5 Framework para Desenvolvimento de Agentes Stubs Desenvolvidos com M-Law

Assim como os trabalhos relacionados previamente apresentados no capítulo 3, o objetivo dessa dissertação é permitir o desenvolvimento de sistemas multiagentes mais confiáveis. Em especial, esse trabalho destina-se ao desenvolvimento de agentes para um middleware específico chamado M-Law que oferece suporte para linguagem XMLaw, ambos brevemente apresentados no capítulo 2.

O Framework aqui proposto destina-se a resolver o problema de criação de agentes stubs para sistemas multiagentes abertos desenvolvidos com o middleware M-Law. A coordenação da execução também pode ser alcançada facilmente com a devida instanciação do framework.

PUC-Rio - Certificação Digital Nº 0420988/CA

5.1 Agentes Genéricos

A motivação dos agentes genéricos é facilitar a forma como agentes stubs são criados e executados. Considerando que o número de execuções de agentes stubs para a simulação de um cenário pode crescer em ordem quadrática e que a criação de um único agente stub requer a criação de uma classe própria e, eventualmente, uma outra responsável por comparar comportamento esperado do obtido e controlar a execução dos mesmos, fica bastante evidente a necessidade de se facilitar a criação de agentes stubs, assim como a verificação do comportamento obtido da execução de um agente real.

Um agente genérico é, portanto, uma solução para o problema do desenvolvimento de agentes stubs. Seu objetivo é fornecer ao desenvolvedor uma maneira simples e flexível de representar o comportamento de um agente. O framework aqui apresentado fornece uma solução para a manipulação de agentes genéricos.

Um agente genérico é um agente XMLaw, isto é, ele estende a classe *Agent* oferecida pelo middleware M-Law. No entanto, diferente da implementação de um agente real na qual recomenda-se o uso de herança para a criação de novos agentes [7], aos agentes genéricos recomenda-se o uso direto da classe, isto é, a simples criação de uma instância já é suficiente.

Caso o desenvolvedor necessite de funcionalidades extras, o agente genérico

pode ser estendido e modificado, sobrescrevendo-se os comportamentos existentes ou criando novos.

5.2 Arquitetura da Solução

O framework para desenvolvimento de agentes stubs oferece em seus pontos fixos funcionalidades para a manipulação de agentes genéricos. Na figura 5.1 podemos ver a arquiteruta da solução, desenvolvida a partir do padrão MVC.

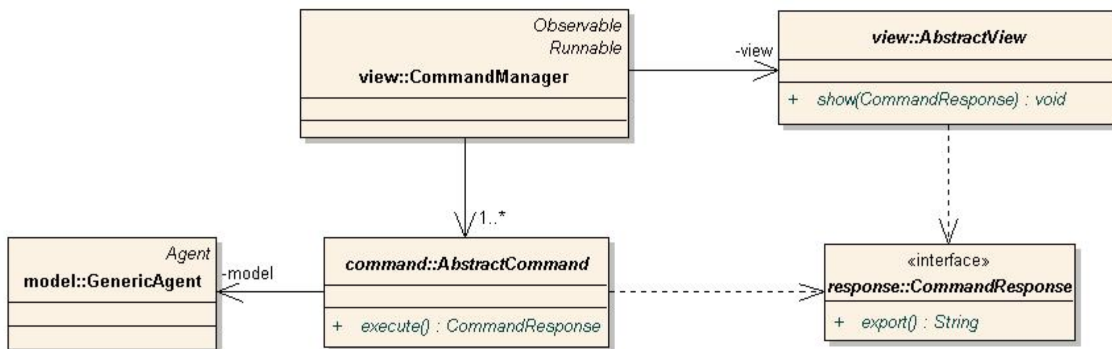

Figura 5.1: Diagrama Estrutural para Agentes Genéricos

Nessa arquitetura, o modelo é representado por uma instância de agente que somente deve ser acessado pelas classes do tipo *AbstractCommand*. Classes do tipo comando são responsáveis por manipular o modelo, gerando resultados que podem ser exibidos pela camada de visualização, aqui representada pela classe *AbstractView*. A classe *CommandManager* é a responsável por receber as requisições, executar os comandos e delegar para a camada de visualização o resultado da execução.

Os comandos não são executados diretamente por chamadas de métodos, mas sim por requisições escritas no canal de entrada do *CommandManager*. Dessa forma, as requisições dizem quais comandos devem ser ativados.

5.2.1 Estrutura de Comandos

A estrutura de comandos é baseada no padrão *Command* [12]. Cada operação a ser desempenhada sobre um agente genérico é escrita como uma classe do tipo *AbstractCommand*. Cada comando possui um nome, descrição e fornece um pequeno texto explicando como deve ser utilizado.

Ao serem executados alguns comandos requerem argumentos. Por exemplo, o comando para entrada de organização requer a identificação de execução da organização, enquanto que o comando de ajuda pode ser executado sem argumento algum.

PUC-Rio - Certificação Digital Nº 0420988/CA

PUC-Rio - Certificação Digital Nº 0420988/CA

Internamente, a classe *CommandManager* é responsável por escolher o comando a ser executado e informar os argumentos de sua execução, de acordo com a requisição recebida. Além disso, os comandos também possuem um esquema para validação dos argumentos informados. Dessa forma, eles não devem ser executados caso o valor de seus argumentos não sejam apropriados.

A modelagem dos argumentos é feita pela classe *Argument* como podemos ver na figura 5.2.

