6
Estudo de Caso

O capítulo anterior apresentou o framework para desenvolvimento de agentes stubs utilizando XMLaw conceitual, estrutural e operacionalmente. Algumas instâncias também foram apresentadas para que exemplos pudessem ser apresentados. Neste capítulo, algumas aplicações de teste que foram desenvolvidas a partir do framework serão apresentadas.

A primeira aplicação é uma abordagem para a execução de diversos agentes stubs a partir de um único script de testes. A partir de um único arquivo é possível definir diversos agentes e simular seu comportamento de forma síncrona.

A segunda aplicação está relacionada com a primeira. Com a mesma idéia de script de testes utiliza-se um filtro para substituir partes estratégicas do script, previamente instrumentado, e com isso gerar diversos scripts diferentes que são executados para desempenhar testes de carga no sistema aberto.

A terceira aplicação, assim como a segunda, também utiliza-se da primeira aplicação. A partir do mesmo script definido para primeira aplicação, cria-se uma extensão do framework Junit que se utiliza da primeira aplicação para executar teste de unidade nos agentes do exemplo do aeroporto, apresentado mais a frente.

6.1
Múltiplos Agentes Genéricos

O capítulo 4 preocupou-se com a definição de cenários de teste. O conceito é uma simulação de uma possível interação no sistema aberto. Com o framework apresentado podemos implementar diversas instâncias para desempenhar esse tipo de simulação. Esta aplicação é uma solução para a simulação de cenários. Ela pretende fornecer ao desenvolvedor uma forma de implementar um cenário em um único arquivo, sem se preocupar com o espalhamento de scripts para cada agente criado. Outra preocupação é garantir que os agentes sempre executarão de forma síncrona, podendo garantir que o cenário sempre será repetido de forma fiel.

6.1.1

Simulação de cenários

Conforme explicado na seção 4.2, podemos simular cenários de execução para testarmos um agente real que foi implementado. Uma primeira abordagem para resolver o problema usaria diversos agentes genéricos com canais de entrada do tipo FileInputStream, segundo a exemplificação na listagem 5.2.

O objetivo seria simular o cenário e poder repeti-lo sempre que necessário. Infelizmente, essa abordagem não promove sempre uma repetição fiel de um cenário uma vez que os agentes são executados de forma assíncrona. Dessa maneira, não podemos definir comportamentos esperados. Logo, não podemos compará-los com aqueles obtidos.

Outro ponto contra a abordagem é o uso de diversos arquivos. No caso, um para cada agente stub. O inconveniente é justamente a criação distribuída do cenário, tornando sua compreensão confusa.

Uma segunda abordagem, mais madura, usa os agentes genéricos como solução. No entanto, o problema de sincronismo e distribuição dos arquivos é levado em conta. A solução é simples: uma implementação que utiliza diversos agentes genéricos sincronizados por um interpretador de script. Este interpretador, *ScriptInterpreter*, recebe em sua entrada um único canal que possui todas as requisições necessárias para execução do cenário. A figura 6.1 ilustra a arquitetura da solução.

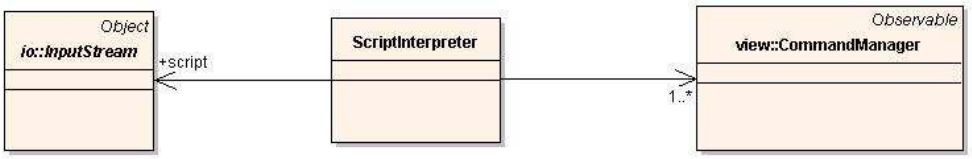

Figura 6.1: Estrutura de classes para a execução de diversos agentes

A implementação necessária envolve diversas instâncias de agentes genéricos, representados no diagrama por sua classe de controle *CommandManager*. O interpretador de script recebe as mesmas requisições que um agente genérico receberia. Porém, nesse caso, as requisições estão agrupadas seqüencialmente para que o interpretador saiba para qual agente direcionar a requisição.

A organização do script fica diferenciada, portanto, pela primeira linha, onde são declarados os agentes a serem utilizados, e pelas linha que indicam que os comandos posteriores serão destinados ao agente em questão. A sintaxe adotada é: na primeira linha os nomes de agentes a serem declarados, separados pelo caractere de espaço. As linhas terminadas em dois pontos indicam que as requisições posteriores serão destinadas ao agente mencionado antes de dois pontos. O caractere # representa comentário. Um exemplo por ser visto na listagem 6.1.

```
1  agent1 agent2 agent3
2
3  #comments
4
5  agent1:
6  #requests for agent1
7
8  agent2:
9  #requests for agent2
10
11 agent1:
12 #new requests for agent1
```

Listagem 6.1: Exemplo de um Script para Simular Cenários de Teste.

