SAF: Scala Agent Framework

Pagnini Lorenzo
lorenzo.pagnini2@studio.unibo.it

Antonelli Alberto
alberto.antonelli2@studio.unibo.it

Tentoni Daniele
daniele.tentoni2@studio.unibo.it

March 2021

Si vuole creare una libreria in Scala che fornisca un approccio semplice e il più possibile funzionale alla programmazione orientata agli agenti. Attraverso JADE e il linguaggio Java, spesso diviene molto lungo e complesso sviluppare sistemi ad agenti in confronto ad altre tecnologie. La libreria deve ridurre la verbosità dell'implementazione degli agenti e fornire delle Api che siano in grado di essere lette praticamente come il linguaggio umano, per favorire al massimo la comprensione del codice scritto.

Contents

1 Obiettivi **3**
- 1.1 Scenari di utilizzo . . . 3
- 1.2 Politica di autovalutazione . . . 4

2 Analisi dei requisiti **6**
- 2.1 Requisiti utente . . . 6
- 2.2 Requisiti non funzionali . . . 6
- 2.3 Requisiti intrinseco . . . 6

3 Progettazione **7**
- 3.1 Strumenti adottati . . . 7
  - 3.1.1 VCS . . . 7
  - 3.1.2 Build Automation . . . 8
  - 3.1.3 Continuous Integration . . . 8
  - 3.1.4 Bacheca di lavoro . . . 9
- 3.2 Struttura . . . 9
  - 3.2.1 Containers . . . 9
  - 3.2.2 Agenti . . . 10
  - 3.2.3 Stato . . . 11
  - 3.2.4 Comportamenti / Behaviours . . . 12
  - 3.2.5 Messaggi . . . 14
  - 3.2.6 Templates . . . 14
- 3.3 Comportamento . . . 14
- 3.4 Interazione . . . 15

4 Dettagli implementativi **17**

5 Autovalutazione **19**

6 Conclusioni **22**
- 6.1 Lavori futuri . . . 22
- 6.2 Che cosa abbiamo imparato . . . 23

7 Istruzioni per il rilascio **24**
- 7.1 Compilare il progetto . . . 24
- 7.2 Aggiungere la libreria . . . 24

8 Esempi di utilizzo **25**
- 8.1 Sintassi . . . 25

1 Obiettivi

La libreria è un wrapper di JADE, come quella sviluppata durante le prime lezioni di laboratorio, ma scritta in Scala. Deve mettere a disposizione delle Api per poter creare gli agenti, assegnargli dei comportamenti e metterli in esecuzione. Deve essere possibile mettere in comunicazione più agenti all'interno dello stesso ambiente.

Per meglio esprimere questi concetti forniamo un esempio dell'uso di Jade in Java:

```
1  public class DateAgent extends Agent {
2    private int counter;
3
4    @Override
5    protected void setup() {
6      counter = 10
7      addBehaviour(oneShotBehaviour());
8    }
9
10   private Behaviour oneShotBehaviour() {
11     return new OneShotBehaviour() {
12        @Override
13        public void action() {
14            System.out.println("The value of counter is: " +
                  counter);
15        }
16     };
17   }
18 }
```

e il suo corrispondente in Scala usando SAF:

```
1 val counter : SafState[Int] = 10.state
2
3 val behaviour = oneShot { a =>
4   val state = a.internal.get("counter")
5   println(s"The value of ${state name} is: ${state.get}")
6 }
7
8 Agent("Jade Agent") state counter setup addBehaviour behaviour
```

In modo funzionale e più conciso è stato possibile scrivere lo stesso agente.

1.1 Scenari di utilizzo

Gli utenti possono integrare ed utilizzare la libreria nei loro progetti Scala. In particolare, si è pensato di re-implementare le principali funzionalità che Jade mette a disposizione

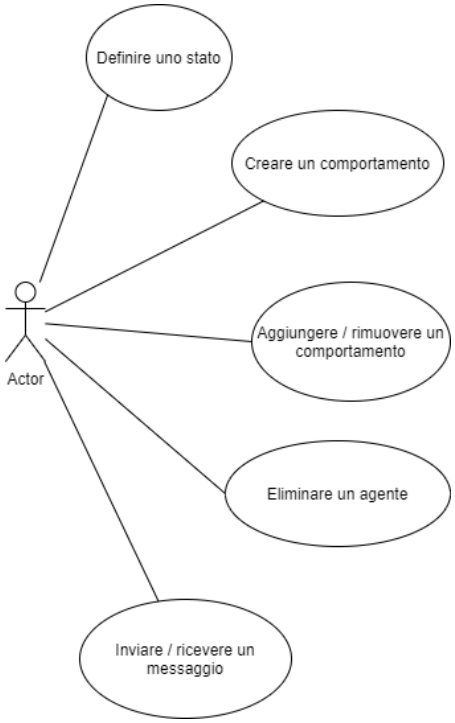


Figure 1: Diagramma dei casi d'uso di un agente

per la gestione degli agenti. L'utente potrà definire dei container e, al loro interno, degli agenti. Per ogni agente sarà possibile associarli un nome, uno stato e descrivere il suo comportamento all'interno della funzione *setup*.

I principali vantaggi per i quali l'utente dovrebbe utilizzare la nostra libreria sono:

- Istanziare gli agenti e i behaviours e accedere alle relative primitive in modo funzionale, risparmiando tempo;
- Migliorare la leggibilità, riducendo la verbosità e il numero di righe di codice necessarie alla generazione degli agenti.

1.2 Politica di autovalutazione

Per verificare l'efficacia della libreria, abbiamo prodotto un'applicazione d'esempio che usi la programmazione ad agenti, sia in linguaggio Scala utilizzando SAF che in linguaggio Java utilizzando Jade. L'applicazione scritta in Scala deve risultare più compatta e leggibile.

Il processo di autovalutazione mira a raggiungere i requisiti preposti basandoci sulla:

- scelta di adottare uno sviluppo agile del software: in particolare si è deciso di seguire un approccio di tipo Test Driven Development;

- realizzazione dei test per verificare il funzionamento corretto delle singole funzionalità messe a disposizione. In particolare verrano sviluppati test per verificare le 'azioni' che un agente può effettuare e tutti i comportamenti che possono essere eseguiti;

- qualità del software prefissandoci come obiettivo una *code coverage* non inferiore al 70%;

- efficacia dei risultati del progetto rappresentata dalla facilità d'uso da parte del programmatore del sistema distribuito ad agenti della nostra libreria.

