBehaviourContext : IAgentContext
	{
		void WaitForTermination(System.Type behaviourType);
		void WaitForTermination(params System.Type[] behaviourTypes);
		void WaitForTermination(string behaviourName);
```

```
            void WaitForTermination(params string[] behaviourNames);
        }
}
```

Il metodo *WaitForTermination* può accettare come parametri:

- `System.Type behaviourType ->` il behaviour corrente è sospeso finchè non terminano tutte le istanze del tipo di behaviour specificato. Non è possibile mettersi in attesa della terminazione di tutti i behaviour che hanno lo stesso tipo del behaviour corrente. Qualora venga specificato come parametro il tipo del behaviour corrente, il metodo ritorna subito onde evitare un'attesa infinita.
- `params System.Type[] behaviourTypes ->` il behaviour corrente è sospeso finchè non terminano tutte le istanze dei tipi di behaviour specificati (se ne possono passare quanti se ne vogliono). Come nel caso precedente la specifica del tipo del behaviour che invoca il metodo non comporta alcun effetto.
- `string behaviourName ->` il behaviour corrente viene sospeso finchè termina il behaviour con il nome specificato. Se tale behaviour non è al momento in esecuzione (perchè ad esempio già terminato), il metodo non resta in attesa. Se viene specificato il nome del behaviour che ha invocato il metodo la chiamata torna subito onde evitare l'attesa indefinita.
- `params string[] behaviourNames ->` il behaviour corrente viene sospeso finchè terminano tutti i behaviour con i nomi specificati. Come nel caso precedente la specifica del nome del behaviour che ha invocato il metodo non ha alcun effetto.

Capitolo 5: Gestione del MAS

La gestione della piattaforma e quindi del MAS può essere realizzata in due modi differenti:
- tramite un oggetto *PlatformController*
- tramite l'interfaccia *IAgentServiceManager*

Entrambi questi strumenti permettono di controllare il ciclo di vita della piattaforma (vedi 2.6), istanziare ed attivare gli agenti: in pratica sono un pannello di controllo della piattaforma.
PlatformController viene utilizzata quando si esegue la piattaforma in modalità batch: in questo caso è necessario creare una classe apposita che gestisca il ciclo di vita della piattaforma e degli agenti (vedi 4.5). Tale classe risulta inoltre essere la scelta più opportuna qualora si desideri riscrivere il driver della piattaforma ad esempio nel caso occorra incorporare la piattaforma in un'altra applicazione.
L'interfaccia *IAgentServiceManager* costituisce il punto di accesso alla piattaforma in uno scenario di comunicazione inter-processo.

5.1 La classe PlatformController

La classe *PlatformController* contiene le proprietà per la lettura dello stato della piattaforma e metodi per la gestione degli agenti, della piattaforma e per il caricamento dei moduli, rispettivamente attraverso le proprietà *Services* e *Modules*. Di seguito vengono esposte e commentate le principali definizioni dei metodi utilizzabili dal programmatore. Un esempio del loro utilizzo è esposto nel paragrafo 3.5. La classe è contenuta nel namespace *AgentService.Platform* (che è quindi necessario includere per il suo utilizzo).

Proprietà per l'accesso in lettura ai dati membro:

- `PlatformState PlatformState`
  Restituisce lo stato della piattaforma (vedi 2.6 per informazione sui possibili stati).
- `PlatformDescription PlatformDescription`
  Restituisce un oggetto *PlatformDescription* contenente come suoi dati membro le caratteristiche della piattaforma. Per accedere a quest'ultime sono state implementate le seguenti proprietà in modalità di lettura:

    - `string Name ->` nome della piattaforma.

  - `bool Dynamic ->` *true* se la piattaforma supporta la registrazione dinamica degli agenti, *false* altrimenti.
  - `bool Mobile ->` *true* se la piattaforma supporta agenti mobili.
  - `string TransportProfile ->` profilo di trasporto della piattaforma.

Metodi per la gestione del ciclo di vita della piattaforma:

- `public void StartPlatform();`
  Avvia la piattaforma che passa dallo stato idle a quello ready.

- `public void RunPlatform();`
  Esegue la piattaforma, attivando prima i core agents e poi gli user agents, e la porta nello stato running.

- `public void StopPlatform();`
  Stoppa la piattaforma, portandola nello stato di ready.

- `public void RestartPlatform();`
  Riavvia la piattaforma. Ciò implica prima lo shutdown della piattaforma e poi il suo avvio.

- `public void ShutdownPlatform();`
  Chiude la piattaforma. Se è in running, chiama prima lo StopPlatform.

5.1.1 Servizi accessibili attraverso la proprietà Services

Metodi per la gestione degli agenti:

- `public AID CreateAgent(string name, string type, string config, bool bPersist);`
  Questo metodo crea un agente con i parametri che gli vengono passati: il nome dell'agente che si vuole creare, il tipo dell'agente (l'*AgentTemplate* che si è definito), una stringa che individua il file di configurazione, un bool che deve essere posto a *true* se si vuole che l'agente creato sia persistente, *false* altrimenti. Infatti a differenza della Knowledge (vedi cap 4.1), l'agente non possiede differenti livelli di persistenza. Il metodo ritorna l'AID dell'agente appena creato.

- `public void ActivateAgent(params AID[] aids);`
  Attiva la lista di agenti che gli vengono passati come parametro tramite il loro AID.

- `public void ActivateAllAgents();`
  Attiva tutti gli agenti creati.

- `public void WaitForAllAgents();`
  Attende che tutti gli agenti in esecuzione terminino i rispettivi compiti.

- `public void KillAgent(params AID[] aids);`
  Termina la lista di agenti che vengono passati come parametro tramite il loro AID.

- `public void KillEmAll();`
  Termina tutti gli agenti in esecuzione.

- `public void SuspendAgent(params AID[] aids);`
  Sospende la lista di agenti passati come parametro tramite il loro AID.

- `public void SuspendAllAgents();`
  Sospende tutti gli agenti attivi.

- `public void WakeUpAgent(params AID[] aids);`
  Risveglia la lista di agenti passati come parametro (agenti che erano stati precedentemente sospesi).

- `public void WakeUpAllAgents();`
  Risveglia tutti gli agenti sospesi.

5.1.2 Servizi accessibili attraverso la proprietà Modules

Metodi per il caricamento dei moduli:

- `public ModuleError UpLoadAssembly(string asmPath, bool bReplaceIfExists);`
  Carica un assembly nello storage della piattaforma. Come parametri vuole il path dell'assembly da caricare, ed un bool che indica se si vuole che l'assembly, qualora fosse presente nello storage, sia sovrascritto o meno. Il metodo ritorna un null se il caricamento è andato a buon fine, un oggetto *ModuleError* altrimenti. Si rimanda a 4.5.2 per la descrizione di *ModuleError*.
  E' importante sapere che un modulo può essere caricato nello storage solamente se la piattaforma si trova nello stato di *Ready*, pena la generazione di un errore.

- `public ModuleError RemoveAssembly(string assemblyName);`
  Rimuove dallo storage l'assembly con il nome passatogli come parametro. Il metodo ritorna null se la rimozione è andata a buon fine, un oggetto *ModuleError* altrimenti.

Capitolo 6: Estendere AgentService

L'architettura di AgentService è fortemente modulare: diverse funzionalità della piattaforma (base ed aggiuntive) sono state sviluppate implementando delle plug-in che una volta installate, la piattaforma carica al suo avvio. Il maggior vantaggio che costituisce un'architettura di questo tipo è non solo la possibilità di customizzare le funzionalità base della piattaforma sostituendo i plug-in forniti di default con plug-in realizzati da terzi che implementano le stesse funzionalità, ma soprattutto la possibilità di arricchire la piattaforma di nuove funzionalità implementando dei plug-in che integrati nel ciclo di vita della piattaforma, svolgono compiti particolari.

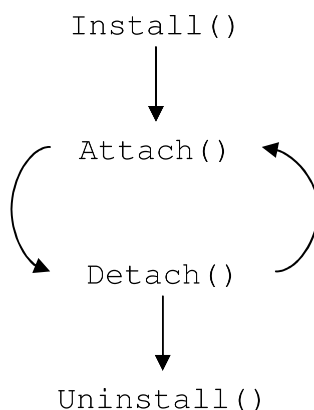
Il code-base di AgentService espone delle interfacce per implementare i plug-in. In questo capitolo:

- analizzeremo le funzionalità esposte da queste interfacce;
- vedremo come i plug-in interagiscono con la piattaforma;
- implementeremo un modulo aggiuntivo che effettua lo snooping dei messaggi.

6.1 Creazione di moduli

Per poter creare un modulo è necessario implementare l'interfaccia *IPlatformModule*. Tale interfaccia deve essere utilizzata da qualsiasi tipo di modulo (core o optional). Nel caso in cui si voglia creare un core module, è necessario implementare una delle seguenti interfacce che derivano da IPlatformModule: *ILoggingModule, IMessagingModule, IPersistenceModule, IStorageModule*.

Ogni modulo ha un proprio ciclo di vita che può essere schematizzato tramite il diagramma sottostante.



Il ciclo di vita di un modulo viene gestito dalla piattaforma. I metodi a fianco riportati sono definiti nell'interfaccia *IPlatformModule*. Per prima cosa un modulo deve essere installato: ciò avviene tramite una chiamata al metodo *Install*`()`da parte della piattaforma quando quest'ultima si trova nello stato Binding ed è la prima volta che viene avviata. Il modulo viene poi caricato e scaricato dalla piattaforma invocando i suoi metodi *Attach*`()` e *Detach*`()` quando si trova rispettivamente nello stato Starting e

ShuttingDown. Infine il metodo *Uninstall*`()` viene invocato quando il modulo viene disinstallato dalla piattaforma (in seguito al comando da console *asdrv -u*).

6.1.1 L'interfaccia IPlatformModule

L'interfaccia *IPlatformModule* definisce le funzionalità comuni che tutti i moduli caricati nella piattaforma devono possedere.