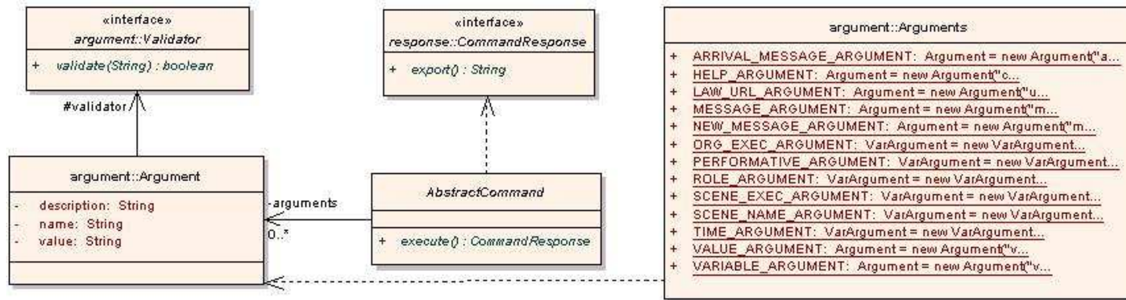

Figura 5.2: Diagrama de Classes para o esquema de argumentos

Cada instância da classe *Argument* representa um tipo de argumento, enquanto que *Arguments* provê todas as instâncias necessárias à aplicação. Os argumentos possuem, além de nome e descrição, validadores, representados pela interface *Validator*, que podem informar se o valor neles configurado é valido ou não. Assim, um argumento criado para representar a identificação de execução de uma organização só pode admitir valores de tipos inteiros e, por isso, tal argumento possui um *NumberValidator*. Caso seja passado um inteiro que não represente a execução de uma organização, o mesmo será tratado do lado do servidor. A validação para a execução de um comando é, portanto, a verificação de que todos os seus argumentos possuem valores com a sintaxe correta.

O resultado da execução de um comando é sempre uma instância de *CommandResponse*. Após a execução o resultado é enviado a camada de visualização, que é a responsável por exibir a resposta, ou seja, escrevê-la no canal de saída. Existem diversos tipos de resposta, cada uma representa um resultado de uma execução diferente. No entanto, todos devem respeitar a interface *CommandResponse*.

Como citado anteriormente, a classe *CommandManager* possui um canal de entrada por onde as requisições são recebidas como seqüências de caracteres. A estrutura que implementa os canais será melhor explicado na seção 5.2.3. Ao receber uma requisição espera-se que a primeira palavra seja o nome do comando, as palavras seguintes sejam os argumentos e a seqüência deva acabar com um caractere de fim de linha, indicando o fim da requisição.

Caso um usuário não esteja satisfeito com os comandos disponíveis para manipulação do agente genérico, existe a possibilidade de extensão. Para isso, basta

que a arquitetura seja respeitada e novos comandos, do tipo *AbstractCommand*, sejam criados e incluídos no *CommandManager*. Um cenário que justifica a extensão é a criação de agentes restritos a um único domínio ou aplicação. Por exemplo, suponha um sistemas de leis que controla um mercado de ações. Uma extensão poderia ser feita considerando comandos que enviam mensagens que consultam a cotação de determinada ação. Dessa forma, o esforço de execução de diversos comandos para a criação, configuração, envio e recebimento de mensagens seriam reduzidos. Outro caso poderia ser a análise de compra, que pode envolver algum algoritmo que se deseja automatizar.

5.2.2
Suporte à Variáveis

A execução de um agente seria muito restrita se fosse limitada à execução de comandos sobre um modelo. A atual versão para a criação de agentes genéricos permite a criação e edição de variáveis durante o tempo de execução. Assim os comandos especificados pelos desenvolvedores podem ser ainda mais flexibilizados.

O suporte à variáveis é um ambiente que pode ser compartilhado por diversos agentes genéricos. Desse modo, uma variável declarada por um agente pode ser aproveitada por outro. O uso de variáveis globais pode ser perigoso caso o número de declarações seja muito grande. Para contornar o problema sugerimos que o desenvolvimento siga algum padrão para nomeação de variáveis.

Assim como as requisições enviadas aos comandos seguem um formato de *String*, as variáveis também são mapeadas para o ambiente seguindo um formato de *String* para os valores assumidos e para o nome de declaração. A manipulação das variáveis é feita a partir de comandos, possibilitando que o desenvolvedor dispare requisições para a criação e edição de variáveis.

O ambiente não é restrito apenas para variáveis que podem ser declaradas e editadas. Os comandos, ao executarem, podem receber variáveis e considerarem o valor nelas contido como argumento. Outro ponto interessante é a manipulação de mensagem. Para o ambiente, uma mensagem nada mais é do que um conjunto de variáveis. Por exemplo, quando a mensagem bar_msg é criada existe um mapeamento para o ambiente, onde serão criadas variáveis para os campos da mensagem. Assim temos:

– bar_msg - a representação textual da mensagem
– bar_msg.sender - o agente que enviará/enviou a mensagem
– bar_msg.senderRole - o papel do agente que enviará/enviou a mensagem
– bar_msg.receiver - o agente destinatário
– bar_msg.receiverRole - o papel do agente destinatário

PUC-Rio - Certificação Digital Nº 0420988/CA

- bar_msg.protocol - indica se mensagem é destinada ao mediador ou a outro agente
- bar_msg.id - identificador da mensagem
- bar_msg.preformative - performativa da mensagem

Caso seja necessário incluir outros campos na mensagem, o desenvolvedor pode configurar, por exemplo, a variável bar_msg.info para incluir o conteúdo info dentro da mensagem. Quando o comando de envio de mensagem for executado, os valores das variáveis referentes à mensagem de argumento serão recuperados e incluídos dentro de um objeto mensagem para envio.