2 Analisi dei requisiti

2.1 Requisiti utente

Di seguito verranno elencati i principali requisiti utente:

- Possibilità di generare agenti;
- Assegnargli uno stato;
- Definirne i comportamenti;
- Gestire il loro ciclo di vita;
- Permettere la comunicazione tra di essi;
- Possibilità di poter testare il codice prodotto dall'utente.

2.2 Requisiti non funzionali

Di seguito verranno elencanti i principali requisiti non funzionali:

- Sarà necessario realizzare API minimali, concise e autoesplicative. Dovrà essere lasciato all'utente la possibilità di implementare soltanto le parti necessarie alla definizione e implementazione degli agenti;
- Sarà necessario implementare API efficienti;
- Il numero degli agenti che possono essere implementati in una piattaforma non deve considerarsi un vincolo.

2.3 Requisiti intrinseco

- Sarà necessario riadattare il modello ad agenti di Jade per soddisfare i requisiti di progetto.

3 Progettazione

In un primo momento ci siamo consultati, attraverso delle call, per discutere sulla scelta della migliore architettura da adottare. Durante questa fase abbiamo realizzato diversi schemi UML. Questi, ci hanno aiutato nel organizzare la struttura dell'applicativo e nel avviare la successiva fase di implementazione. Questa fase si è conclusa quando abbiamo ottentuto un'architettura che ci convincesse, in grado di coprire tutte le funzionalità che volevamo realizzare.

Nella fase successiva, di implementazione, sono emerse delle problematiche che non si erano considerate nel precedente step, che ci hanno portato a modificarne la struttura. Abbiamo lavorato cercando di creare la migliore architettura da realizzare producendo codice che potesse essere utile per suddividere il lavoro tra i componenti del gruppo, tralasciando in un primo momento la possibiltà di refactoring del codice. Solo in una fase successiva, ove possibile, si è lavorato per ridurre la verbosità della libreria.

Come detto, il design dell'applicativo è stato rivisitato più volte. Il gruppo ha deciso di strutturare il progetto attraverso un approccio incrementale, partendo dalla creazione della piattaforma (container e agenti) e aggiungendo mano a mano le altre funzionalità (stato, creazione dei behaviour, comunicazione degli agenti) e del refactoring delle parti precendenti per portare una maggior leggibiltà della libreria.

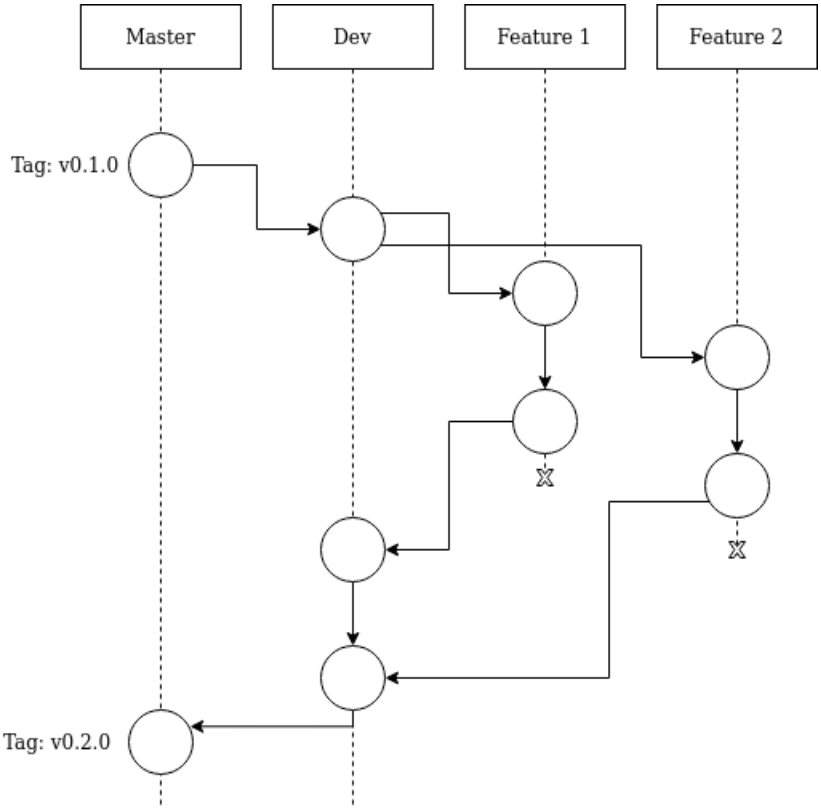
Il linguaggio adottato per questo progetto è Scala. Il motivo per il quale si è deciso di utilizzare questo linguaggio è dato dal fatto che, estendendo la Java Agent Development Framework attraerso SAF è possibile utilizzarlo in un modo maggiormente funzionale attraverso tale linguaggio.

Questo sicuramente porta con se i vantaggi di Scala. Basandosi su Java e sulla JVM è possibile utilizzare qualsiasi libreria compatibile con Java stesso, sfruttando i collegamenti che tra i due linguaggi iniziano a farsi sempre più sottili, visto anche l'inserimento delle lambda function e il costrutto var in Java 14 già presenti in Scala.

3.1 Strumenti adottati

3.1.1 VCS

Si è deciso di utilizzare *Git* per effettuare il versioning del codice durante lo sviluppo attraverso la piattaforma *GitLab*.



L'utilizzo che ne abbiamo fatto è descritto nella figura: il branch di default è sempre il *master*, al quale *development* è sempre allineato. Ogni volta che si implementava una nuova funzionalità o si risolveva una fix, veniva creato una branch a parte. Successivamente, a lavoro ultimato, veniva creata una merge request per mergiare sul branch *dev*. Ogni merge request su *development* deve essere approvata almeno da un altro membro del gruppo.

3.1.2 Build Automation

Per la build automation si è deciso di usare *SBT* poichè è il sistema ufficiale adottato da Scala.

3.1.3 Continuous Integration

Si è deciso di usare la *CI* di GitLab, già utilizzata in altri progetti. Questa ci ha permesso di automatizzare l'esecuzione dei test necessari per verificare la correttezza del lavoro svolto e poter poi effettuare dei rilasci senza regressioni.