```
namespace AgentService.Platform.Modules
{
	public interface IPlatformModule
	{
		string Name { get; }
		string Description { get; }
		ModuleCategory Category { get; }
		bool IsMain { get; }
		bool CanSupport(string capabiltyInterface);
		ModuleError GetLastError();
		bool Attach(IPlatformContext hostingPlatform, string configFile);
		bool Detach();
		bool Install(PlatformContext context, string configFile, ref
				InstallationInfo instInfo);
		bool UnInstall(InstallationInfo instInfo);
	}
}
```

Proprietà:

- `string Name;`
  Restituisce il nome del modulo.

- `string Description;`
  Restituisce una descrizione riassuntiva del modulo.

- `ModuleCategory Category;`
  Restituisce la categoria a cui appartiene il modulo. Tale categoria è realizzata tramite l'enum *ModuleCategory*.

```
public enum ModuleCategory
{
	//AgentService platform core modules
	Core,
	//Messaging subsystem modules
	Messaging,
	//Persistence subsystem modules
	Persistence,
```

```
        //Storage subsystem modules
        Storage,
        //Logging subsystem modules
        Logging,
        //Other additional modules
        Other
    }
```

- `bool IsMain;`
  Restituisce *true* se il modulo è in grado di realizzare le funzionalità base della categoria di appartenenza, *false* in caso contrario.

<u>Metodi</u>:

- `bool CanSupport(string capabiltyInterface);`
  Controlla se il modulo supporta le caratteristiche definite nell'interfaccia passata come parametro. Il metodo ritorna *true* se le caratteristiche sono supportate, *false* altrimenti.

- `ModuleError GetLastError();`
  Restituisce un oggetto *ModuleError* rappresentante l'ultimo errore che si è verificato nel modulo; ritorna *null* nel caso in cui non si siano verificati errori.

- `bool Attach(IPlatformContext hostingPlatform, string configFile);`
  Tale metodo è chiamato ogni volta in cui viene caricato il modulo. Nel suo corpo è quindi necessario scrivere ciò che il modulo deve fare in tale circostanza. A tale metodo vengono passati dalla piattaforma come parametri un oggetto *IPlatformContext* (analizzato in seguito) e il nome di un file di configurazione (senza percorso), che può essere utilizzato per leggere o scrivere delle informazioni. Si noti che il path del *configFile* è noto alla piattaforma ed è memorizzato all'interno dell'implementazione di default dell'interfaccia *IPlatformContext* passata per parametro. Il metodo restituisce *true* se l'operazione termina senza errori.

- `bool Detach();`
  E' invocato dalla piattaforma ogni volta in cui il modulo viene scaricato. Restituisce *true* se l'operazione termina senza errori.

- `bool Install(IPlatformContext context, string configFile, ref InstallationInfo instInfo);`
  Invocato dalla piattaforma quando il modulo viene installato, restituisce *true* se l'installazione termina con successo. Il codice scritto nel suo corpo può utilizzare i metodi di *IPlatformContext* ed accedere al contenuto del *configFile* e di *instInfo*, quest'ultimo è il repository delle informazioni d'installazione del modulo. La classe *InstallationInfo* è una classe base astratta che deve essere ridefinita per poter essere utilizzata. Tale classe è vuota: viene quindi lasciata al programmatore piena libertà sulla sua implementazione. Una volta definita una *User_InstallationInfo*

(derivata da InstallationInfo), essa deve essere istanziata all'interno del metodo *Install*: la piattaforma passa infatti a tale metodo un oggetto InstallationInfo come riferimento inizializzato a null. L'oggetto User_InstallationInfo deve poter essere serializzabile dal momento che le informazioni memorizzate in tale oggetto vengono salvate su file e in un secondo tempo recuperate per essere utilizzate dal metodo *Uninstall*`()`. Per tal motivo è obbligatorio che questa classe sia serializzabile.

- `bool UnInstall(InstallationInfo instInfo);`
  Metodo chiamato in fase di disinstallazione del modulo, restituisce *true* se il processo va a buon fine. La piattaforma passa come parametro l'oggetto *InstallationInfo* creato in fase d'installazione.

6.1.2 Le interfacce IPlatformContext e IPlatformContextEx

L'interfaccia *IPlatformContext* definisce le funzionalità base utilizzabili da ciascun modulo. Tali funzionalità possono essere utilizzate nel corpo dei metodi *Install*`()` ed *Attach*`()`. *IPlatformContextEx*, che offre un paio di funzionalità in più, è utilizzabile nella stessa identica maniera, ma solamente dai moduli base (core modules).

IPlatformContext si compone di:

- eventi standard definiti per la piattaforma;
- proprietà per l'accesso in lettura ad informazioni sulla piattaforma;
- metodi per la creazione e gestione di eventi definiti dall'utente (custom).

L'interfaccia è contenuta nel namespace `AgentService.Platform`

Eventi standard della piattaforma:

```
public interface IPlatformContext
{
	// Fires every time an agent is created.
	event PlatformEventHandler OnAgentCreated;

	// Fires every time an agent is resumed.
	event PlatformEventHandler OnAgentResumed;

	// Fires every time an agent is stopped.
	event PlatformEventHandler OnAgentStopped;

	// Fires every time an agent is woken up.
	event PlatformEventHandler OnAgentWokenUp;

	// Fires every time a message is sent.
	event PlatformEventHandler OnMessageSent;
```

```
        // Fires every time a message is sent.
        event PlatformEventHandler OnMessageReceived;

        // Fires every time a bulk of messages is received.
        event PlatformEventHandler OnMessageBatch;

        // Fires every time a conversation is requested.
        event PlatformEventHandler OnConversationRequested;

        // Fires every time a conversation is accepted and then opened.
        event PlatformEventHandler OnConversationAccepted;

        // Fires every time a comversation is refused.
        event PlatformEventHandler OnConversationRefused;

        // Fires every time a conversation is closed.
        event PlatformEventHandler OnConversationClosed;

        // Fires every time an agent is persisted.
        event PlatformEventHandler OnAgentPersisted;

        // Fires every time an assembly is uploaded into the platform storage.
        event PlatformEventHandler OnAssemblyUploaded;

        // Fires every time an assembly is removed from the platform storage.
        event PlatformEventHandler OnAssemblyRemoved;
    }
```

PlatformEventHandler è l'handler predefinito per tutti i tipi di eventi standard della piattaforma. E' definito tramite il seguente delegate:

```
public delegate void PlatformEventHandler(object sender, PlatformEventArgs args);
```

PlatformEventArgs:

PlatformEventArgs è la classe che costituisce l'argomento di tutti gli handler che saranno registrati. Contiene come dati membro il tipo ed il nome dell'evento. Nel caso che l'evento sia di tipo custom, il nome dell'evento distingue il tipo dell'evento.
Le proprietà :

- `PlatformEvent Type`
- `string Name`

permettono l'accesso in lettura ai rispettivi dati membro.
Il costruttore è il seguente:

```
public PlatformEventArgs(PlatformEvent type, string name)
```

Costruisce l'oggetto in base al tipo di evento indicato (*parametro type*). Assegna automaticamente il nome in base a tale tipo. Se invece l'evento è custom, il nome dell'evento è definito dal parametro *name*.

PlatformEvent è un enumeratore che definisce i possibili eventi standard della piattaforma:

```
public enum PlatformEvent
{
      AgentCreated,
      AgentResumed,
      AgentStopped,
      AgentWokenUp,
      AgentPersisted,
      MessageBatch,
      MessageSent,
      MessageReceived,
      ConversationRequested,
      ConversationAccepted,
      ConversationRefused,
      ConversationClosed,
      AssemblyRemoved,
      AssemblyUploaded,
      Custom
}
```

Tramite questi strumenti il programmatore può quindi registrare l'handler del modulo per l'evento che gli interessa, ed assegnargli poi le funzionalità che desidera esso compia.

Proprietà per accesso alle informazioni sulla piattaforma:

```
public interface IPlatformContext
{
      //Gets the platform base directory
      string PlatformBaseDir