5.2.3 Estrutura de Canais

Quando falamos em flexibilidade no contexto do framework aqui apresentado, estamos nos referindo à forma como é feita a entrada e saída para as requisições e respostas. Sabemos que os agentes genéricos recebem requisições que estão mapeadas em comandos que executam operações no agente. Essa seção apresentará a forma utilizada para se encaminhar uma requisição até o agente e como descobrir o resultado das execuções.

Segundo as seções anteriores, a gerência dos agentes é feita pela classe *CommandManager*. A mesma classe é responsável por encaminhar as requisições que chegam através de canais. Um canal é modelado como uma *Stream* java [30].

Ao utilizarmos java, o processo de entrada e saída é sempre baseado em *Streams*. Existem, portanto, duas interfaces distintas, uma para entrada e outra para saída. São elas: *InputStream* e *OutputStream*, respectivamente. Diversas implementações são fornecidas de acordo com o tipo de meio utilizado: sistema de arquivos, conexão de rede, linha de comando. O grande benefício adquirido pela estrutura de canais é que a aplicação desenvolvida fica restrita somente a interface, podendo ser aplicada com qualquer tipo de implementação.

O desenvolvimento do framework tirou proveito dessa abordagem. Conforme ilustrado na figura 5.3 podemos ver que são utilizados três canais distintos: entrada, saída e controle.

A entrada é o canal por onde serão lidas as requisições. Seu formato deve estar de acordo com aquele descrito na seção 5.2.1, separados por caracteres de “nova linha”. O canal de saída é o local onde serão escritas todas as respostas dos comandos. O canal de controle serve para o controle automatizado. Em especial, existe um comando assert que compara se dois valores são iguais. Quando executado, grava 0(zero) no canal de controle caso a igualdade seja verificada, ou 1(um) se os valores são distintos. Essa funcionalidade permite a construção de programas que verifiquem se os valores escritos no canal são os esperados.

PUC-Rio - Certificação Digital Nº 0420988/CA

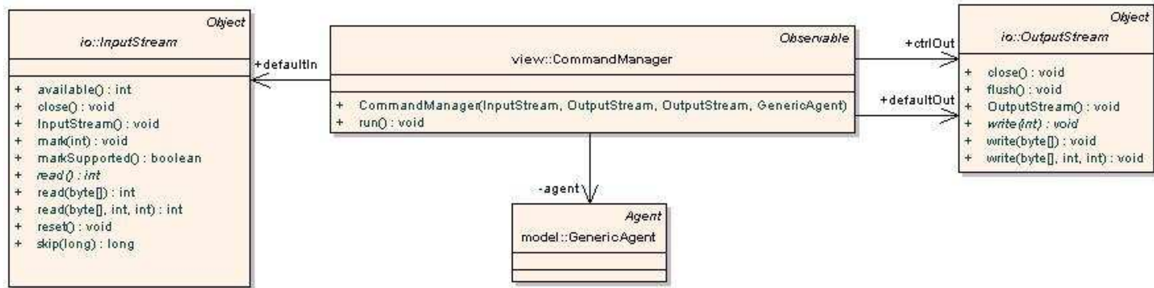

Figura 5.3: Esquema de canais para os agentes genéricos

Abaixo, na listagem 5.1 temos um exemplo de instanciação para um agente genérico:

```
1 ...
2 GenericAgent agent = new GenericAgent("command␣line␣agent");
3 CommandManager manager = new CommandManager(System.in, System.out, new
      StubOutputStream(), agent);
4 ...
```

Listagem 5.1: Exemplo de instanciação de agentes genéricos para linha de comando

O construtor, para o *CommandManager*, recebe os três canais na seguinte ordem: canal de entrada, canal de saída e canal de controle. No exemplo da listagem 5.1 é gerada uma instância que recebe os comandos pela linha de comando do console, onde a aplicação estiver executando. O resultado é escrito na própria tela do console. O canal de controle não é utilizado. Por isso, recebe uma instância de canal Stub, isto é, a escrita feita não é considerada. Essa configuração permite a interação direta do desenvolvedor com o sistemas de leis. Assim, ele pode escrever os comandos desejados na própria tela de execução e observar os resultados obtidos. Essa é uma abordagem que permite ao usuário entender melhor as leis que regem o sistema. No entanto, a repetição dos comandos escritos é algo trabalhoso. Para que a repetição possa ser feita de forma eficiente é melhor utilizarmos outro tipo de canal para entrada, como podemos ver na listagem 5.2.

```
1 ...
2 FileInputStream scriptStream = new FileInputStream("script.txt");
3 FileOutputStream outputStream = new FileOutputStream("output.txt");
4 GenericAgent agent = new GenericAgent("script␣agent");
5 CommandManager manager = new CommandManager(scriptStream, outputStream, new
      StubOutputStream(), agent);
6 ...
```