3.1.4 Bacheca di lavoro

Per avere una visione sull'andamento generale del progetto e sulle specifiche feature da svolgere o completate, abbiamo utilizzato *Trello*.
Si sono realizzate delle etichette personalizzate da associare alle feature da implementare per una migliore organizzazione per ogni membro del team. Inoltre si sono create delle colonne in cui raggruppare le feature a seconda del suo stato di sviluppo. Alcune di queste sono:

- *To Do*: contiene le card associate alle features da sviluppare;
- *Done*: contiene le card associate alle features sviluppate;
- *Bug*: contiene le card associate alle features che presentano bug da 'fixare';
- *Sprint*: contiene le card che riassumono le features da svolgere per ogni sprint.

La bacheca è possibile consultarla direttamente cliccando il link: Trello.

3.2 Struttura

3.2.1 Containers

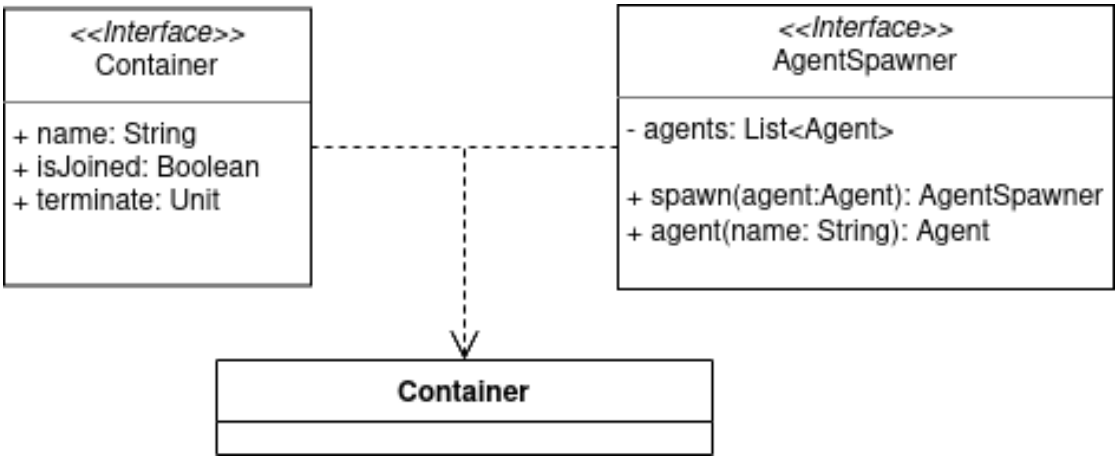


Figure 2: Diagramma dei casi d'uso di una piattaforma

Rappresenta l'astrazione di un contenitore dentro cui possono essere eseguiti gli agenti. Nell'immagine sono riportati i casi d'uso dell'utente. Le operazioni sono basilari ma sufficenti per coprire l'ambito del progetto.

Viene data la possibilità di creare un agente all'interno del container, data una con-

figurazione iniziale dell'agente e un insieme di comportamenti da aggiungere ad esso all'atto della creazione prima dell'avvio. Allo stato attuale non è possibile creare un agente dentro ad un container senza avviarne il ciclo di controllo dei comportamenti.

In qualunque momento è possibile vedere la lista degli agenti creati all'interno di un container e ricercarne uno specifico tramite il nome dato al momento della creazione.

3.2.2 Agenti

All'interno del progetto sono stati definiti diversi tipi di agenti, assegnado ad ognuno di essi precise funzionalità. Questo per dare una sequenzialità nell'esecuzione delle API offerte.
Se avessimo creato un unico agente all'interno del sistema, si dava la possibilità all'utente finale (o utilizzatore) di poterle utilizzare senza un ordine logico. Per fare un esempio, nel definire un agente, non è necessario prima creare uno stato, ma deve essere necessariamente definito almeno un comportamento da eseguire alla creazione dell'agente all'interno di un container. Solamente dopo aver completato questa fasi deve essere possibile aggiungere un agente all'interno di un container.

Gli agenti presenti nell'architettura sono i seguenti, elencati nell'ordine di utilizzo dal sistema:

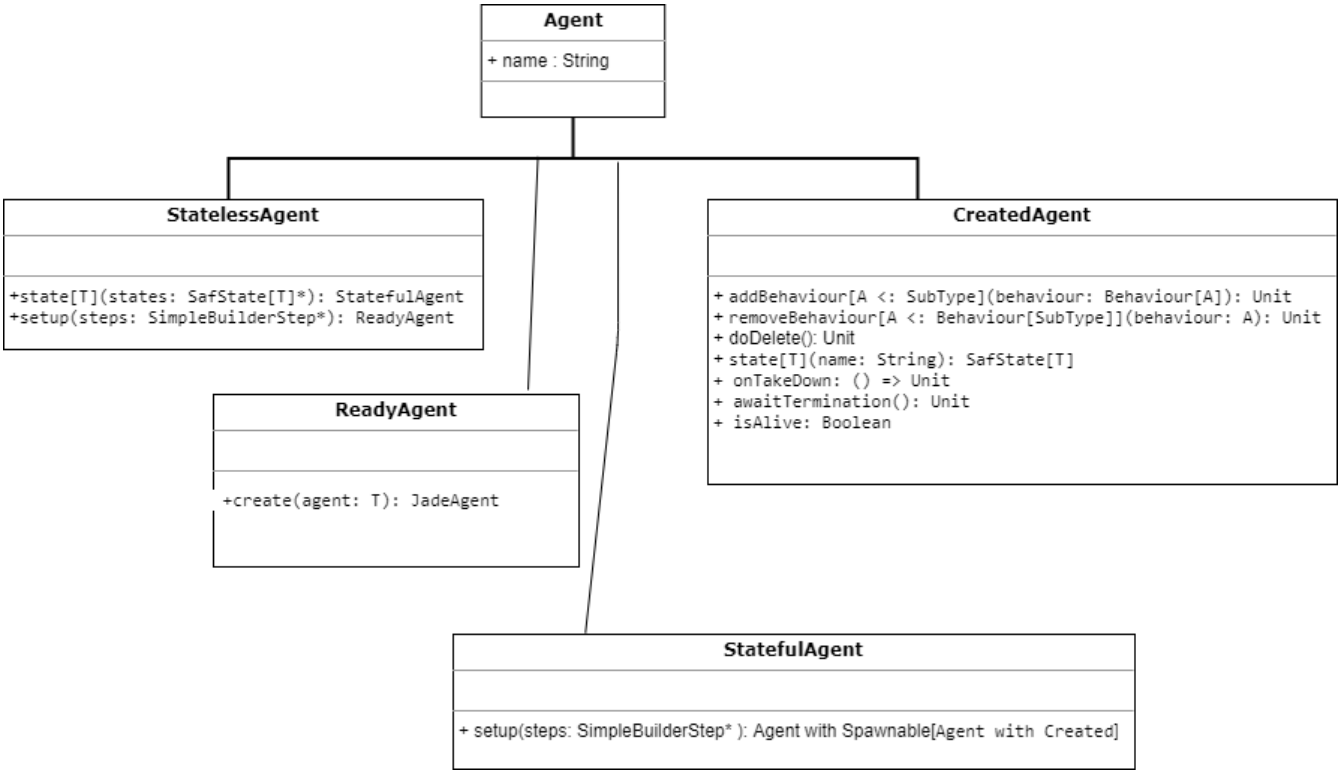
- *Stateless*: Un nuovo agente si trova in questo stato appena dichiarato (vedi appendice per la dichiarazione di un agente). Da questo stato deve essere possibile definire lo stato dell'agente oppure dichiarare subito i Behaviours da eseguire al suo interno;