A comunicação com os agentes genéricos continua respeitando o esquema de canais. Nesse caso um pipe é utilizado da mesma forma da listagem 5.3. Conforme indicado no script, o interpretador direciona a requisição para o canal certo. Para o canal de saída e controle é utilizado um canal composto. Um canal composto é uma implementação que admite diversos canais internos. Ao escrever nesse canal, o conteúdo é redirecionado a todos aqueles que o compõem, seguindo a implementação do padrão *composite* [12]. Nesse caso, o canal composto é formado de dois canais para arquivos: um individual e outro compartilhado. Dessa forma, é possível observar o comportamento de cada agente ou o comportamento de todos eles ao mesmo tempo.

Ao escrever uma requisição no canal de entrada, a aplicação espera até que ela seja completamente executada. Assim, a questão do sincronismo é resolvida e podemos repetir sempre a execução de um mesmo cenário, usando um único script para isso.

6.1.2 Exemplo do Aeroporto

Em sua publicação original [7], a plataforma XMLaw foi ilustrada pelo exemplo do aeroporto de Toronto. No exemplo é considerado que aeroportos modernos possuem em seus salões de embarque diversos serviços que podem entreter o passageiro enquanto espera pelo seu vôo.

Um sistema governado por leis foi então proposto para atender uma pessoa que está num salão de embarque. Existem quatro papéis de agentes que podem interagir: announcer, customer, seller e bank. O agente announcer é responsável por divulgar os serviços disponíveis no aeroporto aos novos agentes que entram no sistema. Customer é o agente que deseja usufruir de algum dos serviços anunciados pelo agente announcer. O agente seller é quem disponibiliza o serviço e o agente bank é quem recebe os pagamentos pelos serviços prestados.

A interação dos agentes é feita por meio de quatro cenas. A primeira cena envolve o agente announcer e o agente customer, quando o último entra no salão do aeroporto. O protocolo de interação permite que o agente announcer informe para o agente customer os serviços disponíveis no salão aeroporto. A segunda cena envolve os mesmos agentes. Nela, o agente customer escolhe um dos serviços anunciados pelo agente announcer. Na terceira cena o agente customer em contato com o agente seller, escolhe um dos serviços oferecidos. Na última cena o agente customer entra em contato com o agente bank e paga pelo serviço solicitado ao agente seller.

Para simular uma interação que exercite as cenas descritas para lei do aeroporto, podemos supor um cenário de teste com seis agentes: announcer, customer, UCI, Cinemark, Itau e BB. O primeiro agente desempenha o papel de announcer e o segundo de customer. Neste cenário, o agente customer escolherá um serviço de cinema, portanto, dois agentes, UCI e Cinemark, representam a prestação do serviço de cinema. Outros dois agentes também desempenham o papel de bank, são eles Itau e BB.

A descrição do cenário é feita seguindo a especificação para definição de scripts apresentada anteriormente. Primeiramente, antes de iniciar qualquer cena, devemos declarar os agentes e associá-los aos respectivos papeis.

```
1  announcer customer UCI Cinemark Itau BB
2
3  announcer:
4  addLaw http://www.les.inf.puc-rio.br/xmlaw/airport.xml
5  set airportOrgId $last.orgExecutionId
6  enterOrg $airportOrgId
7  performRole $airportOrgId announcer
8
9  customer:
10 enterOrg $airportOrgId
11 performRole $airportOrgId customer
12
13 UCI:
14 enterOrg $airportOrgId
15 performRole $airportOrgId seller
16
17 Cinemark:
18 enterOrg $airportOrgId
19 performRole $airportOrgId seller
20
21 Itau:
22 enterOrg $airportOrgId
23 performRole $airportOrgId bank
24
25 BB:
26 enterOrg $airportOrgId
27 performRole $airportOrgId bank
```

Listagem 6.2: Criação de Agentes

A primeira linha, como descrito anteriormente, é referente à declaração dos agentes. Nela podemos ver o nome de todos os seis agentes que serão utilizados.

A partir da linha 3 as requisições são direcionadas ao agente announcer. Ele fica responsável pela publicação da lei, a configuração da variável $airportOrgId, que referencia o id da organização publicada, de entrar na organização e desempenhar o papel de announcer. Vemos que aos outros agentes são delegadas as requisições para entrada de cena e desempenho de papel. Note que todos os agentes podem utilizar a variável configurada pelo agente announcer.