      //Gets the platform core modules configuration path
      string PlatformCoreConfPath

      //Gets the platform core modules bin path
      string PlatformCoreBinPath

      //Gets the platform additional modules configuration path
      string PlatformAddConfPath

      //Gets the platform additional modules bin path
      string PlatformCoreAddPath

      //Gets the platform description
      PlatformDescription Description
}
```

Metodi per creazione e gestione degli eventi custom:

```
public interface IPlatformContext
```

```
{
       int AddEvent(string evtName);
       bool RemoveEvent(int token);
       bool RaiseEvent(int token, PlatformEventArgs args);
       bool Register(string evtName, PlatformEventHandler handler);
       bool Deregister(string evtName, PlatformEventHandler handler);
}
```

- `int AddEvent(string evtName);`
  Aggiunge l'evento passato come parametro al registro di eventi custom. Restituisce un numero intero che rappresenta il token di possesso dell'evento. Per poter accedere e modificare l'evento è quindi necessario possedere il token (essere cioè a conoscenza del suo valore). Se l'evento era già presente nel registro, il metodo restituisce 0.

- `bool RemoveEvent(int token);`
  Rimuove dal registro di eventi custom l'evento identificato dal token passato per parametro. Restituisce *true* se l'operazione ha avuto successo.

- `bool RaiseEvent(int token, PlatformEventArgs args);`
  Lancia l'evento associato al token passato per parametro. Come parametro il metodo vuole anche un oggetto *PlatformEventArgs* che descrive l'evento. Restituisce *true* se l'evento è stato trovato nel registro e quindi lanciato.

- `bool Register(string evtName, PlatformEventHandler handler);`
  Registra un handler per l'evento passato come parametro. Il metodo necessita come parametro anche il delegate che sarà registrato. Restituisce *true* se la registrazione ha avuto successo.

- `bool Deregister(string evtName, PlatformEventHandler handler);`
  Cancella l'handler precedentemente registrato per l'evento: entrambi devono essere passati come parametri. Restituisce *true* se l'operazione va a buon fine.

Quindi quando il programmatore vuole implementare ed utilizzare eventi da lui gestiti deve:
- scrivere il codice dell'evento custom
- aggiungerlo al registro degli eventi custom
- lanciare l'evento
- quando non più necessario, è possibile:
  - rimuovere la registrazione dell'handler per l'evento
  - rimuovere l'evento dal registro degli eventi custom

<u>IPlatformContextEx</u>:

IPlatformContextEx aggiunge delle funzionalità all'interfaccia *IPlatformContext*. E' utilizzabile solo dai moduli base. Solo a loro è infatti permesso lanciare eventi standard

della piattaforma e spedire dei comandi alla piattaforma. E' anch'essa contenuta nel namespace `AgentService.Platform`.

```
public interface IPlatformContextEx : IPlatformContext
{
	void Raise(PlatformEvent eventType, PlatformEventArgs args);
	void PostCommand(PlatformCommand cmd);
}
```

- *Raise*: lancia un evento della piattaforma. Un esempio del suo utilizzo è il seguente:

```
	this.hapContext.Raise(PlatformEvent.MessageSent,
	                      new PlatformEventArgs(PlatformEvent.MessageSent, “"));
```

  Nell'esempio viene lanciato l'evento "Messaggio inviato" e viene creato un nuovo *PlatformEventArgs* a cui si passa il tipo dell'evento ed il suo nome.

- *PostCommand*: invia il comando passato per parametro alla piattaforma. Il comando inviato può essere uno di quelli già predefiniti all'interno della piattaforma, oppure può essere creato secondo necessità dal programmatore. Per fare questo bisogna creare un oggetto derivando la propria classe da *PlatformCommand*, che è una classe base astratta contenente come dati membro il PlatformState e un flag che indica, se posto a *true*, che il comando deve essere eseguito solamente se il *PlatformState* presente in questa classe coincide con quello della piattaforma, altrimenti viene richiesto che la piattaforma si trovi almeno in quello stato. Vediamone un esempio:

```
public class StartCommand : PlatformCommand
{
	public StartCommand() : base(PlatformState.Ready, true)
	{}

	public override void Execute(ICommandContext cmdCtx)
	{
		cmdCtx.StartPlatform();
	}

	public override bool ParseArguments(string args)
	{
		return true;
	}
}
```

  La classe richiama il costruttore della classe base che è disponibile nelle seguenti versioni:

  `public PlatformCommand()`
  Il costruttore di default pone *PlatformState* a *FailStart* e il flag a *true*.

  `public PlatformCommand(PlatformState state, bool bMatchExact)`
  Crea un oggetto *PlatformCommand* con lo specificato stato e flag.

Nell'esempio il comando viene eseguito solo se la piattaforma si trova nello stato *Ready*.

E' poi necessario ridefinire il metodo *Execute*, che ha per parametro un oggetto *ICommandContext*, interfaccia che fornisce tutti i metodi per la gestione della piattaforma e degli agenti. E' necessario inserire nel corpo di tale metodo il codice costituente il comportamento del comando. Nel nostro caso *Execute* avvia la piattaforma.
Se ritenuto utile, è possibile ridefinire *ParseArguments*, che ha la funzione di analizzare un eventuale stringa con parametri in input.

Dall'interfaccia IPlatformModule (come detto prima) derivano quattro interfacce (una per ciascuna delle funzionalità core della piattaforma) che aggiungono le funzionalità specifiche che i core modules devono implementare:

- ILoggingModule -> per il corretto funzionamento del modulo di logging
- IMessaggingModule -> per il corretto funzionamento del modulo di messaggistica
- IPersistenceModule -> per il corretto funzionamento del modulo di persistenza
- IStorageModule -> per il corretto funzionamento del modulo di storage

6.1.3 L'interfaccia ILoggingModule

ILoggingModule definisce le operazioni comuni che un modulo di Logging della piattaforma deve possedere. Quest'interfaccia implementa a sua volta *IPlatformModule*: il modulo di logging che il programmatore può scrivere dovrà quindi implementare i membri di entrambe queste interfacce.

```
namespace AgentService.Platform.Modules
{
	public interface ILoggingModule : IPlatformModule
	{

		LoggingLevel Level { get; set; }
		string Format { get; set; }
		void Append(LoggingLevel level, string message);
		void Clear();
		StringCollection GetMessages();
		bool HasClient { get; }
		ILogClient CreateClient(AID aid, int token);
	}
}
```

Proprietà:

- `LoggingLevel Level;`

Permette di impostare e di leggere il livello di logging attualmente impostato nella piattaforma (per maggiori dettagli vedi 4.3).

- `string Format;`
  Permette di impostare e leggere il formato del logging.

Metodi:

- `void Append(LoggingLevel level, string message);`
  Invia il messaggio passato per parametro al sottosistema di logging. Tale messaggio verrà scritto solo se il livello di logging passato per parametro è maggiore o uguale al livello di logging impostato per la corrente esecuzione della piattaforma.

- `void Clear();`
  Cancella tutti i messaggi archiviati nel sottosistema di logging.

- `StringCollection GetMessages();`
  Restituisce la collezione di messaggi memorizzati nel sottosistema di logging.

- `bool HasClient;`
  Restituisce *true* se il modulo prevede un client custom.

- `ILogClient CreateClient(AID aid, int token);`
  Crea e restituisce un client del sottosistema di logging per l'agente specificato come parametro.

6.1.4 L'interfaccia IMessaggingModule

IMessaggingModule definisce le funzionalità base che un modulo di messaggistica deve possedere per poter essere integrato in AgentService ed implementa a sua volta *IPlatformModule*: il modulo di messaging che il programmatore può scrivere dovrà quindi implementare i membri di entrambe queste interfacce. Comprende metodi per la gestione dei messaggi e delle conversazioni.