- *Stateful*: Un agente dopo aver inizializzato il suo stato interno deve poter dichiarare i comportamenti da avere al suo interno;

- *Ready*: Dopo che l'utente ha dichiarato i comportamenti, l'agente si trova in questo stato. Da qui è possibile creare l'agente presso un container.

- *Created*: L'agente si trova in questo stato dopo che è stato creato all'interno di un Container. In tale momento, espone le funzionalità per aggiungere e/o rimuovere Behaviours, per gestire la comunicazione con altri agenti e il suo ciclo di vita;

Si può intervenire sul ciclo di vita di un agente tramite l'operazione *doDelete*. Qualunque sia l'attuale stato di vita dell'agente, effettua una transizione dell'agente a deleted, eliminandolo dal container.



3.2.3 Stato

Durante la discussione dell'architettura da adottare, abbiamo pensato fosse utile creare uno stato esterno a Jade, così da poter gestire più facilmente l'interazione con lo stato all'interno dell'esecuzione di un agente. Ogni agente gestice uno stato globale (*GlobalState*). Quest'ultimo è composto da tanti stati semplici (*SimpleState*) che memorizzano il nome dello stato e il suo valore.

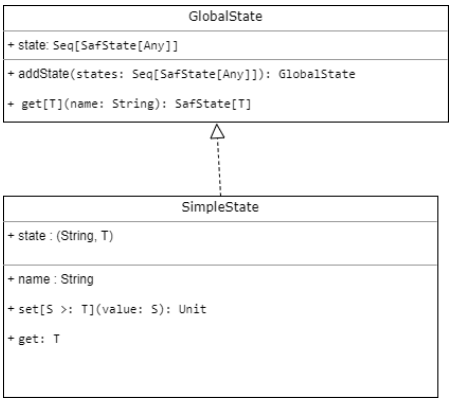


Figure 3: Stato di un agente

3.2.4 Comportamenti / Behaviours

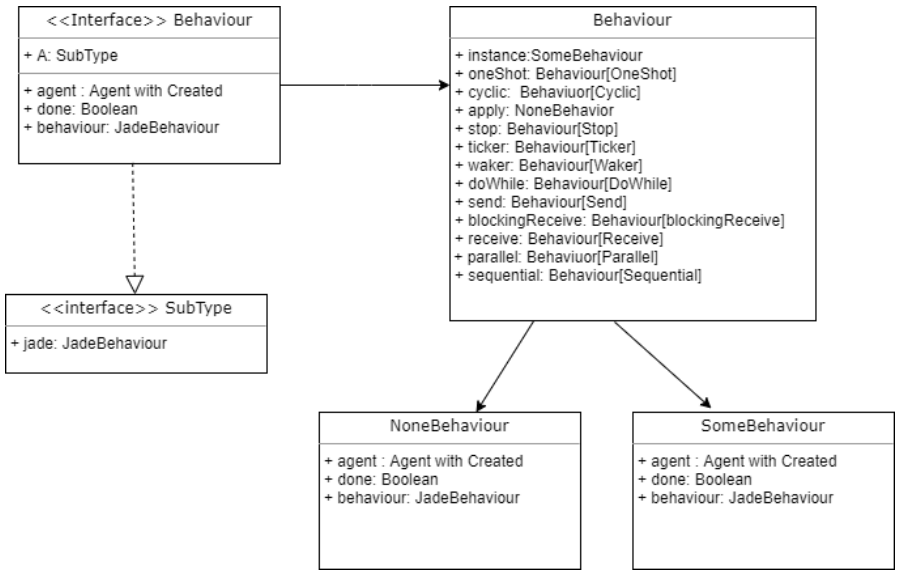


Figure 4: Interfaccia Behaviour

Da come si vede nell'immagine (figura 4) un Behaviours può essere di due tipi:

- NoneBehaviour: rappresenta un behaviour "non instanziato", ossia non tiene traccia dell'agente che aggiunge quel comportamento.
- SomeBehaviour: rappresenta un behaviour "instanziato", ossia contiene un riferimento all'agente della libreria Jade che aggiuge quel determinato comportamento.

Il secondo di questi, risulta particolarmente utile durante l'aggiunta dei Behaviour composti che vedremo dopo, anche se è stata presa in considerazione la possibilità di entrambe le tipologie per tutti i behaviour. Nel capitolo dei dettagli implementativi sarà specificato il motivo di questa divisione.