Após a entrada na organização de todos os agentes, podemos começar a simular a primeira cena da organização onde o agente announcer oferece ao agente customer uma lista com todos os serviços disponíveis no salão de embarque.

```
27 ...
28 ############################
29 #####ANNOUNCEMENT SCENE#####
30 ############################
31
32 customer:
33 startScene $airportOrgId announcement
34 set announcementSceneId $last.sceneExecutionId
35 enterScene $airportOrgId $announcementSceneId customer
36 msg m1 request
37 set m1.hello hello
38 set m1.orgExecutionId $airportOrgId
39 set m1.sceneExecutionId $announcementSceneId
40 set m1.receiver1 announcer
41 set m1.receiverRole1 announcer
42 set m1.senderRole customer
43 send $m1
44
45 announcer:
46 receive 5 rm1
47 enterScene $airportOrgId $announcementSceneId announcer
48 reply $rm1 m2 inform
49 set m2.services movies;date;food
50 send $m2
51
52 customer:
53 receive 5 rm2
54 assert $rm2.services movies;date;food
```

Listagem 6.3: Cena de Anúncio

O agente customer é o primeiro a atuar criando a cena de anúncio. Ao criá-la, ele também configura uma variável para armazenar o valor de execução da cena criada. Evidentemente, após essas operações ele entra na cena e envia uma mensagem ao agente announcer, através da cena, contendo uma mensagem de "hello", conforme especificado no protocolo. O agente announcer, ao receber a mensagem, entra na cena e cria uma resposta para o agente customer, configurando um campo na mensagem que indica os serviços disponíveis, conforme especifica o protocolo da cena. Finalmente, a mensagem é enviada a customer e, ao recebê-la, o agente pode comparar o valor do campo services da mensagem com o valor esperado, executando para isso um comando de assert.

Essa troca de mensagens finaliza a primeira cena. Podemos, então, dar continuidade criando a cena de seleção de serviços que envolve os mesmos dois agentes.

```
54 ...
55 ############################
56 ######SELECTION SCENE#######
57 ############################
58
59 customer:
60 startScene $airportOrgId selection
61 set selectionScene $last.sceneExecutionId
62 enterScene $airportOrgId $selectionScene customer
63 msg m3 request
64 set m3.service movies
65 set m3.orgExecutionId $airportOrgId
66 set m3.sceneExecutionId $selectionScene
67 set m3.receiver1 announcer
68 set m3.receiverRole1 announcer
69 set m3.senderRole customer
70 send $m3
71
72 announcer:
73 receive 5 rm3
74 enterScene $airportOrgId $selectionScene announcer
75 reply $rm3 m4 inform
76 set m4.products IndianaJones;SpiderMan;StarWars
77 send $m4
78
79 customer:
80 receive 5 rm4
81 reply $rm4 m5 request
82 set m5.product SpiderMan
83 send $m5
84
85 announcer:
86 receive 5 rm5
87 reply $rm5 m6 inform
88 set m6.sellers UCI;Cinemark
89 send $m6
90
91 customer:
92 receive 5 rm6
93 assert $rm6.sellers UCI;Cinemark
```

Listagem 6.4: Cena de Seleção

Mais uma vez, o agente customer inicia a interação criando a cena para escolha de serviço. Como na interação anterior, ele configura um variável que possui o identificador de execução da cena em questão. Seguindo o protocolo descrito na lei, o agente envia uma mensagem ao agente announcer indicando que escolheu o serviço “movies”. O agente announcer, por sua vez, reponde com os filmes que estão disponíveis no momento. O agente customer escolhe o filme SpiderMan e o agente announcer indica dois agentes que disponibilizam o filme, UCI e Cinemark. Por fim, o agente customer verificar se o valor da mensagem recebida possui o nome dos agentes esperados através de outro comando de assert.