```
namespace AgentService.Platform.Modules
{
	public interface IMessagingModule : IPlatformModule
	{
		bool CreateAgentQueue(AID aid);
		bool RemoveAgentQueue(AID aid);
		bool HasMessages(AID aid);
		bool SendMessage(AgentMessage message);
		AgentMessage ReceiveMessage(AID aid);
		AgentMessage PeekMessage(AID aid);
```

```
            AgentMessageCollection GetAllMessages(AID aid);
            IMsgClient CreateClient(AID aid);
            bool HasClient
            bool RemoveMessage(AID aid, AgentMessage message);
      }
}
```

Metodi:

- `bool CreateAgentQueue(AID aid);`
  Crea un coda di messaggi per l'agente specificato. Restituisce *true* se l'operazione è andata a buon fine.

- `bool RemoveAgentQueue(AID aid);`
  Rimuove la coda di messaggi posseduta dall'agente specificato. Restituisce *true* se l'operazione va a buon fine.

- `bool HasMessages(AID aid);`
  Controlla se la coda dell'agente specificato possiede dei messaggi. Restituisce *true* in caso affermativo, false altrimenti.

- `bool SendMessage(AgentMessage message);`
  Effettua la consegna del messaggio passato per parametro. Restituisce *true* se il messaggio viene mandato con successo.

- `AgentMessage ReceiveMessage(AID aid);`
  Riceve un messaggio estraendolo dalla coda di messaggi dell'agente specificato. Restituisce tale messaggio.

- `AgentMessage PeekMessage(AID aid);`
  Restituisce il primo messaggio presente nella coda dell'agente passato per parametro. Il messaggio non viene estratto dalla coda.

- `AgentMessageCollection GetAllMessages(AID aid);`
  Recupera tutti messaggi presenti nella coda dell'agente passato per parametro. I messaggi non vengono estratti dalla coda. Il metodo restituisce la collezione contenente tali messaggi.

- `IMsgClient CreateClient(AID aid);`
  Crea e restituisce un client del modulo di messaggistica per l'agente passato per parametro.

- `bool HasClient {get};`
  Restituisce true se il modulo ha un client di tipo custom.

- `bool RemoveMessage(AID aid, AgentMessage message);`

Rimuove il messaggio passato per parametro dalla coda dell'agente. Il metodo restituisce *true* se l'operazione è andata a buon fine.

6.1.5 L'interfaccia IPersistenceModule

IPersistenceModule definisce le funzionalità che un modulo di gestione della persistenza della piattaforma deve possedere per operare correttamente ed implementa a sua volta *IPlatformModule*: il modulo *Persistence* che il programmatore può scrivere dovrà quindi implementare i membri di entrambe queste interfacce.

```
namespace AgentService.Platform.Modules
{
      public interface IPersistenceModule : IPlatformModule
      {
            bool PersistAgent(AgentService.Agent.AgentPersistent agent);
            bool ResumeAgent(AID aid, ref AgentService.Agent.AgentPersistent agent);
            bool RemoveAgent(AID aid);
            bool PersistKnowledge(AID aid, string name, KnowledgePersistent
                                  knowledgePersistent);
            bool HasClient { get; }
            IPersistenceClient CreateClient(AID aid);
      }
}
```

Metodi:

- `bool PersistAgent(AgentService.Agent.AgentPersistent agent);`
  Effettua la persistenza di tutti i dati (sensibili alla persistenza) dell'agente passato per parametro. Restituisce *true* se l'operazione ha avuto successo.

- `bool ResumeAgent(AID aid, ref AgentService.Agent.AgentPersistent agent);`
  Effettua il resume dell'agente passato per parametro dallo storage. Il metodo utilizza come parametro anche un reference a tale agente, così da poterlo modificare direttamente. Restituisce *true* se l'operazione ha avuto successo.

- `bool RemoveAgent(AID aid);`
  Rimuove tutti i dati dell'agente passato per parametro dal sottosistema di persistenza.

- ```
  bool PersistKnowledge(AID aid, string name,
                        KnowledgePersistent knowledgePersistent);
  ```
  Effettua la persistenza della knowledge passata come parametro da un dato agente. Restituisce *true* se l'operazione va a buon fine.

- `bool HasClient { get; }`
  ```

Restituisce *true* se il modulo di persistenza ha implementato un client custom.

- `IPersistenceClient CreateClient(AID aid);`
  Crea e restituisce un client del modulo di persistenza per l'agente passato per parametro.

6.1.6 L'interfaccia IStorageModule

IStorageModule definisce le funzionalità del modulo che gestisce il sistema di storage degli assembly. Quest'interfaccia implementa a sua volta *IPlatformModule*: il modulo di storage che il programmatore può scrivere dovrà quindi implementare i membri di entrambe queste interfacce.

```
namespace AgentService.Platform.Modules
{
	public interface IStorageModule : IPlatformModule
	{
		bool ClearStorage();
		bool AddAssembly(string path, bool bReplaceIfExists);
		bool RemoveAssembly(string name);
		string GetAssemblyFor(string template);
		System.Reflection.AssemblyName[] GetDependentAssemblies(string
									assemblyName);
		bool Synchronize();
	}
}
```

Metodi:

- `bool ClearStorage();`
  Cancella l'intero contenuto dello storage. Restituisce *true* se l'operazione ha avuto successo. Quest'operazione è pericolosa: cancella infatti tutti gli assembly caricati dalla piattaforma e di conseguenza anche quelli contenenti eventuali agenti scritti dall'utente.

- `bool AddAssembly(string path, bool bReplaceIfExists);`
  Aggiunge l'assembly identificato dal path passato per parametro allo storage. Restituisce *true* se l'operazione è andata a buon fine. Ulteriore parametro è un *bool* indicante se l'assembly deve essere sostituito se già presente nello storage.

- `bool RemoveAssembly(string name);`
  Rimuove dallo storage l'assembly con il nome passato per parametro. Restituisce *true* se l'operazione ha successo.

- `string GetAssemblyFor(string template);`

Restituisce il nome dell'assembly in cui si trova l'*AgentTemplate* passato per parametro.

- `System.Reflection.AssemblyName[] GetDependentAssemblies(string assemblyName);`
  Restituisce un array contenente tutti i nomi degli assembly dipendenti da quello passato per parametro.

- `bool Synchronize();`
  Sincronizza il contenuto fisico dello storage con il repository dei tipi di *AgentTemplate*. Restituisce *true* se la sincronizzazione è andata a buon fine.
  E' bene chiamare sempre tale metodo dopo aver aggiunto o rimosso assembly dallo storage.

6.2 Esempio di modulo addizionale – SnoopingModule

In questa sezione descriveremo l'implementazione di un modulo addizionale che si occupa di fare lo snooping dei messaggi inviati tra gli agenti e scrive su un file (creato in una directory dedicata) la data e l'ora di ogni singolo messaggio inviato o ricevuto. Nell'implementare il modulo verrà descritto come implementare i metodi dell'interfaccia *IPlatformModule* e come fare in modo che il modulo si agganci agli aventi che espone l'interfaccia *IPlatformContext*.

```
using System;
using System.IO;
using AgentService.Platform.Modules;

namespace AgentService.Platform.Modules.Additional
{
	[Serializable]
	public class SnoopingInfo : InstallationInfo
	{
		/// path of snooping's directory
		public string snoopingDirectoryPath;

		/// Constructs a SnoopingInfo object.
		/// <param name="baseDir">path of snooping's directory</param>
		public SnoopingInfo(string path)
		{
			this.snoopingDirectoryPath = path;
		}
	}
```

Il modulo in questione necessita quasi sempre di informazioni di installazione, che nel nostro caso sono costituite solo dal percorso della directory in cui verranno scritti i file che contengono i dati tracciati dal modulo. Si definisce perciò la classe *SnoopingInfo* che deriva da *InstallationInfo* e si pone come membro della classe il percorso della directory in cui salvare il file. Ricordiamo che è obbligatorio attribuire alla classe appena definita l'attributo *Serializable*.

```
/// Summary description for SnoopingModule.
public class SnoopingModule:IPlatformModule
{
	/// base logging directory for the log file
	private string baseSnoopingDirectory;

	/// Gets the base logging directory for the log files of agents
	public string BaseSnoopingDirectory
	{
		get
		{
			return System.IO.Path.Combine(this.hapContext.PlatformBaseDir,
			                              this.baseSnoopingDirectory);
		}
	}

	/// absolute snooping path
	private string snoopingPath;

	/// Gets the absolute snooping path
	public string SnoopingPath
	{
		get
		{
			return this.snoopingPath;
		}
	}

	/// log file extension used
	private string snoopingExtension;

	/// Gets the log file extension used
	public string SnoopingExtension
	{
		get
		{
			return this.snoopingExtension;
		}
	}

	/// Writer object for the platform log file
	private StreamWriter writer;

	/// Platform context, platform accessible data
	private AgentService.Platform.IPlatformContext hapContext;

	/// Construcs a SnoopingModule object
	public SnoopingModule()
	{
	}

	/// name of the module
	private string name = "Snooping Module";

	/// Gets the name of the module
	public string Name
```

```
{
        get
        {
                return this.name;
        }
}

/// description of the module
private string description = "Snooping of messages and conversations:
                                record on file";

/// Gets a summary description of the module
public string Description
{
        get
        {
                return this.description;
        }
}

/// Gets the module category
public AgentService.Platform.Modules.ModuleCategory Category
{
        get
        {
                return AgentService.Platform.Modules.ModuleCategory.Other;
        }
}

/// Gets a bool value indicating if the module is
/// able to perform the main functionality of the
/// belonging category.
public bool IsMain
{
        get
        {
                return true;
        }
}

/// Checks wheter the module supports features defined in a given
///interface.
/// If the these features are supported the module reference can be
///subject to cast against the requested interface.
/// <param name="capabiltyInterface">full name of the interface that
/// <returns>true if the specified interface is supported
public bool CanSupport(string capabiltyInterface)
{
        return false;
}

/// Last Error occurred in the module
private ModuleError lastError;

/// Returns the last error occurred in the module, null if there
/// are no errors to return
```

```
/// <returns>Reference to the error.
public ModuleError GetLastError()
{
      return this.lastError;
}
```