Elencando i comportamenti che è possibile utilizzare all'interno degli agenti offerti dal DSL, si è deciso di mantenere la suddivisione creata da Jade in comportamenti semplici e composti. Di seguito vengono elencati quelli che possono essere utilizzati:

1. **Semplici**
    - **OneShot**: prevede un'unica esecuzione di un insieme di operazioni;
    - **Cyclic**: prevede una ripetizione ciclica di un insieme di operazioni fintanto che l'agente è in esecuzione;

2. **Composti**

   - **Parallel**: prevede l'esecuzione parallela di più comportamenti. Deve essere possibile indicare delle politiche di terminazione (ad esempio termina quando un qualunque comportamento termina, quando tutti terminano, ecc...);

   - **Sequential**: prevede l'esecuzione di una sequenza di comportamenti schedulati sequenzialmente uno dopo l'altro. Termina quando l'ultimo comportamento della sequenza viene completato;

   - **Waker**: prevede l'esecuzione ritardata, per un periodo di tempo dato in input, di un comportamento OneShot;

   - **Ticker**: prevede l'esecuzione periodica, per un periodo di tempo dato in input, di un comportamento.

Inoltre SAF vuole estendere tale insieme anche con i seguenti comportamenti non presenti in Jade:

- **DoWhile**: estende il comportamento del cyclic. Permette l'esecuzione ciclica aggiungendo una condizione di terminazione definita dall'utente. !!! Riguardare le slide e la lezione di Ciatto.

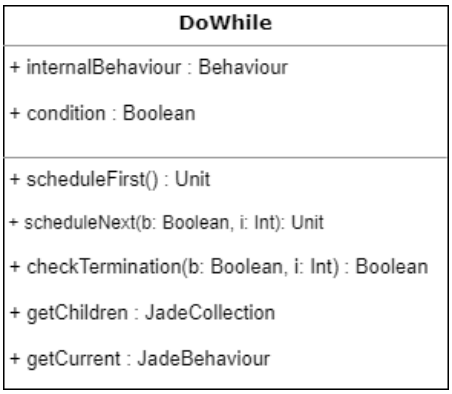


Figure 5: Comportamento DoWhile

- **Send**: invia un messaggio ad uno o più destinatari. I destinatari sono identificati con il nome dell'agente definito in fase di creazione.

- **Receive**: permette di ricevere un messaggio (e quindi di rimanere in ascolto) in maniera non bloccante (asincrono). Inoltre è possibile specificare due parametri: il primo è una funzione che rappresenta il comportamento da eseguire dopo la ricezione del messaggio, il secondo è il template del messaggio che si vuole ricevere. Questo significa che se l'agente riceve un messaggio con un template diverso

da quello specificato, tale messaggio verrà ignorato. Se non viene specificato, il template di default è composto solamente da una *performative* di tipo info.

- **Blocking Receive**: permette di ricevere un messaggio (e quindi di rimanere in ascolto) in maniera bloccante (sincrono). I parametri che questo behaviour definisce sono gli stessi delle Receive spiegati al punto precedente.

- **Stop**: termina l'esecuzione dell'agente in modo "gentile", cioè permettendogli di rilasciare eventuali risorse allocate e avvisando l'ambiente della sua prossima terminazione.

3.2.5 Messaggi

I messaggi rappresentano in SAF l'unica 'entità' che è possibile inviare ad altri agenti. In particolare si è deciso di definire una case class `Message` composta da due campi: il contenuto del messaggio che verrà letto dai destinatari e il *performative*. Quest'ultimo è un codice numerico che identifica il tipo di mesasggio inviato (ad esempio può rappresentare una richiesta, un rifiuto, una risposta, ...).

Come spiegato nella prossima sezione, i messaggi possono essere utilizzati per creare dei template specifici a secondo delle esigenze specifiche.

3.2.6 Templates

I template identificano una possibile struttura di un messaggio utilizzati dagli agenti per *classificare* i messaggi che devono essere inviati e ricevuti attraverso i relativi comportamenti di SAF.

Visto il gran numero di funzioni *ad alto livello* che JADE mette a disposizione per la creazione di un template, si è deciso di non re-implementare la classe template SAF ma di utilizzare quella fornita dalla piattaforma JADE. Questo significa che gli agenti utilizzeranno le classi JADE per creare dei messaggi da incorporare, se necessario, all'interno di uno o più template.

3.3 Comportamento

Un'entità Container genera al suo interno gli agenti date le istruzioni dall'utente e vi aggiunge i comportamenti in esso dichiarati.

Una volta aggiunti ad un Container, gli Agenti vengono subito messi in esecuzione, iniziando quindi il loro ciclo di controllo dei comportamenti fino a quando non ven-

gono terminati forzatamente tramite un comportamento al loro interno o un segnale esterno. Ad ogni esecuzione del ciclo, viene controllato se sono presenti ancora uno o più comportamenti non ancora terminati. Per esempio, un comportamento OneShot dopo una esecuzione del ciclo di controllo, viene eseguito. Successivamente, verrà marcato come terminato e non verrà mai più eseguito dal ciclo di controllo. Un comportamento Cyclic, al contratio, non verrà mai marcato come eseguito e continuerà ad essere messo in esecuzione.

I messaggi vengono creati dagli agenti che devono inviarli, specificandone il contenuto e i destinatari. A seconda del loro contenuto, è possibile definire i template che, confrontati con esso, possono restituire un match positivo. Tra i Containers è possibile scambiarsi messaggi sapendo il nome del Container a cui inviarli e il nome dell'Agente destinatario.

Di seguito viene presentato un semplice schema di esecuzione di una creazione di un agente, della sua aggiunta ad un Container e della sua terminazione.

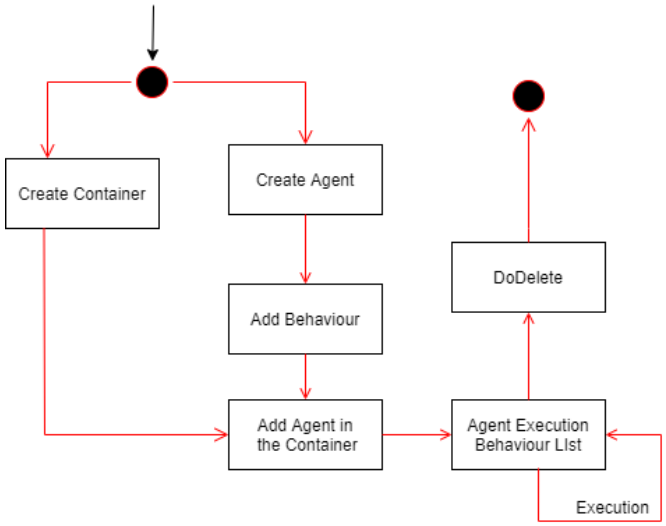


Figure 6: Esecuzione standard

3.4 Interazione

Gli agenti possono comunicare tra loro attraverso l'esecuzione di comportamenti che prevedano l'invio e la ricezione di messaggi. I template possono essere confrontati con messaggi e resituire quindi un risultato positivo o no. In questo modo, gli agenti possono leggere solamente i messaggi che interessano loro. A seconda del loro stato, gli agenti eseguono i diversi comportamenti che sono stati definiti al suo interno. Tali comportamenti, nel caso in cui definiscano un cambiamento di stato dell'agente, eseguono

la transizione specificata verso lo stato di destinazione e sucessivamente ne riprendono l'esecuzione.