Agora que o agente customer conhece os agentes que disponibilizam o serviço que deseja, ele pode, seguindo a o protocolo especificado na terceira cena, negociar o preço do serviço com um dos agentes.

```
93  ...
94  ############################
95  #####NEGOTIATION SCENE######
96  ############################
97
98  customer:
99  startScene $airportOrgId negotiation
100 set negotiationScene $last.sceneExecutionId
101 enterScene $airportOrgId $negotiationScene customer
102 msg m7 cfp
103 set m7.product SpiderMan
104 set m7.price 8
105 set m7.orgExecutionId $airportOrgId
106 set m7.sceneExecutionId $negotiationScene
107 set m7.receiver1 UCI
108 set m7.receiverRole1 seller
109 set m7.senderRole customer
110 send $m7
111
112 UCI:
113 receive 5 rm7
114 enterScene $airportOrgId $negotiationScene seller
115 reply $rm7 m8 propose
116 set m8.product SpiderMan
117 set m8.price 10
118 send $m8
119
120 customer:
121 receive 5 rm8
122 reply $rm8 m9 accept-proposal
123 set m9.info ok
124 send $m9
125
126 UCI:
127 receive 5 rm9
128 reply $rm9 m10 inform
129 set m10.bank Itau
130 send $m10
131
132 customer:
133 receive 5 rm10
134 assert $m10.bank Itau
```

Listagem 6.5: Cena de Negociação

Neste cenário, o agente customer decidiu interagir com o agente UCI. Ele, portanto, cria uma cena de negociação e configura seu identificador de execução em uma variável. Em seguida envia para o agente UCI uma mensagem informando que esta disposto a pagar oito reais para assistir o filme SpiderMan. O agente UCI rebate com uma proposta de dez reais. O agente customer aceita, então, o preço pedido e o agente UCI informa qual o agente a ser contatado para o pagamento do serviço, no caso, o Itaú.

```
...
############################
#######PAYMENT SCENE########
############################

customer:
startScene $airportOrgId payment
set paymentScene $last.sceneExecutionId
enterScene $airportOrgId $paymentScene customer
msg m14 request
set m14.amount 10
set m14.to UCI
set m14.orgExecutionId $airportOrgId
set m14.sceneExecutionId $paymentScene
set m14.receiver1 Itau
set m14.receiverRole1 bank
set m14.senderRole customer
send $m14

Itau:
receive 5 rm14
enterScene $airportOrgId $paymentScene bank
reply $rm14 m15 inform
set m15.receipt receiptkey=12345
send $m15

customer:
receive 5 rm15
assert $rm15.receipt receiptkey=12345
```

Listagem 6.6: Cena de Pagamento

Finalizando, o agente customer inicia a última cena para fazer o pagamento do serviço escolhido. Como feito até então, ele inicia a cena de pagamento e armazena seu identificador de execução em uma variável. Para fazer o pagamento, o agente envia uma mensagem indicando o valor que deseja pagar ao agente UCI. O agente Itaú recebe o pagamento e envia um recibo ao agente customer para comprovar seu pagamento. A partir disso, o agente customer, ou melhor, seu usuário, pode assistir um filme enquanto espera pelo seu vôo.

6.2 Utilizando Agentes Stubs para Teste de Carga

Esta instância surgiu como um ferramenta de ajudar à análise de criticalidade de agentes de software para o estudo de caso SELIC. Este estudo é uma tentativa de aplicar a linguagem XMLaw, assim como o middleware M-Law, para a solução de problemas para o Banco Central do Brasil.

“O SELIC - Sistema Especial de Liquidação e Custódia - é o depositário central dos títulos da dívida pública federal interna. O Sistema também recebe os registros das negociações no mercado secundário e promove a respectiva liquidação, contando com módulos por meio dos quais são efetuados os leilões de títulos pelo

Tesouro Nacional ou pelo Banco Central. Quanto às negociações, o sistema acata comandos de compras e vendas à vista ou a termo, definitivas ou compromissadas, adotando os procedimentos necessários às movimentações financeiras e de custódia envolvidas na liquidação dessas operações, realizadas uma a uma e em tempo real. Por intermédio do SELIC também é efetuada a liquidação das operações de mercado aberto e de redesconto com títulos públicos decorrentes da condução da política monetária" [1].

Basicamente, a negociação de títulos no sistema SELIC é mediado pelo Banco Central. Existem, portanto, diversas regras que devem ser respeitadas durante uma negociação, assim como diversas entidades que estão dispostas a negociar esses títulos.

Nesse estudo de caso, a parte das regras que lidam com operações compromissadas foram modeladas seguindo a linguagem XLMaw. As negociações envolvem a participação de agentes que representam instituições financeiras que negociam com o agente Banco Central.

A modelagem resultou em duas cenas distintas chamadas de: Operação de compra/venda de títulos compromissada com preço unitário aberto e Operação de a recompra/revenda de títulos referente a compra/venda de títulos compromissada com preço unitário aberto, respectivamente identificadas por OpCompPUAberto e OpRecompPUAberto.