Dopo aver fornito un'implementazione per le proprietà ed i metodi informativi esposti dall'interfaccia *IPlatformModule* viene implementato il metodo *Install()*. Nel caso del tipo *SnoopingModule*, il metodo verifica l'esistenza del file di configurazione e, qualora presente, legge da esso la directory in cui verranno mantenuti i log dei dati tracciati dal modulo e l'estensione per tali file di log. Se non esiste alcun file di configurazione il modulo utilizza dei valori di default: una sottodirectory nell'installazione di AgentService che ha il nome *snooping* e *.log* come estensione dei file. Al termine di questa operazione viene creato l'oggetto *SnoopingInfo* e salvato nelle informazioni di installazione il percorso della directory utilizzata dal modulo.

```
/// Installs the module in the platform.  Called by the pltaform when
/// the module is installed.
/// <param name="hostingPlatform">platform accessible data</param>
/// <param name="configFile">configuration file</param>
/// <param name="instInfo">installation info object used to store install
/// information
/// <returns>true if the installation process has been successful
public bool Install(AgentService.Platform.IPlatformContext
        hostingPlatform, string configFile, ref InstallationInfo instInfo)
{
      this.hapContext = hostingPlatform;

      if (System.IO.File.Exists(configFile) == false)
      {
            // ok generate default configuration file
            System.IO.StreamWriter writer = new
                                         System.IO.StreamWriter(configFile);

            this.baseSnoopingDirectory = "snooping" +
                                        System.IO.Path.DirectorySeparatorChar;
            writer.WriteLine(this.baseSnoopingDirectory);

            this.snoopingExtension = ".log";
            writer.WriteLine(this.snoopingExtension);

            writer.Close();
            this.snoopingPath = System.IO.Path.Combine
                                        (this.hapContext.PlatformBaseDir,
                                         this.baseSnoopingDirectory);

            instInfo=new SnoopingInfo(this.snoopingPath);
      }
      else
      {
            // ok read the configuration file
            try
```

```
        {
                System.IO.StreamReader reader = new
                                    System.IO.StreamReader(configFile);
                this.baseSnoopingDirectory = reader.ReadLine();

                if (this.baseSnoopingDirectory.EndsWith("" +
                    System.IO.Path.DirectorySeparatorChar) == false)
                this.baseSnoopingDirectory +=
                             System.IO.Path.DirectorySeparatorChar;

                this.snoopingPath = System.IO.Path.Combine
                                   (this.hapContext.PlatformBaseDir,
                                    this.baseSnoopingDirectory);

                this.snoopingExtension = reader.ReadLine();
                reader.Close();
                instInfo=new SnoopingInfo(this.snoopingPath);
        }
        catch(System.Exception ex)
        {
                this.lastError = new ModuleError(this,"Could not read
                                                  configuration file",ex);
                return false;
        }
    }

    if (System.IO.Directory.Exists(this.snoopingPath) == false)
    {
        try
        {
                System.IO.Directory.CreateDirectory(this.snoopingPath);
        }
        catch(System.IO.IOException ioex)
        {
                this.lastError = new ModuleError(this, "Could not create
                                               snooping directory", ioex);
                return false;
        }
    }

    return true;
}
```

La disinstallazione del modulo prevede per questo tipo solo la cancellazione della directory utilizzata per tracciare i dati. Viene perciò recuperato il percorso dall'oggetto *SnoopingInfo* precedentemente creato durante l'installazione.

```
/// Uninstall the module. Called by the platform when the module is
///removed permanently from the platform
/// <param name="instInfo">installation info object used to install the
/// module
/// <returns>true if the uninstall process has been successfull</returns>
public bool UnInstall(InstallationInfo instInfo)
{
    SnoopingInfo snInfo= instInfo as SnoopingInfo;
```

```
        if (snInfo!=null)
        {
                this.snoopingPath=snInfo.snoopingDirectoryPath;

                System.IO.Directory.Delete(this.snoopingPath,true);
                return true;
        }
        return false;
}
```

Il metodo *Attach(...)* viene invocato quando il modulo è caricato dalla piattaforma in fase di start. Dal file di configurazione vengono lette le informazioni necessarie per individuare la directory in cui mantenere il log dei dati tracciati e l'estensione del file di log. Il modulo aggiorna inoltre il file di log scrivendo al suo interno un messaggio di inizio sessione e si registra per essere notificato degli eventi *OnMessageSent*, *OnMessageReceived*, *OnMessageBatch*.

```
/// Attaches the module to the platform. Called by the platform
/// when the module is loaded.
/// <param name="hostingPlatform">reference to the instance of the hosting
///platform
/// <param name="configFile">configuration file</param>
/// <returns>returns true is the operation has been successfull and with
///no errors
public bool Attach(AgentService.Platform.IPlatformContext hostingPlatform,
                   string configFile)
{
        this.hapContext = hostingPlatform;
        try
        {
                System.IO.StreamReader reader = new
                                          System.IO.StreamReader(configFile);
                this.baseSnoopingDirectory = reader.ReadLine();
                this.snoopingPath = System.IO.Path.Combine
                           (this.hapContext.PlatformBaseDir,
                            this.baseSnoopingDirectory);
                this.snoopingExtension = reader.ReadLine();
                reader.Close();
        }
        catch(System.Exception ex)
        {
                this.lastError = new ModuleError(this,"Could not read
                                          configuration file",ex);
                return false;
        }

        string snoopingFile = string.Format("{0}{1}{2}",
                this.snoopingPath,
                hostingPlatform.Description.Name,
                this.snoopingExtension);

        try
        {
```

```
            this.writer = new System.IO.StreamWriter(snoopingFile,true);
            this.writer.AutoFlush = true;
            this.writer.WriteLine("{0} :: Session
                                      Begin.",System.DateTime.Now);

      }
      catch(System.IO.IOException ioex)
      {
            this.lastError = new ModuleError(this, "Error opening platform
                                   snooping file " + ioex.Message, ioex);
            return false;
      }

      this.handler = new PlatformEventHandler(this.snoopMessage);

      this.hapContext.OnMessageSent += handler;
      this.hapContext.OnMessageReceived += handler;
      this.hapContext.OnMessageBatch += handler;
      return true;
}
```

Quando il modulo viene scaricato dalla piattaforma occorre deregistrarsi dagli eventi a cui ci si era registrati in fase di caricamento e scrivere un messaggio di fine sessione sul file di log.

```
/// Detaches the module from the platform Called by the platform when
/// the module is unloaded
/// <returns>true if the operation has been successful</returns>
public bool Detach()
{
      if (this.handler != null)
      {
            this.hapContext.OnMessageSent -= handler;
            this.hapContext.OnMessageReceived -= handler;
            this.hapContext.OnMessageBatch -= handler;
      }
      try
      {
            this.writer.WriteLine("{0} :: Session
                                      End.",System.DateTime.Now);
            this.writer.Close();
      }
      catch(System.IO.IOException ioex)
      {
            this.lastError = new ModuleError(this, "Error closing platform
                                        snooping file " + ioex.Message, ioex);
            return false;

      }
      return true;
}
```

Il metodo privato *snoopMessage(...)* è quello che di fatto implementa il tracciamento dei dati determinando l'evento verificatosi e scrivendo l'opportuno messaggio sul file di log.

```
private void snoopMessage(object sender, PlatformEventArgs args)
{
	lock(this.writer)
	{
		switch(args.Type)
		{
			case PlatformEvent.MessageSent:
			{
				this.writer.WriteLine("{0} :: message sent",
								System.DateTime.Now);
				break;
			}

			case PlatformEvent.MessageReceived:
			{
				this.writer.WriteLine("{0} :: message received",
								System.DateTime.Now);
				break;
			}

			case PlatformEvent.MessageBatch:
			{
				this.writer.WriteLine("{0} :: all messages
							received", System.DateTime.Now);
				break;
			}

			default:break;
		}
	}
}
	}
}
```

Capitolo 7: Il servizio di supporto delle ontologie

AgentService fornisce un supporto per le ontologie che permette di creare e mantenere ontologie da impiegare per la rappresentazione di basi di conoscenza e soprattutto del contenuto dei messaggi.
Un'ontologia è un insieme esaustivo di elementi che rappresenta un dominio del discorso. Il dominio del discorso è composto da tutto ciò di cui si parla e su cui si ragiona relativamente ad un argomento.
Le ontologie per AgentService forniscono tutti quegli elementi necessari per portare avanti una conversazione fra agenti. In pratica un'ontologia fornisce delle classi da istanziare e usare come contenuto di un messaggio e tutta una serie di regole ed indicazioni per un uso corretto dello strumento ontologico.
Tramite l'ontologia, la rappresentazione semantica e sintattica è rigorosa e condivisa dagli agenti, evitanto così errori di interpretazione e inconsistenze varie.
Il servizio per le ontologie prevede inoltre la validazione del contenuto dei messaggi, ovvero la verifica che esso sia stato creato in conformità con le regole dell'ontologia (per esempio, l'obbligatorietà di una certa proprietà di uno oggetto, la cardinalità, vincoli vari…).