Listagem 5.2: Exemplo de instanciação de agentes genéricos usando arquivos

Nesse exemplo, as requisições são lidas a partir de um arquivo de script. Elas devem possuir o mesmo formato do exemplo anterior. A saída é feita também em um arquivo. Portanto, durante a execução, os resultados dos comandos são escritos no arquivo output.txt. Assim como no exemplo anterior, as assertivas não estão sendo

PUC-Rio - Certificação Digital Nº 0420988/CA

consideradas. Essa configuração permite ao desenvolvedor fazer diversas execuções de forma eficiente já que as requisições estão armazenadas no arquivo de script. A verificação da execução pode ser conferida posteriormente no arquivo output.txt. O exemplo da listagem 5.3 demonstra como podemos automatizar o teste de um agente genérico.

```
1  ...
2  FileInputStream scriptStream = new FileInputStream("script.txt");
3  PipedInputStream pipeInScript = new PipedInputStream();
4  PipedOutputStream pipeOutScript = new PipedOutputStream(pipeInScript);
5
6  PipedInputStream pipeInCtrl = new PipedInputStream();
7  PipedOutputStream pipeOutCtrl = new PipedOutputStream(pipeInCtrl);
8
9  TestApp app = new TestApp(scriptStream, pipeOutScript, pipeInCtrl);
10
11 FileOutputStream outputStream = new FileOutputStream("output.txt");
12 GenericAgent agent = new GenericAgent("tested agent");
13 CommandManager manager = new CommandManager(pipeInScript, outputStream,
       pipeOutCtrl, agent);
14 ...
```

Listagem 5.3: Exemplo de instanciação de agentes genéricos usando teste automatizado

Para o exemplo de teste automatizado, uma nova aplicação é inserida na execução: TestApp. Podemos entender esse elemento como algo que lê o script de teste e repassa para o agente genérico. O inverso acontece com o canal de controle, o agente genérico escreve e a aplicação lê. A comunicação entre a aplicação de testes e o agente genérico é feita através de *Pipes*. Um pipe é na verdade uma implementação de uma *Stream* que está sempre casada com um outro par. Tudo o que é escrito no pipe de entrada sai pelo pipe de saída. Por isso o nome pipe (cano em inglês). Assim, a aplicação de testes recebe um script e também, recebe um pipe de entrada ligado a um pipe de saída que é repassado ao framework. O mesmo princípio é aplicado com o canal de controle, um pipe é utilizado com sua entrada voltada para o framework e a saída para a aplicação de teste.

A automação do teste pode ser implementada dentro da aplicação de teste. Quando um comando de assertiva é lido, a aplicação pode saber o resultado da execução e esperá-lo na saída de controle. Se algum distúrbio for encontrado, ou seja, o valor 1(um) for lido do canal de controle, a aplicação pode implementar meios de notificar o desenvolvedor.

5.3 Utilização do Framework

A seção anterior apresentou como está organizada estruturalmente a implementação do Framework. Nesta seção, apresentaremos como devemos utilizar o

PUC-Rio - Certificação Digital Nº 0420988/CA

Framework, ou seja, quais são os comandos e argumentos já implementados. Algumas peculiaridades quanto ao comportamento dos comandos também serão apresentadas.

5.3.1 Argumentos e Comandos

Existem, no total, treze argumentos diferentes implementados. Cada um deles possui uma definição e valores que são configuráveis. Um mesmo tipo de argumento pode ser utilizado em diversos comandos diferentes. Assim, o conhecimento dos argumentos existentes é importante para que o desenvolvedor saiba como argumentar corretamente a execução dos comandos. Os argumentos que possuem algum tipo de validação possuem em sua definição os tipos de valores que lhes podem ser atribuídos.

PUC-Rio - Certificação Digital Nº 0420988/CA

Nome Argumento	Descrição	Valores Aceitos
orgExecution	Identificador de uma organização em execução	números naturais
sceneExecution	Identificador de uma cena em execução	números naturais
scene	O nome de uma cena	qualquer valor
role	O papel a ser desempenhado	qualquer valor
command	O nome de um comando existente	qualquer valor
time	O tempo de espera pela chegada de uma mensagem	números naturais
newMessage	O nome de uma nova mensagem a ser criada	qualquer valor
messageName	Uma mensagem que foi criada no ambiente	apenas o nome de variáveis criadas no ambiente
arrivalMessage	O nome a ser atribuído a uma mensagem recebida	qualquer valor
var	O nome de uma variável	qualquer valor
value	Um valor ou conteúdo de uma variável	Uma (String) que não comece por $ ou uma que comece por $, que mas represente uma variável no ambiente
url	Uma url que aponte para um arquivo de lei	qualquer valor

A seguir estão listados todos os treze comandos implementados. O formato da requisição é o mesmo para todos. No entanto, o nome do comando, tipo e quantidade

PUC-Rio - Certificação Digital Nº 0420988/CA

PUC-Rio - Certificação Digital Nº 0420988/CA

dos argumentos difere para cada requisição.