As cenas envolvem dois agentes que representam instituições financeiras, onde um assumirá o papel de comprador e o outro de vendedor, e o agente banco central que assume o papel selic. A complexidade do protocolo de interação não é importante para esta análise.

O estudo de caso tem como intuito validar um novo componente de software incluído na arquitetura do M-Law para analisar a criticalidade de agentes. Segundo esse trabalho [21], um agente pode ser considerado crítico de acordo com a quantidade de mensagens que ele troca com outros agentes. A idéia é, portanto, tornar o agente selic crítico. Em outras palavras, iniciar diversas execuções da cena regida pelo protocolo acima para verificar se o módulo de criticalidade está funcionando corretamente. É importante também que todas essas cenas sejam criadas de maneira assíncrona para permitir diversas trocas de mensagens em um dado instante. Caso contrário diversas mensagens seriam trocadas de forma serial, não acarretando em mudanças de criticalidade para o agente selic.

Como vimos nos estudos de caso anteriores não é difícil criar um script que simule a execução da cena em questão. A questão aqui é como gerar um script que represente diversos outros scripts para que as cenas sejam executadas forma assíncrona.

[1] http://www.andima.com.br/selic/oquee.asp

Podemos utilizar, portanto, a idéia de diversos agentes stubs em um só script e, adicionalmente, com o uso de um filtro substituir algumas partes desse script para gerar agentes diferentes.

A definição de filtro é uma classe estendida de *InputStream* que por sua vez possui um atributo do tipo *InputStream*. Tudo que for escrito no filtro é repassado ao atributo, mas antes um tratamento é feito no conteúdo escrito. Esse tratamento é o que caracteriza o comportamento do filtro, podendo adicionar ou remover caracteres do conteúdo que foi escrito. Outro uso interessante é que um filtro pode ter como atributo outro filtro, permitindo que seja construída uma cadeia de tratamento para o conteúdo escrito em determinado canal.

O mais importante no uso de filtros é que as aplicações que recebem canais de entrada não precisam de alterações. Com isso, a aplicação aqui reutilizada não precisa de modificação alguma, podendo ser reutilizada como uma caixa preta.

Assim sendo, a solução do problema encontra-se na definição de filtro que substitua um fragmento de texto indicado, chamado de marcação, por outro. Se colocarmos a marcação justamente na formação dos nomes dos agentes, com a criação de uma simples iteração, podemos variar dinamicamente a criação dos scripts.

A classe que implementa esse tipo de filtro foi batizada de *ReplaceTagFilter*. Em seu construtor recebe a marcação e o texto que deve ser substituído. Dessa forma, um script desenvolvido para exercitar as leis do sistema selic pode ser escrito com marcações pré-definidas e, após a aplicação de um filtro iterativamente, podemos formar diversos scripts diferentes.

Podemos ver na listagem 6.7 um script de testes que utiliza como marcação a string {execIndex}. Em 6.8 e 6.9 vemos a geração dos scripts quando um filtro é aplicado com a marcação em questão como argumento e os parâmetros 1 e 2 como texto de substituição.

```
1  IFA[{execIndex}] IFB[{execIndex}]
2
3  IFA[{execIndex}]:
4  receive 0 startMsgIF1
5  set orgId $startMsgIF1.orgExecutionId
6  set OpCompPUAbertoScene $startMsgIF1.sceneExecutionId
7  enterOrg $orgId
8  performRole $orgId comprador
9
10 IFB[{execIndex}]:
11 receive 0 startMsgIF2
12 enterOrg $orgId
13 performRole $orgId vendedor
14 ...
```

Listagem 6.7: Script com Marcações

```
1  IFA[1] IFB[1]
```

```
2
3  IFA[1]:
4  receive 0 startMsgIF1
5  set orgId $startMsgIF1.orgExecutionId
6  set OpCompPUAbertoScene $startMsgIF1.sceneExecutionId
7  enterOrg $orgId
8  performRole $orgId comprador
9
10 IFB[1]:
11 receive 0 startMsgIF2
12 enterOrg $orgId
13 performRole $orgId vendedor
14 ...
```

Listagem 6.8: Uso de Filtro com Parâmetro 1

```
1  IFA[2] IFB[2]
2
3  IFA[2]:
4  receive 0 startMsgIF1
5  set orgId $startMsgIF1.orgExecutionId
6  set OpCompPUAbertoScene $startMsgIF1.sceneExecutionId
7  enterOrg $orgId
8  performRole $orgId comprador
9
10 IFB[2]:
11 receive 0 startMsgIF2
12 enterOrg $orgId
13 performRole $orgId vendedor
14 ...
```

Listagem 6.9: Uso de Filtro com Parâmetro 2

Concluímos que, uma classe, para execução do teste de estresse, pode ser criada. Seu papel será instanciar interpretadores e aplicar filtros em seus canais de entrada. Os interpretadores devem ser alocados dentro de *Threads* para que a execução seja assíncrona. Conforme podemos ver na figura 6.2.