7.1 Le metaclassi ontologiche

Ogni ontologia fornisce un insieme di elementi che appartengono a generiche e ben determinate categorie (*metaclassi*).
Una qualsiasi ontologia contiene:

- *Concetti*: rappresentano oggetti astratti o concreti e sono in pratica il soggetto o i complementi di una frase. Dei concetti possono essere per esempio: persona, cane, pianeta, lavoro…
- *Predicati*: si tratta di affermazioni che possono essere *vere* o *false*. Coinvolgono al loro interno i concetti. Un esempio di predicato è dato da: `(Padrone: Cane, Persona)` il quale consiste nell'affermazione (vera o falsa) che la persona sia il padrone del cane. Ovviamente un predicato dovrà essere istanziato creando quindi anche un'istanza del concetto `persona` e del concetto `cane`.
- *Azioni* (AgentAction): sono particolari concetti che esprimono dei comandi impartiti da un agente ad un altro agente. Un esempio: `(Vendi: Libro, Persona)` e cioè l'ordine impartito ad un agente venditore per un acquisto di un libro da parte di una determinata persona. Ovviamente ha senso l'istanziazione di un'azione.

- *IRE*: sono le query. Rappresentano quindi delle domande. Contengono al loro interno un predicato contenente una variabile. Esempio: `(Query: Padrone, VariabileCane)`, che indica la richiesta di fornire la lista di tutti i cani di proprietà di un dato padrone.
- *Variabili*: sono usate, come già spiegato, nelle IRE. Sono oggetti indeterminati che rappresentano per esempio concetti o aggregati di concetti.
- *Primitive*: rappresentano i tipi primitivi quali stringhe, interi, booleani.
- *Aggregati*: rappresentano una lista, array o qualsiasi insieme di più elementi.

Quest entità sono detti *tipi di schemi*. Quando si progetta un'ontologia bisogna dotarla di schemi che si riconducono ai *tipi di schemi* fondamentali prima riportati.
Per esempio un'ontologia può essere dotata del concetto *Persona*, di cui esiste uno schema derivato dal tipo di schema *Concetto*; sarà presente l'azione *Vendi*, rappresentata da un tipo di schema *AgentAction* etc.

7.1.1 Attributi e vincoli

Ogni schema può essere dotato di *attributi* (detti anche *slot*). Gli attributi sono analoghi alle proprietà o dati membro delle classi C#. Un attributo di uno schema, a differenza di una proprietà di una classe, è dotato di *vincoli* e *obbligatorietà*. Con l'obbligatorietà si esprime il fatto che un attributo sia obbligatorio oppure opzionale. Lo schema *Persona* sarà dotato di un attributo *Cognome* obbligatorio e un altro, per esempio *Età*, opzionale.
E' previsto anche il *vincolo di cardinalità massima* e *minima* riguardo al numero di valori che possono essere contenuti in un attributo.
Oltre al vincolo di cardinalità, è possibile introdurre vincoli creati *ad hoc*, per soddisfare particolari necessità legate ad una particolare ontologia.

7.1.2 Schemi

Per ogni elemento del dominio del discorso è dunque previsto uno schema. Ogni schema deriva dalla classe *ObjectSchema*. Nel namespace `AgentService.Agent.Ontology.Schema` si trovano gli schemi relativi al *concetto*, *predicato*, *IRE* etc. Quando si vuole per esempio creare uno schema *Persona* in un'ontologia, bisognerà fornire l'indicazione che *Persona* è di tipo *ConceptSchema*. Analogamente per ogni altro elemento.
ObjectSchema e quindi tutti gli altri schemi sono dotati dei seguenti metodi.
Metodi:

- `public static ObjectSchema GetBaseSchema()`
  Ritorna un generico *ObjectSchema*. Ogni particolare schema ritornerà un generico schema del suo proprio tipo (*ConceptSchema*, *PredicateSchema*…).

- `public abstract void Add(string name, ObjectSchema slotSchema, int optionality)`
  Aggiunge uno slot allo schema, indicando nome, tipo di schema e opzionalità.

- `public abstract void Add(string name, ObjectSchema slotSchema)`
  Come nel precedente, ma non è indicata l'opzionalità. In tal caso lo slot è considerato obbligatorio.

- `public abstract void Add(string name, ObjectSchema elementsSchema, int cardMin, int cardMax)`
  Aggiunge allo schema uno slot dotato di cardinalità massima e minima.

- `public abstract void Add(string name, ObjectSchema elementsSchema, int cardMin, int cardMax, string aggType)`
  Aggiunge uno slot ad uno schema indicando il tipo di aggregato (si tratta di un particolare vincolo).

- `public abstract void AddSuperSchema(ObjectSchema superSchema)`
  Aggiunge uno super schema allo schema. Lo schema corrente eredita dal super schema tutti gli slot.

- `public abstract void AddFacet(string slotName, Facet f)`
  Dota uno slot di un vincolo definito dall'utente.

- `public abstract string GetTypeName()`
  Ritorna il nome dello schema.

- `public abstract string[] GetNames()`
  Ritorna i nomi degli slot dello schema.

- `public abstract void Validate(AbsObject abs, Ontology onto)`
  Valida un descrittore astratto rappresentante un oggetto, secondo le regole dell'ontologia indicata.

- `public abstract ObjectSchema GetSchema(string name)`
  Ritorna lo schema associato ad uno slot.

- `public abstract bool ContainsSlot(string name)`
  Verifica che lo schema contenga un dato slot.

- `public abstract bool IsMandatory(string name)`
  Verifica che uno slot sia obbligatorio.

- `public abstract bool IsCompatibleWith(ObjectSchema s)`
  Verifica che lo schema sia compatibile con un dato schema.

- `public abstract bool DescendsFrom(ObjectSchema s)`
  Verifica che lo schema discenda dallo schema indicato.

- `public abstract Facet[] GetFacets(string slotName)`
  Ritorna la lista di vincoli associati ad uno slot.

- `public abstract AbsObject NewInstance()`
  Fornisce un'istanza di un descrittore astratto di tipo analogo allo schema.

7.1.3 La classe Ontology

Ogni ontologia deve derivare dalla classe *Ontology* contenuta nel namespace `AgentService.Agent.Ontology`.
La classe *Ontology* memorizza al suo interno gli schemi ontologici e li associa alle classi che essi rappresentano (e che i programmatori devono istanziare quando creano il contenuto dei messaggi).
Inoltre mantiene una lista di *ontologie base* di cui l'ontologia che si sta progettando è una specializzazione. Un'ontologia base è molto utile perché evita allo sviluppatore di ontologie la creazione di schemi di largo impiego, che quindi sono creati una volta per tutte in qualche ontologia base.
Costruttori:

- `public Ontology(string name, Ontology Base, Introspector introspector)`
  Istanzia un'ontologia con un dato nome, una data ontologia e un dato *introspector* (il suo utilizzo sarà chiarito in seguito).

- `public Ontology(string name, Ontology Base)`
  Analogo al precedente, ma senza indicazione dell'introspector il che significa che si usa quello fornito di default.

- `public Ontology(string name, Ontology []Base, Introspector introspector)`
  Analogo al primo, solo che è fornita una lista di ontologie base.

- `public Ontology(string name, Ontology []Base)`
  Questo costruttore non indica l'introspector.

- `public Ontology(string name, Introspector introspector)`
  Non si indica nessuna ontologia base.

- `public Ontology(string name)`
  Non si indicano né ontologie base né introspector.

Metodi:

- `public string GetName()`
  Questo metodo ritorna il nome dell'ontologia.

- `public void Add(ObjectSchema schema)`

Il metodo aggiunge uno schema all'ontologia.

- `public void Add(ObjectSchema schema, Type Class)`
  Aggiunge uno schema all'ontologia associandovi anche un tipo di classe.

- `public ObjectSchema GetSchema(string name)`
  Ritorna lo schema con il nome fornito in ingresso.

- `public Type GetClassForElement(string name)`
  Ritorna il tipo di classe associato ad uno schema con un dato nome.

- `public AbsObject FromObject(object obj, Ontology globalOnto)`
  Ritorna un *descrittore astratto* per un dato oggetto, fornendo anche l'ontologia. Il suo utilizzo sarà chiarito in seguito.

- `public AbsObject FromObject(Object obj)`
  Come il precedente, ma senza indicazione riguardo l'ontologia.

7.1.4 Progettazione di vincoli

Nel namespace `AgentService.Agent.Ontology.Schema` è presente l'interfaccia *Facet* che deve essere implementata dai vincoli definiti dall'utente.
Questa interfaccia contiene un solo metodo che deve essere implementato dal nuovo vincolo:

- `void Validate(AbsObject val, Ontology onto)`
  Riceve in ingresso il descrittore astratto rappresentante l'oggetto da convalidare e l'ontologia contenente le regole per validarlo.
  Il metodo contiene il codice necessario per validare il descrittore astratto.