Add Law Command	
Forma da Requisição	addLaw ulr
Descrição	Procura a url fornecida como argumento e repassa para o servidor. O servidor tenta publicar a lei. Caso não seja possível, uma mensagem de erro será retornada. Em caso de sucesso, será informado o número de execução para a nova organização publicada

Enter Organization Command	
Forma da Requisição	enterOrg orgExecution
Descrição	Pede ao servidor para incluir esse agente na organização que está executando com o id fornecido como argumento

Start Scene Command	
Forma da Requisição	startScene orgExecution scene
Descrição	Pede ao servidor para criar a scene fornecida como argumento dentro da organização, cuja identificação também é fornecida

Enter Scene Command	
Forma da Requisição	enterScene orgExecution sceneExecution
Descrição	Pede ao servidor para incluir esse agente na cena que está executando com o identificador fornecido em sceneExecution dentro da organização representada por orgExecution

Perform Role Command	
Forma da Requisição	performRole orgExecution role
Descrição	Pede ao servidor para que o agente em questão desempenhe o papel indicado em role dentro da organização fornecida em orgExecution

Set Command	
Forma da Requisição	set var value
Descrição	Cria a variável (ou modifica o valor caso exista) de nome var colocando como valor value, que pode ser um valor qualquer ou o conteúdo de uma variável

Create Message Command	
Forma da Requisição	msg newMessage performative
Descrição	Cria um conjunto de variáveis que representa uma mensagem com nome fornecido em newMessage. O conteúdo da variável newMessage.performative será igual ao fornecido em performative

Send Message Command	
Forma da Requisição	send messageName
Descrição	Resgata os valores inseridos no ambiente de variáveis pelo comando Create Message e cria uma nova mensagem que é enviada em seguida. MessageName indica o nome do conjunto, ou nome da mensagem, criado pelo comando Create Message

Receive Message Command	
Forma da Requisição	receive time arrivalMessage
Descrição	Espera pela chegada de uma mensagem por, no máximo, time segundos. Caso uma mensagem chegue, um conjunto de variáveis com o nome arrivalMessage será criado no ambiente de variáveis. Caso o argumento time seja igual a zero, o comando é bloqueante até a chegada de uma mensagem

Help Command	
Forma da Requisição	help command
Descrição	Procura pelo comando fornecido em command e exibe no canal de saída instruções de como ele funciona
Forma da Requisição	help
Descrição	Exibe no canal de saída as instruções de todos os comandos existentes

PUC-Rio - Certificação Digital Nº 0420988/CA

Reply Message Command	
Forma da Requisição	reply messageName newMessage performative
Descrição	Cria uma mensagem de resposta para a mensagem representada pelo conjunto de variáveis começados pelo nome messageName. O nome da nova mensagem será newMessage e sua performativa será igual ao valor do argumento performative. Como uma mensagem pode ter diversos destinatários, a resposta da mensagem é implementada colocando-se o agente em questão como remetente e o antigo remetente é adicionado a lista de destinatários

Assert Command	
Forma da Requisição	assert value value
Descrição	Verifica se o valor fornecido pelos argumentos value são iguais. O valor pode ser o conteúdo de uma variável ou uma string com valor esperado. O comando escreve no canal de controle o valor 0(zero) caso os valores sejam iguais. Quando eles são diferentes, o comando escreve 1(um)

Disconnect Command	
Forma da Requisição	disconnect
Descrição	Desconecta o agent em questão do sistema aberto

PUC-Rio - Certificação Digital Nº 0420988/CA

5.3.2 A Variável “last”

Um recurso interessante do suporte à variáveis é a variável *last*, criada por todos os comandos que recebem algum tipo de mensagem do servidor: addLaw, enterOrg, startScene, enterScene e performRole.

O objetivo é permitir que o desenvolvedor possa resgatar o conteúdo de mensagens enviadas pelo servidor. Assim, quando qualquer um dos comandos citados é executado, um conjunto de variáveis iniciados por “last” armazena o conteúdo da mensagem recebida pelos comandos.

Ao executar um script, um desenvolvedor pode optar por guardar o valor do identificador de execução de uma organização para usar em outros comandos.

Essa abordagem torna o script mais flexível, pois o desenvolvedor não precisa saber exatamente o número da organização publicada. A listagem 5.4 exemplifica o uso da variável last:

```
1  ...
2  addLaw http://www.les.inf.puc-rio.br/laws/exampleLaw.xml
3  set orgId $last.orgExecutionId
4  ...
5  performRole $orgId role
6  ...
7  startScene $orgId scene
8  ...
9  enterScene $orgId $sceneId
10 ...
11 set $message.orgExecutionId $orgId
12 ...
```

Listagem 5.4: Exemplo de uso da variável last.

No exemplo, podemos observar que a presença da variável "last" possibilitou a recuperação do identificador de execução da organização publicada. A recuperação da variável $orgId é necessária em diversos trechos do script: desempenho de papel, criação de cena, entrada de cena e configuração de mensagem.

5.4 Validação do Framework

Para que seja possível garantir o correto funcionamento das aplicações desenvolvidas pelo framework, foram feitos diversas validações referentes aos componentes desenvolvidos. Os testes foram elaborados utilizando-se JUnit [33].

A classe *GenericAgentTestCase* se encarrega de simular um ambiente para que os testes possam ser executados. Sua função é iniciar localmente um agente mediador e publicar uma lei de teste que sera utilizada para os demais testes.

