Aula 7 - SOAR: Controlando o WorldServer3D

Atividade 1

Abra o código da classe SimulationSOAR.java, que contém o método main. Observe o uso da classe NativeUtils para resolver o problema do gerenciamento do código JNI para diferentes versões de sistema operacional. Entenda como esse gerenciamento é feito e registre em seu relatório.

Segundo a documentação do arquivo *NativeUtils.java*, o arquivo apontado é carregado no diretório temporário do sistema e então carregado. O arquivo temporário é deletado ao sair da aplicação.

No arquivo *SimulationSOAR.java* observamos que o código JNI (*Java Native Interface)* tem que ser carregado de acordo com a arquitetura de suporte. Por isso existe um passo de detecção de qual é o sistema operacional em funcionamento para carregar as *libs* corretamente.

```
String osName = System.getProperty("os.name").toLowerCase(Locale.
ENGLISH);
    String osArch = System.getProperty("os.arch").toLowerCase(Locale.
    ENGLISH);

    try {
  if(osName.contains("win")){
          if(osArch.contains("64")) {
            System.out.println("Windows 64 bits");
            NativeUtils.loadFileFromJar("/win64/Soar.lib");
            NativeUtils.loadFileFromJar("/win64/Soar.dll");
            NativeUtils.loadFileFromJar(
            "/win64/Java_sml_ClientInterface.dll");
          }
          else {
            System.out.println("Windows 32 bits");
            NativeUtils.loadFileFromJar("/win32/Soar.lib");
            NativeUtils.loadFileFromJar("/win32/Soar.dll");
            NativeUtils.loadFileFromJar(
            "/win32/Java_sml_ClientInterface.dll");
          }
      } else if(osName.contains("mac")){
    System.out.println("MacOSX");
```

Observe que essa solução é estendida ao carregamento de arquivos com código SOAR, para uso do controlador. Como isso é implementado no Demo ?

De forma semelhante é feito o carregamento do arquivo de produções chamado *soar-rules.soar* que depois é fornecido para a estrutura nomeada *SimulationTask* que é inicializada com a criatura e as regras. Importante dizer que é independente de plataforma, por isso não está dentro das condições dependentes de arquitetura.

```
}
NativeUtils.loadFileFromJar("/soar-rules.soar");
String soarRulesPath = "soar-rules.soar";
String soarDebuggerPath = "";
Integer soarDebuggerPort = 12121;
```

O loop principal de simulação do DemoSOAR também se encontra no método main. Explique seu funcionamento.

Logo abaixo do carregamento do arquivo de produções, como visto anteriormente, é instanciado um objeto *SimulationTask*, inicializado pelos métodos *initializeEnvironment* e *initializeCreatureAndSOAR*. Após uma espera (*Thread.Sleep*) então inicia-se o loop principal que é um laço infinito que chama um passo da simulação pelo método *runSimulation* do objeto *simulationTask* anteriormente instanciado com uma seguinte pequena espera (*Thread.Sleep*).

```
//Start enviroment data
SimulationTask simulationTask = new SimulationTask();
simulationTask.initializeEnviroment(Boolean.FALSE);
simulationTask.initializeCreatureAndSOAR(soarRulesPath,true,soarDebuggerPath,soarDebuggerPort);

// Run Simulation until some criteria was reached
Thread.sleep(3000);

while(true)
{
    simulationTask.runSimulation();
    Thread.sleep(100);
}
```

Acesse o código da classe SimulationTask.java, para compreender em mais detalhes o que está acontecendo. Observe que essa classe já se utiliza das classes de apoio em WS3DProxy. Entenda e explique como é feito o acesso ao WorldServer3D, por meio do WS3DProxy.

A entidade *SimulationTask* tem internamente um proxy WS3DProxy para o WorldServer3D. Investigando os pacotes do WS3DProxy, observa-se que ele é responsável por criar a conexão com o WorldServer3D via socket, disponibiliza métodos para lidar com o WorldServer3D, como inicializá-lo e criar criatura, além de encapsular a leitura de dados (método *prepareAndSetupCreatureToSimulation*) do WorldServer3D a partir dos sensores e também executar (método *processResponseCommands*) no mundo virtual os comandos (*eat, move, get*) recebidos.

Observe que a classe SimulationTask utiliza-se da classe SoarBridge, para ter acesso ao SOAR. Explique como é feita a leitura do estado do ambiente no WorldServer3D, e como esses dados sensoriais são enviado para o SOAR. Da mesma forma, explique como os dados enviados pelo SOAR são aproveitados para controlar a criatura no WorldServer3D. Registre suas conclusões no relatório de atividades.