7.1.5 Descrittori astratti e introspector

La potenzialità dell'ontologia è il fatto che rappresenta in modo univoco e rigoroso le regole che descrivono un dato dominio del discorso. All'atto della validazione si rende necessario poter fare un paragone fra l'oggetto da validare e lo schema che ne rappresenta la categoria nell'ontologia.
Il processo di validazione in genere è totalmente nascosto all'utente, ma per completezza viene descritto ugualmente, anche per poter introdurre l'introspector.
Per mediare fra oggetti e schemi si introducono i *descrittori astratti*: delle rappresentazione che uniscono le particolarità degli *schemi* con i *valori degli oggetti*. In pratica sono schemi che contengono slot istanziati con i valori delle proprietà degli oggetti associati agli schemi.
Per poter creare questi descrittori astratti, si passa all'ontologia, tramite il metodo *FromObject*, l'oggetto da validare e si ottiene il descrittore astratto che descrive l'oggetto.

Per slot di tipo primitivo o comunque di cardinalità massima pari a 1, non ci sono problemi ad ottenere il *valore* dalle corrispondenti *proprietà* dell'oggetto *associato* allo *schema*. Per slot di tipo *aggregato*, con particolari strutture dati che non siano il tipico *ArrayList*, *array* etc. si rende necessario uno strumento che consenta di utilizzare i *metodi di accesso* alla struttura dati. Infatti, se si crea un'ontologia con uno schema dotato di uno slot per esempio del tipo particolare di grafo, la classe *Ontology* non conosce *a priori* i metodi di accesso al grafo.
I metodi di accesso alle consuete strutture dati sono disponibili nell'*introspector* fornito di default. Se si rendesse necessario disporre di un proprio introspector, occorre implementare l'interfaccia *Introspector* disponibile nel namespace `AgentService.Agent.Ontology.Validation.`
Per implementare tale interfaccia occorre definire il metodo:

- `AbsObject Externalise(Object obj, ObjectSchema schema, Type Class, Ontology referenceOnto)`
  Questo metodo ottiene in ingresso l'oggetto da convertire in descrittore astratto, lo schema associato all'oggetto, la classe associata allo schema e l'ontologia da usare.

Il metodo ritorna un descrittore astratto. Come si può intuire, viene utilizzato dal metodo *FromObject.*

7.2 Progettare un'ontologia

Per progettare un'ontologia occorre derivare la propria classe ontologica dalla classe *Ontology*.
Per ogni elemento del dominio del discorso ed ogni suo slot bisogna definire una stringa contenente il suo nome.
L'insieme di queste stringhe (definite come dati membro della classe ontologica) dà origine al vocabolario dell'ontologia. Quando si richiamerà uno schema, bisognerà utilizzare il nome dello schema contenuto nel vocabolario.

```
public class MusicShopOntology: Ontology
{
      public static readonly string ontology_name = "Music-shop-ontology";
      // VOCABULARY
      public static readonly string item = "Item";
      public static readonly string item_serial = "serial_number";
      public static readonly string cd = "CD";
      public static readonly string cd_name = "name";
      public static readonly string cd_tracks = "tracks";
      public static readonly string track = "Track";
      public static readonly string track_title = "title";
      public static readonly string track_duration = "duration";
```

Nel codice sopra riportato è illustrato un frammento dell'ontologia per un negozio musicale.

Fra gli altri elementi è previsto un concetto generico (*item*) dotato di *numero seriale*, un concetto *CD* dotato di attributi per il *titolo* e le *tracce* (un attributo di tipo *aggregato*). Una *traccia* è un altro concetto ed è dotato di *titolo* e *durata*.

```
private static Ontology theInstance = new MusicShopOntology();
public static Ontology Instance
{
    get
    {
        return theInstance;
    }
}
```

L'ontologia deve essere dotata di una proprietà *Instance* che fornisce l'unica istanza dell'ontologia.
Il costruttore dell'ontologia deve essere privato:

```
private MusicShopOntology(): base(ontology_name, BasicOntology.Instance)
```

Come si vede nella riga di codice sopra, il costruttore invoca il costruttore della classe *Ontology* passandovi il nome dell'ontologia e l'istanza dell'ontologia base.

7.2.1 BasicOntology

Naturalmente è possible creare una propria ontologia base (o insieme di ontologie base), ma nel namespace `gentService.Agent.Ontology` è si trova un'ontologia base chiamata *BasicOntology*. Questa ontologia fornisce gli schemi per i principali tipi primitivi, per un concetto di larghissimo impiego, ovvero AID, oltre allo schema per il messaggio di tipo FIPA-ACL e gli schemi per gli operatori SL0, illustrati successivamente.
Questa ontologia base è sufficiente per un tipico utilizzo di un'ontologia. Se si rendessero necessarie delle aggiunte, si può sempre pensare di estenderla con una propria ontologia base.

7.2.2 Aggiunta degli schemi e degli slot

Nel costruttore privato dell'ontologia vengono aggiunti gli schemi che compongono l'ontologia e questi vengono dotati degli slot opportuni.
Relativamente ai concetti introdotti più sopra nel vocabolario, si illustra il codice per la creazione di schemi e slot:

```
Add(new ConceptSchema(cd), typeof(CD));
Add(new ConceptSchema(item), typeof(Item));
Add(new ConceptSchema(track), typeof(Track));
```

Questo è il codice per creare gli schemi. Come si vede, si aggiungono schemi relativi al concetto *CD*, *Item* e *Track*, associandoli alle classi relative agli schemi. In particolare, per esempio, è creato un *ConceptSchema* con il nome uguale al contenuto della stringa *item*.

Ad esso è associata la classe *Item* che rappresenta il concetto *item*, la quale sarà usata dagli sviluppatori degli agenti.

```
[Serializable]
public class Item: Concept
{
      private int serial_number;
      public int Serial_number
      {
            set
            {
                  this.serial_number=value;
            }
            get
            {
                  return this.serial_number;
            }
      }

      public Item()
      {
            serial_number=0;
      }
}
```

Nel codice riportato qui sopra è definita la classe *Item*. Essa ha una proprietà *Serial_Number* che rappresenta l'unico slot del concetto *item*. Il nome della proprietà e dello slot devono essere uguali, anche se la proprietà deve avere la prima lettera maiuscola. La classe deve essere serializzabile, poiché le sue istanze dovranno viaggiare come contenuto dei messaggi.
Inoltre, ai fini della validazione, la classe *Item* deve implementare l'interfaccia (vuota) *Concept* (disponibile in `AgentService.Agent.Ontology.Interfaces` assieme alle interfacce che descrivono gli altri generici elementi delle ontologie: predicati, azioni, primitive…). Ogni classe associata ad uno schema ontologico deve implementare l'interfaccia corrispondente.
Per associare agli schemi gli slot che competono loro, occorre richiamare il metodo Add:

```
ConceptSchema cs = (ConceptSchema) GetSchema(item);
cs.Add(item_serial,(PrimitiveSchema)GetSchema(BasicOntology.integer),
                                                  ObjectSchema.optional);
```

Allo schema *item* viene assegnato lo slot *item_serial*, di tipo *PrimitiveSchema*. Questo slot rappresenta un valore intero. Lo schema associato al tipo intero è un *PrimitiveSchema* definito nella *BasicOntology*.
Inoltre lo slot è reso opzionale, utilizzando l'indicazione contenuta nel dato membro optional in *ObjectSchema*.
Con questo si esaurisce l'implementazione di un'ontologia. Ricapitolando, per creare un'ontologia bisogna espletare i seguenti punti:

- Creare una classe per la propria ontologia, avendo cura di farla derivare da *Ontology*.

- Creare un vocabolario di termini rappresentanti i nomi per ogni schema e attributo.
- Definire una proprietà *Instance* contenente l'istanza dell'ontologia.
- Creare un costruttore privato che richiami il costruttore di *Ontology*, passandogli nome dell'ontologia, eventuali ontologie base e introspector.
- Nel corpo del costruttore aggiungere prima gli schemi relativi all'ontologia e poi dotarli di slot.
- Creare le classi associate agli schemi, avendo cura di creare le relative proprietà associate agli schemi, con il nome uguale a quello dato agli slot corrispondenti. Tali classi devono implementare le interfacce corrispondenti al tipo di schema cui sono associate.