4 Dettagli implementativi

Per quanto riguarda il linguaggio in cui il sistema è stato sviluppato, la scelta è ricaduta, come anticipato in precedenza, su Scala.

Nell'implementazione della libreria, si è cercato di rifarsi alla struttura del framework sviluppato in Java, estendendo ciò era necessario. Riteniamo che sarebbe stato inutile e anche non corretto replicare Jade in Scala, quindi consideriamo questa scelta adeguata con ciò che abbiamo realizato.

Il codice prodotto per il progetto è stato correttamente documentato, cercando di attenersi il più possibile alle linee consigliate nel utilizzo della Scaladoc, mantenedo uno stile coerente con essa. I principali aspetti rilevanti dell'implementazione che di più si distolgono dall'implemetazione Jade sono:

- La presenza di NoneBehaviuor e SomeBehaviour;
- La diversa tipologia di agenti.

La differenzazione tra NoneBehaviour e SomeBehaviour deriva da un accorgimento implemetativo successivo all'analisi iniziale e alla struttura che si era stabilita. Infatti durante la creazione dell'agente, in particolare nel trait spawner siamo caduti in un bug che non permetteva la sua creazione all'interno di un container. Per risolvere questa problematica si è optato per la creazioni di un NoneBehaviour per un qualsiasi behaviour che volesse essere aggiunto, per poi instanziarlo come SomeBehaviuor solo qualora fosse richiamato l'agente stesso.
Oltre a questa motivazione sono stati creati per avere una maggior possibilità di creazione di un comportamento. Il primo permette la creazione senza dover passare l'agente stesso. Al contrario il SomeBehaviuor obbliga il collegamento con l'agente che lo crea. Questa seconda possibiltà risulta utile nei Parallel e Sequential behaviour dove per eseguire diversi comportamenti al loro interno, secondo le condizioni richieste dai due comportamenti composti, risulta obbligatorio avere l'agente che gli esegue.

```
1  def instance[A <: SubType](b: Behaviour[A], a: Agent with Created):
       SomeBehaviour[A] = b match {
2    case NoneBehaviour(sub) => SomeBehaviour(sub, a)
3    case SomeBehaviour(sub, _) => SomeBehaviour(sub, a)
4  }
```

Come precedente descritto nel capitolo 3.1.2 sono stati definiti diversi tipi di agente, dove ad ognuno sono ricollegate specifiche funzionalità. Questa scelta è stata fatta principalmente per due ragioni, la prima per evitare la creazione di un agente all'interno di un container senza che esso possa esporre le funzionalità di creazione dei behaviour, la seconda per rendere la stessa creazione maggiormente strutturata. Come detto in precedenza le varie tipologie sono:

- Stateless;
- Stateful;
- Ready;
- Created.

In questo modo sarà impossibile creare un agente di tipo StateFul (agente che deve ancora dichiarare i suoi behaviour) all'interno di un container, infatti esso sarà obbligato ad eseguire il medoto setup che creerà un ReadyAgent. Il passo successivo sarà la sua creazione all'interno di un container.

Un altra nota implementativa che riteniamo di elencare è quella relativa all'oggetto Setup Steps. Questa contiene i metodi della setup come l'aggiunta e la rimozione di un behaviour, l'invio e la ricezione di un messaggio e il metodo di eliminazione dell'agente stesso.

Nell'aggiunta di un behaviour viene riportata un instanza di "addBehaviour Builder Step" dell'agente e ad esso può essere passata anche una funzione, in modo da portare maggior leggibilità.

Una particolare nota implementativa va fatta sull'utilizzo della CI di GitLab che svolge un ruolo fondamentale per il sistema creato. La CI infatti è stata configurata per effettuare la build del progetto ed eseguire i test.

5 Autovalutazione

L'immagine seguente mostra la coverage raggiunta nei test realizzati della nostra libreria.

Lines of code:	1935	Files:	15	Classes:	45	Methods:	110
Lines per file:	129.00	Packages:	9	Classes per package:	5.00	Methods per class:	2.44
Total statements:	336	Invoked statements:	248	Total branches:	16	Invoked branches:	9
Ignored statements:	0						
Statement coverage:	73.81 %			Branch coverage:	56.25 %		

Figure 7: Coverage aggregato Core + Testkit

All'interno del nostro progetto abbiamo utilizzato sbt come strumento di compilazione, così da sfruttare le sue principali caratteristiche quali il supporto nativo per la compilazione del codice Scala e l'integrazione con molti framework di test.
Oltre all'integrazione di un interprete Scala per debug rapidi e il supporto per progetti misti Java/Scala nel caso di eventuali difficoltà con la scrittura di codice Scala durante l'implementazione della libreria.

Avendo scelto l'approccio test-driven, come richiesto abbiamo previsto la stesura dei test prima di quella del software che verrà sottoposto al test stesso.