Seguindo esse modelo, as classes referentes aos testes de unidade devem ser estendidas de *GenericAgentTestCase* e assim implementar o teste desejado. Por exemplo, suponha o comando *PerformRole* descrito anteriormente. Sua função é informar ao agente mediador que determinado agente agora desempenha um papel no sistema aberto. Além dessa funcionalidade o comando possui um nome para ser invocado e uma quantidade determinada de argumentos com determinado tipo cada um. Para garantirmos que o comando sera corretamente executado quando invocado devemos garantir que essas funcionalidades estejam também bem definidas. Para o nosso exemplo, portanto, teremos uma classe chamada *PerformRoleCommandTestCase* (pois o comando encontra-se implementado na classe *PerformRoleCommand*). Essa classe de teste possui três métodos: *testArgs*, *testName* e *testExecute*. Os dois primeiros verificam so o nome do comando esta consistente com o a listagem de comandos existentes e se os argumentos esperados são os mesmos descritos em sua

PUC-Rio - Certificação Digital Nº 0420988/CA

definição. O terceiro método é referente a funcionalidade do comando conforme demonstrado na listagem 5.5.

```
1  public void testExecute() {
2      assertTrue(command.areArgsOk(new StringTokenizer(orgExecution + "␣ping")));
3
4      CommandResponse response = command.execute();
5      assertNotNull(response);
6
7      Message reply = ((MessageCommandResponse) response).getMessage();
8      assertNotNull(reply);
9
10     assertEquals(MessageContentConstants.CMD_PERFORM_ROLE, reply
11                 .getContentValue(MessageContentConstants.KEY_COMMAND));
12
13     Message last = enviromentVariables.restoreMessage(EnviromentVariables.LAST_MSG
           );
14     assertEquals(MessageContentConstants.CMD_PERFORM_ROLE, last
15                 .getContentValue(MessageContentConstants.KEY_COMMAND));
16 }
```

Listagem 5.5: Exemplo de teste automatizado para o comando PerformRole

Podemos ver que o teste configura o comando com informações referentes ao papel ping na organização criada anteriormente pelo método *setup* do junit. O comando é então executado e o teste verifica se o agente mediador retornou uma mensagem indicando que um papel foi atribuído ao agente e se a variável ”last“ foi configurada corretamente no ambiente. Note que nesse teste não é preciso testar a argumentação errada do comando, como por exemplo, fornecer um papel inválido. Isso devido ao fato de responsabilidade de atribuição do papel ser atribuída ao agente mediador e, portanto, esse teste foi desenvolvido com o middleware M-Law.

Um total de 38 classes de teste foram desenvolvidas. Nem todas estendem *GenericAgentTestCase*, somente quando é necessário fazer alguma simulação do ambiente.

5.5 Guia para o Uso do Framework

Nesta seção será apresentada o uso da instância mais simples que o framework pode oferecer. Como foi exemplificado anteriormente, podemos ter uma instância quando fornecemos a linha de comando do console como canal de entrada e o próprio console como saída.

O objetivo é poder exemplificar como cada comando funciona em uma situação bem simples. Será utilizada a lei ping-pong que representa uma interação muito simples entre dois agentes. Nesse caso serão utilizados duas instâncias de agentes genéricos.

PUC-Rio - Certificação Digital Nº 0420988/CA

PUC-Rio - Certificação Digital Nº 0420988/CA

5.5.1 A Lei Ping-Pong

O objetivo dessa lei é controlar a interação entre dois agentes: ping e pong. A implementação da cena game fornece um protocolo onde duas mensagens são trocadas. Primeiramente, o agente ping deve iniciar a interação enviando uma mensagem com o conteúdo ping ao agente pong. Ao receber a mensagem, o agente pong responde com uma mensagem de conteúdo ping-pong e assim o protocolo da cena é encerrado.

A listagem 5.6 contém o conteúdo da lei no formato XMLaw.

```
1
2  <Laws xmlns:xi="http://www.w3.org/2001/XInclude">
3      <LawOrganization id="ping-pong" name="International␣Ping-Pong␣Organization">
4          <!-- Role definition -->
5          <Role id="ping"/>
6          <Role id="pong"/>
7          <Scene id="game" time-to-live="infinity">
8              <Protocol id="ping-pong-protocol">
9                  <Messages>
10                     <Message id="m1" performative="inform">
11                         <Content>
12                             <Entry key="info" value="ping"/>
13                         </Content>
14                         <Sender role-ref="ping" role-instance="$theSender"/>
15                         <Receivers>
16                             <Receiver role-ref="pong" role-instance="$receiver1"/>
17                         </Receivers>
18                     </Message>
19                     <Message id="m2" performative="inform">
20                         <Content>
21                             <Entry key="info" value="ping-pong"/>
22                         </Content>
23                         <Sender role-ref="pong" role-instance="$theSender"/>
24                         <Receivers>
25                             <Receiver role-ref="ping" role-instance="$receiver1"/>
26                         </Receivers>
27                     </Message>
28                 </Messages>
29                 <States>
30                     <State id="s0" type="initial" label="Initial␣State"/>
31                     <State id="s1" type="execution" label="Ping␣sent"/>
32                     <State id="s2" type="success" label="Pong␣sent"/>
33                 </States>
34                 <Transitions>
35                     <Transition id="t1" from="s0" to="s1" ref="m1" event-type="
                           message_arrival"/>
36                     <Transition id="t2" from="s1" to="s2" ref="m2" event-type="
                           message_arrival"/>
37                 </Transitions>
38             </Protocol>
39             <!-- Creators Roles -->
40             <!-- Papeis que podem criar a cena, ao criar a cena o agente nao entra
                   nela -->
41             <Creators>
42                 <Creator role_ref="ping"/>
```

```
43              </Creators>
44              <!-- Entrance Roles and States -->
45              <!-- Papeis que podem entrar na cena, em determinado estado -->
46              <Entrance>
47                  <Participant role_ref="ping" limit="1">
48                      <State ref="s0"/>
49                  </Participant>
50                  <Participant role_ref="pong" limit="1">
51                      <State ref="s0"/>
52                      <State ref="s1"/>
53                  </Participant>
54              </Entrance>
55          </Scene>
56      </LawOrganization>
57  </Laws>
```

Listagem 5.6: Lei Ping-Pong.