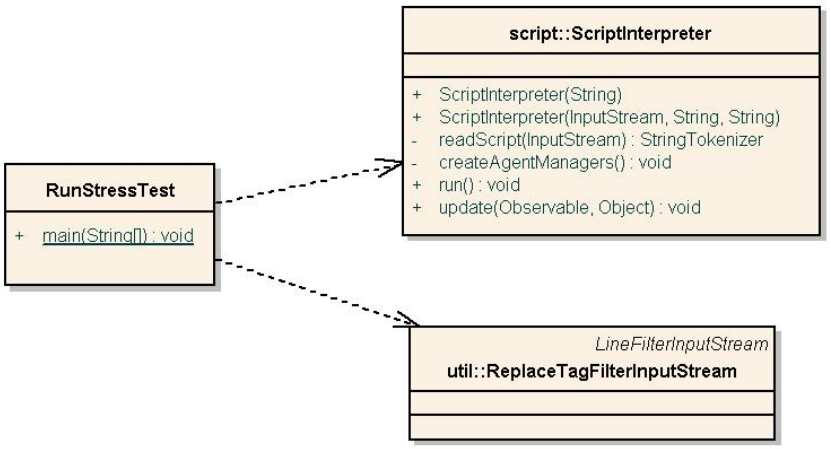

Figura 6.2: Estrutura de Classes para a Execução do Teste de Estresse

A implementação da classe responsável pela execução é bastante enxuta e pode ser visualizada na listagem abaixo. Observe que o número de scripts é fornecido como um parâmetro de entrada. O arquivo de script, lido por um *FileInputStream*, é introduzido dentro de um *ReplaceTagFilterInputStream*, e este finalmente é fornecido ao interpretador de script. A segunda iteração inicia as *Threads* para que se obtenha uma execução assíncrona.

```
1  ...
2  Thread[] interpreters = new Thread[numberOfExecutions];
3  for (int i = 0; i < numberOfExecutions; i++)
4  {
5   try
6   {
7    InputStream scriptInput = new ReplaceTagFilterInputStream(new FileInputStream(
         args[0]), "execIndex","" + (i+1));
8    interpreters[i] = new Thread(new ScriptInterpreter(scriptInput,"agents["+(i+1)+"
         ].txt","agents_ctrl["+(i+1)+"].txt"));
9    }
10   catch (IOException ioe)
11   {
12     System.err.println("Error␣parsing␣script␣file␣" + args[0]);
13     System.exit(0);
14   }
15 }
16 // all interpreters created
17
18 // start all interpreters
19 for (Thread interpreterThread : interpreters)
20 {
21  interpreterThread.start();
22 }
```

Listagem 6.10: Execução do Teste de Estresse

6.3 Teste Unitário de Agentes

Vimos nas seções anteriores duas aplicações produzidas a partir do framework de agentes genéricos. A aplicação apresentada nessa seção é um estudo de caso para execução de testes unitários de agentes para o exemplo do aeroporto. A abordagem fez proveito da primeira aplicação, que usa diversos agentes, e do framework para teste unitário JUnit [33].

6.3.1 Implementação dos Agentes para o Exemplo do Aeroporto

Para o desenvolvimento dos agentes na aplicação do aeroporto foi utilizado um modelo de classes com o padrão *State* [12]. Cada estado representa um comportamento que o agente deve ter conforme a mudança de estados do protocolo na lei. Podemos ver o modelo de classes utilizado para implementação do agente *Announcer* na figura 6.3.