Riprendendo la struttura del progetto per prima cosa è stato necessaria la creazione di un container all'interno di una piattaforma, in modo che possano essere poi generati gli agenti al suo interno.

```
"A container" should {
    val containerName = "testContainer"
    val withoutName = Container()
    "be created without a name, giving him an UUID" in {
      assert(withoutName.name.nonEmpty)
      assertResult(true)(withoutName.isJoined)
    }
    val withName = Container(containerName)
    "be created with a name" in {
      assertResult(containerName)(withName.name)
      assertResult(true)(withName.isJoined)
    }
}
```

Successivamente il lavoro può essere quindi esteso alla creazione degli agenti e dei loro behaviuor. La stessa tipologia di test può essere effettuata per gli agenti.

```
"An agent" should "be created with a name" in {
    val withName: Agent = Agent("Jade Agent")
    assert(withName.name.nonEmpty)
  }
```

Una volta accertati della corretta creazione del container e dell'agente al suo interno il lavoro poi è stato suddiviso tra il refactoring e miglioramento di questi e la creazione ed agiunta di behaviour per i nostri agenti.

```
"start a simple agent" in {
    val agentName = "testAgent"
    val c = Container(containerName) spawn {
    Agent(agentName) setup {
        addBehaviour(oneShot {
            println("Hello World!")
            println("Hello World second!")
          })
        }
    }
    assert(c.name.contains(containerName))
    assert(c.agent(agentName).map(a => a.name).isDefined)
    println(c.agent(agentName).map(a => a.name))
    assert(c.agent(agentName).exists(a => a.name.contains(agentName))
}
```

Il precendente test ha il compito di controllare l'esatta creazione dell'agente all'interno del container, senza però preoccuparsi dell'esecuzione dei behaviour.

Per permettere l'aggiunta di funzionalità agli agenti che esulassero dalla stretta teoria è stato creato un modulo "Saf Testkit Library" distinto dal modulo core. Tra queste possiamo citare sicuramente la possibilità di estendere gli agenti con il trait Logger, che consente di scrivere delle stringhe dentro ad un buffer interno all'agente, come fossero log del sistema e leggerle anche dall'esterno dell'agente stesso. Tale funzionalità è stata essenziale per poter testare in modo più esaustivo le altre varie funzionalità degli agenti per il modulo core. In questo modulo quindi è stato fatta una re-implementazione di tutti in behaviour con l'utilizzo del logger.

Di seguito viene trascritto un esempio di questo utilizzo inerente al CyclicBehaviour implementato:

```
Feature("An Agent can log some strings to help developer to debug")
    Scenario("An Agent with a Cyclic Log Behaviour") {
      Given("An empty Container")
      val cyclicName = "cyclicAgent"
      Given("A Cyclic Log Behaviour")
      val i = 0.state
      val cyclicMessage = "Reached 10 logs"
      val b = cyclic { a: Agent with Created with Logger =>
        // Get the internal i state.
        val iState = a.state[Int]("i")
        if (iState.get < 10)
          iState.set(iState.get + 1)
        else if (iState.get == 10) {
          a.log(cyclicMessage)
```

```
15            a.doDelete()
16          }
17      }
18  }
```

Il test sopra riportato rappresenta il controllo del comportamento degli agenti durante l'esecuzione dei behaviour. Attraverso l'inserimento della log è stato quindi possibile visionare in console il successivo inserimento dei singoli behaviuor in una lista così da verificare la loro effettiva esecuzione.

Tornando al modulo core del nostro progetto, è stata testata inoltre la comunicazione tra gli agenti. Come previsto dalla libreria di Jade e come detto in precedenza sono state implementate la "Receive" e la "BlockingReceive".

La comunicazione tra due agenti prevede alcune possibilità che possono essere intervallate tra loro:

- L'appartenza o meno allo stesso container;
- La comunicazione attraverso l'utilizzo di un template;
- La necessità di risposta del destinatario.

Tenendo quindi conto di tutte queste possibilità, si sono testate la maggior parte delle combinazioni possibili nella comunicazione, come ad esempio l'invio di un messaggio con neccessità di risposta ad un agente che appartiene ad un altro container che ha una ricezione bloccante senza l'utilizzo di un template specifico.

Per concludere pensiamo che la coverage raggiunta e i test creati possano essere soddisfacenti a coprire la maggior parte del codice che è stato scritto.

6 Conclusioni

Alla fine del progetto possiamo dire di essere riusciti a costruire una libreria JADE in Scala, nonostante siamo consapevoli che possa essere ulteriormente migliorata. Il progetti ha richiesta più tempo del previsto ed energie, specialmente per superare il primo scoglio delle competenze di programmazione ad Agenti con la libreria Jade stessa. Superato esso, il successivo è stato riuscire ad ottenere delle interfacce il più possibile generalizzate per permettere future espansioni della libreria.

6.1 Lavori futuri

Non sono state neanche lontamente implementate tutte le funzionalità presenti nella libreria JADE originale, dato che molti esulano dalla programmazione ad agenti teorica vista a lezione. In future versioni della libreria, esse possono essere aggiunte alla libreria ed eventualmente migliorate, sia nell'usabilità che nelle prestazioni. Un esempio potrebbe essere la migrazione degli agenti e del loro flusso computazionale tra diversi Containers e diverse Piattaforme oppure l'implementazione dei metodo di sospensione ed attivazione dell'agente.

Attualmente i messaggi possono solamente venire scambiati sulla stessa Piattaforma e solamente all'interno della stessa macchina che esegue la Piattaforma centralizzata. Nella libreria Jade originale invece essi possono anche essere inviati sulla rete e scambiati anche tra hosts remoti. Una futura versione della libreria potrebbe anche supportare questa utilissima funzione.

Abbiamo accennato nel paragrafo relativo alla progettazione dei Containers, che non è possibile avviare un Container in un momento successivo alla generazione dello stesso. Un'ulteriore espansione della libreria potrebbe essere proprio permettere questa operazione, anche in diversi contesti: uno di essi potrebbe essere la ricezione di un certo messaggio da parte di un altro agente interno al container, se non dal Container stesso, su segnale dell'utente o dell'ambiente all'interno del quale è situato.

Implementando queste tre funzioni, sarebbe possibile avvicinarsi moltissimo all' attuale usabilità delle odierne librerie per lo sviluppo di sistemi distribuiti, come Akka, per la programmazione ad attori, ad oggi molto utilizzata in ambito Enterprise.

Come requisito non funzionale era stato posto quello di implementare API efficienti.

6.2 Che cosa abbiamo imparato

Durante lo sviluppo del progetto abbiamo sia raffinato i concetti appresi durante il corso, come l'utilizzo degli agenti e della loro interazione attraverso i messaggi. Abbiamo trascorso molto tempo a navigare la documentazione del progetto JADE e la Javadoc. Questo ci ha permesso di conoscere concetti non affrontati a lezione e che esulano dalla sola teorica programmazione ad agenti come affrontata a lezione. Essi non sono stati affrontati durante lo sviluppo della libreria, ma non c'è dubbio che possano essere affrontati successivamente in una espansione della libreria.