PUC-Rio - Certificação Digital Nº 0420988/CA

5.5.2 Utilizando os Agentes

A distribuição do framework para agentes genéricos possui alguns exemplos com algumas instâncias implementadas. O exemplo que vamos demonstrar aqui já está incluído e pode ser executado pelo arquivo agent.bat. Essa execução espera receber como argumento o nome do agente a ser criado, por exemplo: agent.bat ping vai criar o agente ping e conectá-lo ao mediador configurado em seu arquivo de configurações.

Como a lei ping-pong diz respeito a dois agentes teremos que executar duas instâncias distintas da aplicação, uma para o agente ping e outra para o agente pong. Portanto, dois consoles distintos devem ser utilizados, um iniciado com a linha de comando “agent.bat ping” e outro com “agent.bat pong”.

Agora podemos começar a escrever comandos no canal de entrada de ambos agentes para simular uma interação. Primeiramente, podemos pedir que ping publique a lei no mediador. Para isso o comando addLaw é utilizado passando-se a url da lei ping-pong que se deseja publicar. Usualmente o arquivo da lei pode ser encontrado em qualquer url. Esta aqui utilizada é apenas um exemplo. Na prática o arquivo está localizado no disco rígido do computador e uma url com cabeçalho file:/// ficaria muito extensa para o exemplo.

```
1  addLaw http://www.les.inf.puc-rio.br/laws/ping-pong-law.xml
2  INFORM the organization ping-pong was published in the open system with execution
       id 211
```

Listagem 5.7: Publicando Lei Ping-Pong.

Após a execução do comando para a publicação da lei, o agente mediador do middleware M-Law envia uma mensagem indicando que a organização foi publicada com o id 211. Com a lei publicada, podemos pedir ao agente ping que

entre na organização ping-pong. Para isso precisamos da informação do seu id de execução: 211.

```
2  ...
3  enterOrg 211
4  INFORM The agent succefully entered into the organization ping-pong [211]
```

Listagem 5.8: Entrando na Organização Ping-Pong.

O próximo passo é pedir que o agente ping desempenhe o papel ping na organização. Para que isso aconteça é utilizado o comando performRole com os argumentos referentes à organização e papel.

```
4  ...
5  performRole 211 ping
6  INFORM The agent performs now the role ping
```

Listagem 5.9: Entrando na Organização Ping-Pong.

Uma vez desempenhando seu papel, o agente ping, segundo descrito na lei, pode criar a cena "game" e entrar na mesma. Note que ao criar uma cena o agente não necessariamente entra nela. Por esse motivo existem dois comandos distintos: startScene e enterScene. O primeiro especifica qual cena será criada e em qual organização. O segundo em qual cena de qual organização e qual será o papel desempenhado pelo agente na cena. Para desempenhar o papel em uma cena é necessário que o agente o desempenhe na organização.

```
6  ...
7  startScene 211 game
8  INFORM The agent started the scene game[215] into the organization ping-pong[211]
9  enterScene 211 215 ping
10 INFORM The agent succefully entered into the scene game [215] under the role ping
```

Listagem 5.10: Criando e Entrando na Cena Game.

Observamos, portanto, que o comando startScene criou uma cena chamada game com um identificador de execução igual a 215. Os identificadores de organização e cena são utilizados para entrada de cena com o comando enterScene.

Podemos agora enviar uma mensagem para o agente pong para convidá-lo a entrar na organização e cena. A criação de uma mensagem é feita utilizando-se o comando msg, passando-se como argumento a performativa que esta mensagem vai possuir. Ao criar uma mensagem, seu conteúdo é transformado em variáveis que podem ser configuradas pelo comando set. Este comando recebe como argumentos o nome da variável e o valor a ser armazenado nela. O comando send recupera todas as variáveis que representam uma mensagem e cria efetivamente uma mensagem XMLaw que é enviada logo em seguida.

```
10 ...
11 msg toPong inform
12 INFORM toPong
```

PUC-Rio - Certificação Digital Nº 0420988/CA

```
13 set toPong.info ping
14 toPong.info set to ping
15 set toPong.orgExecutionId 211
16 toPong.orgExecutionId set to 211
17 set toPong.sceneExecutionId 215
18 toPong.sceneExecutionId set to 215
19 set toPong.receiver1 pong
20 toPong.receiver1 set to pong
21 set toPong.receiverRole1 pong
22 toPong.receiverRole1 set to pong
23 set toPong.senderRole ping
24 toPong.senderRole set to ping
25 send $toPong
26 INFORM toPong
```

Listagem 5.11: Configurando e Enviando uma Mensagem.