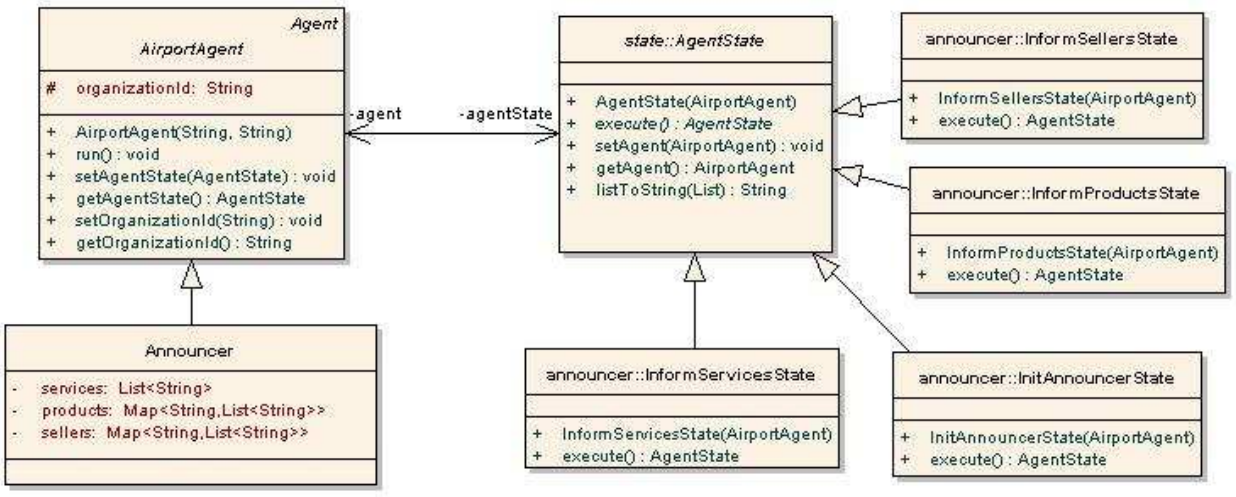

Figura 6.3: Estrutura de Classes para o Agente Announcer

O agente *Announcer* é responsável por divulgar os serviços, produtos e vendedores do aeroporto. Para isso, possui internamente algumas listas e tabelas que relacionam esses itens e possibilitam uma busca quando necessário. Vemos que todos os agentes do aeroporto são implementados com uma associação para uma classe *AgentState*, que representa o estado corrente do agente. Cada estado, ao finalizar sua execução, informa ao agente o próximo estado que deve ser configurado.

É importante então que a implementação dos estados desempenhem corretamente seu papel e informem de maneira inequívoca o próximo estado a ser configurado no agente. Para garantirmos isso podemos desempenhar alguns testes de unidade no agente em questão.

6.3.2 Implementação do Teste Unitário

Nas seções anteriores foi apresentado um script de teste para a aplicação do aeroporto. Tal script preocupa-se apenas em exercitar uma determinada lei publicada. No entanto, podemos aproveitar o mesmo script para realizar testes unitários nos agentes desenvolvidos para o sistema do aeroporto.

O objetivo é remover do script todas as requisições que sejam referentes ao agente que desejamos testar. Com isso, teremos um script incompleto onde podemos colocar o agente real no lugar das requisições removidas. No estudo de caso para o teste de carga foi utilizado um filtro que substituía marcações por algum outro fragmento de texto em questão. Podemos utilizar a mesma idéia e utilizar um filtro que remova as requisições de determinado agentes. Assim teremos o script que desejamos para o teste unitário.

De forma complementar, é preciso também de uma classe que inicie a execução do script e o agente em teste ao mesmo tempo. O modelo de classes para o teste unitário pode ser visto na figura 6.4.

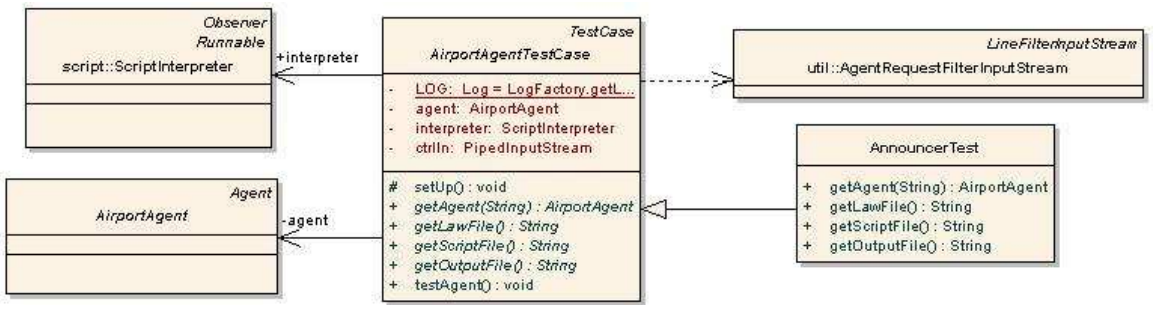

Figura 6.4: Estrutura de Classes para Teste unitário

A implementação da classe responsável pela execução do script e do agente foi implementada em *AirportAgentTestCase* e é uma extensão de framework JUnit [33] pelo fato de estender a classe *TestCase*.