Abbiamo acquisito ulteriore dimestichezza con il linguaggio Scala, rispetto al solo insegnamento nel corso di "Paradigmi di programmazione e sviluppo", frequentato da tutti i partecipanti precedentemente a questo.

7 Istruzioni per il rilascio

7.1 Compilare il progetto

Scarica il repository git ufficiale al seguente *link* e compila le seguenti istruzioni:
`git clone https://gitlab.com/pika-lab/.../sd-project...`
`cd sd-project...`
`sbt reload`

Dopodichè pubblica tramite sbt il pacchetto della libreria che servirà in locale.
`sbt "project core" publishLocal // Core only.`

oppure
`sbt "project testkit" publishLocal // Testkit too.`

7.2 Aggiungere la libreria

Aggiungi le dipendenze corrette al tuo progetto, aggiungendo queste istruzioni al tuo file `build.sbt`:

```
resolvers += Resolver.mavenLocal
libraryDependencies += "it.unibo" && "saf-core" && "0.1.0"
```

// This line add Jade dependency to project from sbt.

```
lazy val root = project in file(".")
unmanagedJars in Compile += file(root.base + "/lib/jade-4.5.0.jar")
```

8 Esempi di utilizzo

8.1 Sintassi

In questo paragrafo viene mostrata la sintassi del DSL SAF attraverso un apposita tabella di esempi. Oltre alla sintassi, devono essere rispettare le seguenti regole, per questioni semantiche:

- L'esecuzione di un agente deve avvenire esclusivamente all'interno di un container che può ospitare più agenti;
- Per creare un agente è necessario definire le sue funzionalità nel seguente ordine:

  ```
  Agent(name) state() setup()
  ```

  Il nome del container e del agente può essere omesso: in questo caso la libreria creerà un identificativo univoco per identificare le relative istanze del container o agente creato. Anche il metodo `state()` per creare un agente privo di stato.

La tabella seguente mostra la sintassi delle principali funzionalità fornite. Più funzionalità possono essere utilizzate nello stesso agente per ottenere un comportamento più articolato. A sinistra della tabella, è riportata la semantica dell'istruzione mentre a destra, è mostrata la sintassi.

Azione / Istruzione	Istruzione DSL
Definizione di un **container** con un nome.	`Container(name)`
Definizione di un **agente**.	`Agent`
Definizione di un **agente con un nome proprio univoco** definito dall'utente.	`Agent(name)`
Definizione di **nuovi campi** che compongono lo **stato di un agente**.	`val counter : SafState[Int] = 10.state` `Agent state(counter)`
Definizione della funzione **setup**.	`Agent setup { statements ... }`

Azione / Istruzione	Istruzione DSL

Aggiungere un nuovo **behaviour**.	`val oneShot = Behaviours.oneShot {`
	`  println("Jade Agent")`
	`}`
	
	`Agent setup addBehaviour(oneShot)`

Eliminare un **behaviour**.	`val oneShot = Behaviours.oneShot {`
	`  println("Jade Agent")`
	`}`
	
	`Agent setup (`
	`  addBehaviour(oneShot)`
	`  removeBehaviour(oneShot)`
	`)`

Eliminare un agente.	`Agent setup doDelete()`

Inviare un messaggio a un agente specifico.	`val message = Message("Hello Agent B!",`
	`  Performative.INFORM)`
	
	`Agent("Agent A") setup (`
	`  send(message, Seq("Agent B"))`
	`)`

Inviare un messaggio a più agenti.	`val message = Message("Hello Agent B!",`
	`  Performative.INFORM)`
	
	`Agent("Agent A") setup (`
	`  send(message, Seq("Agent B", "Agent C"))`
	`)`

Ricezione di un messaggio (non-bloccante) da un altro agente.	`Agent("Agent A") setup receive()`

Ricezione di un messaggio (bloccante) da un altro agente.	`Agent("Agent A") setup (`
	`  addBehaviour(Behaviour.blockingReceive( _ =>`
	`  ()))`
	`)`

Azione / Istruzione	Istruzione DSL

Ricezione di un messaggio con uno specifico template da un altro agente.	`Agent("Agent A") setup (`
	`  addBehaviour(Behaviour.receive(() => _,`
	`  MessageTemplate.`
	`  MatchPerformative(Performative.REQUEST))`
	`)`

Eseguire uno specifico comportamento a seguito della ricezione di un messaggio.	`val message = Message("Hello Agent B!",`
	`  Performative.INFORM)`
	
	`Agent("Agent A") setup (`
	`  addBehaviour(Behaviour.blockingReceive { a =>`
	`  a.addBehaviour(Behaviour.send(message,`
	`  Seq("Agent B")))`
	`  })`
	`)`

In questa ultima tabella vengono forniti degli esempi ottenuti integrando varie funzionalità viste nelle precedenti tabelle:

Azione / Istruzione	Istruzione DSL

Agente che **invia un messaggio e attende la risposta**.	`val message = Message("Hello Agent B!",`
	`  Performative.INFORM)`
	
	`Agent("Agent A") setup (`
	`  addBehaviour(Behaviour.sequential(`
	`  List(Behaviour.send(`
	`  message, Seq("Agent B")),`
	
	`  Behaviour.receive(() => _))`
	`  )`
	`  ))`
	`)`

Azione / Istruzione	Istruzione DSL

Agente che **gestisce la comunicazione in base al suo stato**.	`val msgToSend: SafState[Int] = 2.state`
	
	`val message = Message("Hello Agent B!", Performative.INFORM)`
	
	`Agent("Agent A") state msgToSend setup addBehaviour(Behaviour.cyclic { a: Agent with Created =>`
	
	`if (a.internal.get("msgToSend").get != 0) {`
	
	`val message = Message( s"Hello Agent! My state is ${a.internal.get("msgToSend").get}", Performative.INFORM )`
	
	`val state: SafState[Int] = a.internal.get("msgToSend")`
	
	`a.addBehaviour(Behaviour.send(message, Seq("Agent B")))`
	
	`state.set(state.get - 1) } })`