7.2.3 Progettazione con Protégé

Per evitare di programmare le classi ontologiche, procedimento ripetitivo e facilemente automatizzabile, si può utilizzare *Protégé*: un editor per ontologie personalizzato per progettare le ontologie di AgentService.
Per prima cosa, bisogna aprire il file di progetto *TemplateOnto.pprj* contenente le metaclassi che descrivono il concetto, il predicato, l'azione…
Dopodichè si specializzano queste metaclassi creando gli schemi propri dell'ontologia. Per creare il concetto *item*, si clicca con il tasto destro sulla metaclasse *Concept* e si seleziona *Create Class*.
Per dotare *item* di uno slot si clicca sulla listbox *TemplateSlot* e si seleziona *Create slot*. Si dà il nome allo slot, si seleziona il tipo, si seleziona *required* se lo slot è obbligatorio e si indica la cardinalità minima e massima tramite *at least* e *at most*.
Non è possibile dare a due slot di schemi diversi lo stesso nome, a meno che non si voglia riutilizzare lo stesso slot (azione possibile, se invece di selezionare *Create slot* si seleziona *Add slot*). Questa procedura può portare a dei cambiamenti indesiderati, perché se si riutilizza uno slot, può accadere che in uno schema sia obbligatorio, mentre nell'altro no: riutilizzando lo stesso slot, non è possibile diversificare.
Allora bisogna selezionare il Tab Widget *Slots*, selezionare lo slot in questione e cliccando con il tasto destro selezionare *Change Slot Meta-class*. Invece dello *Standard-slot* si indica la sua specializzazione *AgentService-slot*.
Aprendo la form per la configurazione dello slot (dal Tab Widget *Slots* o dal solito *Classes*) si notano tre campi in più. Quello che interessa è *AgentService-Name*; allo slot si dà un nome qualsiasi nel campo *Name*, mentre nel campo *AgentService-Name* si dà il nome desiderato (che può coincidere con il nome di un altro slot).
Un altro campo definito nell'*AgentService-Slot* è *AgentService-TypedAggregate*, da usare nel caso lo slot sia con cardinalità massima maggiore di 1 e il tipo di Aggregato non sia standard (si ricordi il discorso sull'introspector).
Nel caso si indichi un aggregato non standard, ovviamente si rende necessaria la creazione dello schema relativo all'aggregato.

Se uno slot non è di tipo primitivo, ma è per esempio un concetto definito nell'ontologia, nel campo *Value Type* si seleziona *Class*, quindi nel nuovo campo sottostante si seleziona lo schema che rappresenta lo slot.
Se lo slot è rappresentato da uno schema contenuto in un'ontologia base, si rende necessario creare uno schema fittizio nell'ontologia corrente (anche privo di slot) e nella finestra di configurazione della classe si seleziona il campo AgentService-Ignore, in modo da non rappresentarla nell'ontologia corrente.

7.2.3.1 IRE

Nel caso dell'IRE la creazione dello schema comporta una procedura particolare. Per prima cosa si specializza IRE con la propria query.
Preventivamente occorre aver definito una *variabile* e un *predicato* contenente anche tale variabile.
Una volta creata la propria IRE, si seleziona il Tab Widget *Instances*, si seleziona l'IRE e si crea un'istanza premendo il pulsante *Create Instance* in *Instance Browser*. Infine si indica la variabile nello slot *variable* e il predicato nello slot *proposition*.

7.2.3.2 Ontology

Prima di ultimare il progetto dell'ontologia, bisogna creare un'istanza della classe *Ontology*, con la procedura descritta per IRE. Si deve obbligatoriamente indicare un nome ed eventuali ontologie base e nuovo introspector.

Ora il progetto è finito e si può salvare, avendo cura di esportare il progetto, tramite il menu *file*, selezionando *Convert Project To Format* e quindi *Experimental XML*.

7.2.4 Protege2AgentService

Per compilare l'assembly contenente l'ontologia si utilizza il software *Protégé2AgentService* il quale riceve in ingresso il file xml precedentemente esportato e genera i file di codice C# e assembly, memorizzati nella cartella desiderata.

7.3 SL

SL è un linguaggio per esprimere contenuti. Secondo le specifiche FIPA va utilizzato per rappresentare i contenuti dei messaggi.
In AgentService si utilizza il sottolinguaggio minimale SL0 con qualche estensione.
SL0 fornisce degli operatori che trasportano al loro interno le istanze delle classi ontologiche:

- *Action*: ha due proprietà:
  - Actor: indica l'AID di chi compie l'azione.
  - Act: indica l'azione da compiere.

- *Done*:
  - Action: indica l'azione.
  - Condition: indica l'esito dell'azione. Si tratta di un predicato (*TrueProposition*, *FalseProposition* o qualsiasi altro predicato…).
- *Result*:
  - Action: indica l'azione.
  - Value: indica il risultato dell'azione, ovvero un oggetto creato tramite l'azione.
- *Equals*: operatore di uguaglianza che mette in relazione un oggetto Left con un oggetto Right.

Con SL0 sono forniti anche due proposizioni di larghissimo impiego: *TrueProposition* e *FalseProposition* i quali esprimono il Vero e il Falso.
Le modalità di impiego sono le seguenti:

- L'istanza di un'azione è veicolata dall'operatore *Action*. La risposta all'azione può essere data con *Done* o *Result* a seconda che si voglia esprimere solo l'esito o anche un risultato.
- Predicati e IRE viaggiano senza essere ospitati in alcun operatore. Le risposte ai predicati si esprimono con *Equals* mettendo a primo membro il predicato e a secondo membro il valore vero o falso. Per quanto riguarda le IRE, a primo membro si indica l'IRE mentre a secondo membro il risultato dell'IRE (un oggetto od una lista di oggetti).
- Solo azioni (tramite Action), predicati e IRE possono viaggiare come contenuto primario di un messaggio: tutto il resto può essere ospitato solo nei loro slot.

Le classi che implementano questi operatori si trovano nel namespace `AgentService.Agent.Ontology.SL0`. Questi operatori sono rappresentati come schemi nella *BasicOntology*, così che all'atto della validazione anch'essi vengono convalidati.
E' opportuno quindi indicare fra le ontologie base della propria ontologia anche la *BasicOntology*.

7.4 Conversazioni con supporto ontologico

Per poter usare praticamente le ontologie nelle conversazioni fra agenti, sfruttando anche la funzionalità di validazione del contenuto dei messaggi, in modo che sia conforme con le regole dell'ontologia utilizzata, occorre utilizzare un particolare tipo di conversazione: *IOntologicalConversation*.
Il suo utilizzo è analogo a quanto descritto nel paragrafo 4.2.4. In più essa offre le seguente proprietà:

- `Ontology TheOntology;`
  Restituisce l'istanza corrente dell'ontologia.

- `bool Strict;`
  Se *true*, impone che il processo di validazione sia imposto a tutti I *BodyItem*. In caso contrario, solo il *BodyItem* contenente un operatore SL sarà oggetto di validazione. Questo consente di utilizzare in modo flessibile il supporto ontologico,

introducendo anche contenuti non propriamente ontologici (quindi non soggetti a validazione).

- `string ValidationError;`
  Ritorna la stringa contenente la descrizione dell'errore che ha originato l'eccezione di validazione.

Inoltre *IOntologicalConversation* dispone dei seguenti metodi:

- `bool SendQuery(IRE query);`
  Invia una IRE, evitando all'agente l'onere di formulare l'operatore SL adeguato per inviare l'IRE. Ritorna *true* se il processo è andato a buon fine, in caso contrario occorrerà verificare il *ValidationError*.

- `bool SendPredicate(Predicate predicate);`
  Analogamente, questo metodo permette di inviare un predicato.

- `bool SendAction(AgentAction action);`
  Il metodo invia un'azione.

- `bool SendActionDone(AgentAction action, bool outcome);`
  Viene inviata la risposta contenente l'esito di un'azione, pertanto si dovrà fare riferimento all'azione in questione e al suo esito.

- `bool SendActionResult(AgentAction action, object result);`
  Si invia il risultato di un'azione, che può essere un qualsiasi oggetto.

- `bool SendEqualsPredicate(object left, object right);`
  Invia il predicato '=', indicando ovviamente l'oggetto alla sinistra dell'uguale e quello alla sua destra.

7.4.1 Come aprire una conversazione

Per aprire una conversazione nel DefaultMessageClient, occorre richiamare il metodo OpenConversation, passando anche l'istanza dell'ontologia che si intende utilizzare, come si vede nell'esempio:

```
AgentService.Agent.Runtime.IOntologicalConversation conversation=
this.Runtime.MessageSvc.OpenConversation(new AID("Pippo", this.Owner.HapName),
MusicShopOntology.MusicShopOntology.Instance);
```

Come si può notare, il metodo OpenConversation in questione ritorna una *IOntologicalConversation*.

Aperta la conversazione si può procedere con l'invio e la ricezione di messaggi dal contenuto ontologico, usando i metodi indicati più sopra.

7.4.2 Conversazioni in ingresso

Anche l'agente interlocutore, deve gestire una conversazione ontologica, in modo, per esempio, da attivare il processo di validazione in ricezione (considerando quindi possibili interferenze che potrebbero corrompere il messaggio lungo la linea). Ecco il codice necessario per poter gestire la conversazione ontologica:

```
IOntologicalConversation conv = (IOntologicalConversation)this.IncomingConvs[0];
```

Il processo di validazione è eseguito automaticamente nei metodi di ricezione come *ReceiveMessage*, *WaitForMessage* etc. Ricordiamo che tramite la proprietà *Strict* della conversazione è possibile validare solo l'effettivo contenuto ontologico oppure ogni *BodyItem*.

La libreria AgentService

7.5 I namespace