Esse caso de teste possui referências para o agente a ser executado e a uma classe do tipo *ScriptInterpreter*, responsável pela execução do script. Quatro métodos de template [12] fornecem informação necessária para a configuração do caso de teste, como a lei a ser publicada, o script a ser utilizado, o agente a ser testado e o arquivo de saída para o interpretador.

O caso de teste instancia o interpretador fornecendo como entrada o script filtrado por dois objetos do filtro *AgentRequestFilterInputStream*, que implementa a filtragem das requisições dos agentes. Dois filtros são utilizados em seqüência para a remoção das requisições do agente manager, antes responsável por publicar a lei, e o agente em teste, fornecido pelo método de template. Como canal de controle do interpretador é fornecido um *PipeOutputStream* [30]. Assim, ao executar o teste do JUnit, a implementação do teste *testAgent()* monitora a saída do canal de controle em busca do caracter "1" que representa um erro de assertiva no script de teste. Caso isso aconteça, uma falha é disparada e o framework JUnit notifica o desenvolvedor. Dessa forma, ele pode procurar no arquivo de saída o motivo do erro da assertiva.

A listagem abaixo exemplifica a criação do caso de teste para o agente *Announcer*.

```
1  public class AnnouncerTest extends AirportAgentTestCase {
2      public AirportAgent getAgent(String orgId){
3          return new Announcer(orgId);
4      }
5
6      public String getLawFile(){
7          return "http://www.les.inf.puc-rio.br/xmlaw/airport.xml";
8      }
9
10     public String getScriptFile(){
```

```
11          return "script.txt";
12      }
13
14      public String getOutputFile(){
15          File dir = new File(ScriptInterpreter.REPORTS_DIR);
16          dir.mkdirs();
17          return ScriptInterpreter.REPORTS_DIR + "announcer_unit_out.txt";
18      }
19  }
```

Listagem 6.11: Teste Unitário para o Agente Announcer

Vemos, assim, que os métodos de template fornecem o agente *Announcer*, a lei do aeroporto, o script de teste mencionado anteriormente e um arquivo de saída que o interpretador registra as interações com o sistema aberto. Em caso de notificação de falha o desenvolvedor deve procurar o erro neste arquivo.

É interessante notar que o modelo flexibilizou o uso do script; um mesmo agente pode ser testado com diversos scripts diferentes, onde cada um testa uma funcionalidade distinta.

O uso do framework JUnit possibilita a execução integrada em diversos ambientes, como por exemplo o Eclipse [16], e também já fornece a possibilidade da execução de um conjunto de testes. Com isso o desenvolvedor não precisa executar um a um os casos de teste. Basta agrupá-los em um *TestSuit* [33]. O fato de cada aplicação possuir um tipo de agente diferente dificulta a generalização do teste unitário. Nesse caso o framework JUnit facilitou bastante a customização dos testes, possibilitando inclusive que o modelo empregado seja utilizado em outras aplicações.

Abaixo uma ilustração do caso de teste definido para o agente *Announcer* no ambiente Eclipse sendo executado com sucesso.

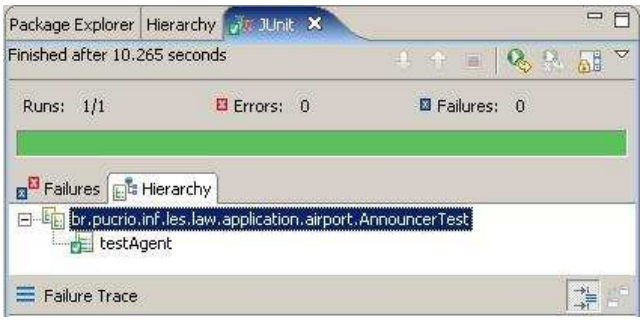

Figura 6.5: Execução do Teste Unitário para o Agente Announcer no Ambiente Eclipse

Durante a implementação do agente *Announcer* cinco erros foram encontrados durante a execução do teste unitário, são eles:

– Ausência de configuração do id de organização em *AirportAgent*;

– Ausência de configuração do agente executor do estado para o estado;
– Ausência do envio da mensagem m6 no estado *InformSellersState*;
– Erro ao reestruturar o código para representação de listas no formato requerido pelo protocolo;
– A listagem de serviços em *InformServicesState* estava sendo obtida pela chave errada.