Segundo a listagem, a mensagem criada sob o nome toPong possui a performativa inform. O primeiro campo configurado foi o campo “info”, que segundo a especificação da lei deve conter o valor “ping”. Para configurar esse campo usou-se o comando set que armazenou o valor “ping” dentro da variável toPong.ping. Como todos os outros comandos, ao ser executado o comando set retorna uma linha indicando que a variável em questão foi configurada com o valor argumentado. Outras configurações necessárias para o envio da mensagem foram configuradas da mesma maneira. O campo orgExecutionId recebeu o identificador da execução da organização assim como o campo sceneExecutionId recebeu o identificador da cena. Esses campos são necessários para se indicar em qual execução de organização e qual execução de cena a mensagem será validada. Os campos receiver1 e receiverRole1 indicam o agente destinatário e qual é seu papel desempenhado. Como uma mensagem pode possuir diversos destinatários, a configuração aos campos receiver2 e receiverRole2 indicariam um segundo receptor. Finalmente o campo senderRole é configurado, indicando o papel do remetente.

O comando send é argumentado com o conjunto de variáveis que possui a mensagem para ser criada e enviada. Quando executa-se send $toPong, como vemos na listagem, estamos utilizando todos os valores configurados pelo comando set previamente.

Com a mensagem enviada para pong, podemos pedir ao agente pong que receba a mensagem e a armazene em um conjunto de variáveis. Para isso podemos utilizar o comando receive, que recebe como argumentos o tempo, em segundos, que o agente deve esperar até receber a mensagem e o nome da mensagem recebida. Caso 0 segundos seja passado como parâmetro, o comando irá bloquear a execução do agente até uma nova mensagem chegar.

```
26 ...
27 receive 5 fromPing
28 INFORM toPong
```

PUC-Rio - Certificação Digital Nº 0420988/CA

Listagem 5.12: Recebendo Mensagem.

Na listagem, o agente pong espera por 5 segundos até receber uma mensagem. Ao recebê-la cria um conjunto de variáveis que começam com fromPing. A resposta da execução do comando indica que a mensagem toPong, com performativa inform, foi recebida. Para simplificar os comandos do agente pong podemos criar variáveis que armazenem o identificador da organização e cena que está interagindo, pode-se usar o comando set argumentando o valor de uma variável com 211 para organização e 215 para cena. No entanto, como se trata de um exemplo explicativo, podemos recuperar esses valores da variável last, que armazena a última mensagem recebida.

```
28 ...
29 set orgId $last.orgExecutionId
30 orgId set to 211
31 set sceneId $last.sceneExecutionId
32 sceneId set to 215
```

Listagem 5.13: Recebendo Mensagem.

Dessa forma é possível sempre utilizar as variáveis orgId e sceneId quando quisermos nos referir as execuções de organização e cena. O agente pong pode, agora, executar todos os comandos que o agente ping executou para entrar na organização, desempenhar papel e entrar na cena, usando, é claro, as variáveis que acaba de declarar.

```
32 ...
33 enterOrg $orgId
34 INFORM The agent succefully entered into the organization ping-pong [211]
35 performRole $orgId pong
36 INFORM The agent performs now the role pong
37 enterScene $orgId $sceneId pong
38 INFORM The agent succefully entered into the scene game [215] under the role pong
```

Listagem 5.14: Entrando em Organização.

Após entrar na cena o agente pong pode enviar uma mensagem de resposta para o agente ping. Note que não há necessidade de criar uma nova mensagem pois o agente pong já possui um conjunto de variáveis iniciadas por "fromPing" que representam a mensagem recebida por ping. Para enviar uma mensagem para ping, o agente pong pode sobrescrever os campos da mensagem que recebeu.

```
38 ...
39 set fromPing.receiver1 ping
40 fromPing.receiver1 set to ping
41 set fromPing.receiverRole1 ping
42 fromPing.receiverRole1 set to ping
43 set fromPing.info ping-pong
44 fromPing.info set to ping-pong
45 set fromPing.senderRole pong
46 fromPing.senderRole set to pong
```

PUC-Rio - Certificação Digital Nº 0420988/CA

Listagem 5.15: Configurando Resposta para Ping

Primeiramente o agente pong configura o agente receptor e seu papel. O conteúdo da mensagem original mapeado em info é sobrescrito por ping-pong para atender às especificações da lei. Finalmente o papel de pong é configurado como remetente. Note que os campos referentes à execução de organização e cena não foram modificados, pois a mensagem deve ser retornada ao destinatário ping pela mesma cena e organização utilizadas para chegar até o agente pong.

Após a configuração da mensagem o agente pong pode enviá-la ao agente ping.

```
46 ...
47 send $fromPing
48 INFORM toPong
```

Listagem 5.16: Respondendo a Ping.

Vale ressaltar que a mensagem retornada pelo envio de fromPing é a mesma referente ao envio da mensagem toPong. Isso se deve por estarmos enviando a mensagem toPong sob o nome de fromPing.

Portanto, assim que ping recebe sua resposta, pode verificar se o conteúdo do protocolo foi respeitado executando um comando de assert.

```
48 ...
49 receive 5 fromPong
50 INFORM toPong
51 assert $fromPong.info ping-pong
52 SUCCESS comparing $fromPong.info (ping-pong) with ping-pong
```

Listagem 5.17: Verificando Conteúdo da Mensagem.

O comando assert verifica o conteúdo do campo info da mensagem fromPong recebida e o compara com o valor esperado: "ping-pong". Como era esperado a assertiva será bem sucedida e vai escrever uma mensagem de sucesso na tela. Além de enviar uma mensagem de sucesso para o canal de saída, este comando escreve, no canal de controle, que não está sendo utilizado neste exemplo, o valor 0 (zero), representando uma assertiva de sucesso.

PUC-Rio - Certificação Digital Nº 0420988/CA