Além da variável proxy, temos a bridge SoarBridge responsável por enviar e receber dados para o Soar. O método *prepareAndSetupCreatureToSimulation* é responsável pela leitura de dados e carregamento deles no Soar pelo método *soarBridge.setupStackHolder*. O método *processResponseCommands* é responsável por capturar os dados enviados pelo Soar que chegam pela chamada *soarBridge.getReceivedCommands*.

```
private void prepareAndSetupCreatureToSimulation(Creature creature, List<Thing> things) thr
{
    if (creature != null)
    {
        (...)
        // Setup StackHolder for run simulation
        soarBridge.setupStackHolder(StakeholderType.CREATURE,simulationCreature);
    }
    (...)
}
```

```
private void processResponseCommands() throws SoarBridgeException, CommandExecException
{
    // get simulation results
    ArrayList<SoarCommand> commandList = soarBridge.getReceivedCommands();

    if (commandList != null)
    {
        (...)
    }
}
```

Acesse o conteúdo do arquivo de regras SOAR: soar-rules.soar e tente entender seu funcionamento. Explique o princípio lógico de seu funcionamento.

O arquivo de regras contém diversos operadores, vamos analisá-los:

- *Wander*: operador que colocar a criatura para girar no próximo eixo. Seu uso é de menor prioridade e acontece no caso de a criatura não ter objetos no campo de visão nem na memória (abordaremos novamente abaixo);
- *See entity without memory count*: faz o agente manter algumas informações na memória, de jóias e de comida, neste caso inserindo pela primeira vez (com contador 1);
- *See entity with memory count*: nesta situação, que já existe o contador, atualiza o contador (remove o número anterior e adiciona uma nova entrada com o número+1) desde que ele não seja igual a 7;
- *Move food*: faz a criatura ir direto para a comida;
- *Eat food*: faz o agente comer a comida;
- *Move jewel*: faz a criatura ir direto para a jóia;
- *Get jewel:* faz a criatura pegar a jóia próxima (distância menor que 30);
- *Avoid brick*: faz o agente evitar o obstáculo (mantém distância de 61).

Em seguida, temos o conjunto de preferências e solução de impasses que resumidamente apresentam as características:

- See Entity (ver entidade):
    - Maior prioridade que mover até jóia e mover até comida
    - Menor prioridade que desviar de obstáculo
    - Igualdade entre ver entidade já com memória ou sem memória já salva
- Jewel (jóia):
    - Pegar jóia tem prioridade sobre ir até uma jóia ou até comida
    - Pegar jóia tem prioridade sobre desviar de obstáculo
    - Mover até jóia tem prioridade aquela com menor distância
    - Pegar jóia tem prioridade aquela com menor distância
- Food (comida):

    - Comer tem prioridade sobre mover até jóia ou até comida
    - Comer tem prioridade sobre evitar obstáculos
    - Mover até comida tem prioridade aquela com menor distância
    - Comer comida tem prioridade aquela com menor distância
- Disputa entre buscar comida ou buscar jóia
    - Se a criatura tiver o nível de combustível <= um limite, prioriza buscar comida
    - Se a criatura tiver o nível de combustível > um limite, prioriza buscar jóia
- Evitar obstáculos:
    - Evitar obstáculos tem prioridade aquele com menor distância
    - Desviar de obstáculos tem prioridade sobre buscar jóia ou comida
- Wander tem a menor prioridade geral, ou seja, só será proposto no caso da criatura não ter outra proposição válida

Atividade 2

O plano de implementação é dividido em quatro partes:

1. disponibilizar os dados no datamap de comunicação com o Soar
2. alterar as produções para considerar as novas informações de leaflets
3. disponibilizar o comando *deliver* para a criatura
4. alterar as produções para ativar o estado final

No passo 1, para disponibilizar os dados no datamap, primeiro alteramos o *SimulationRobot.java* para liberar acesso na lista de leaflets advindos de *Creature.java* pelo método *getLeaflets()*. Então abrimos o *SimulationTask.java* para preencher o espaço de memória necessário e por fim abrimos o *SoarBridge.java* para inserir as WME.

Na solução desenhada, criamos a variável *creatureLeaflets* que será preenchida somente uma vez. A partir dela, adicionamos na criature as WMEs necessárias. Nesta solução, optamos por projetar a criatura para completar todos os Leaflets antes de ir para o ponto de entrega coletar o pagamento. Isto trouxe uma simplificação: a de somar todas as entregas e todos os pagamentos e tratar tudo como uma grande leaflet única.

No passo 2, foi necessário [a] fazer alguns ajustes nos parâmetros dos operadores *moveJewel* para considerar a cor e o status do objetivo, [b] ajustar o *getJewel* para atualizar os contadores, [c] adicionar os operadores que inicializam variáveis de controle e [d] as preferências considerando os objetivos.

Em A, para ajustar o *moveJewel* adicionamos acesso as variáveis e passamos os valores ao propor o operador. Em B, na proposta do operador *getJewel* fazemos a ligação com os dados dos leaflets e no *apply* do operador atualizamos a contagem de quantos foram capturados.

Em C/D, precisamos inicializar a contagem de jewels capturadas, se o objetivo está ou não completo e a contagem de objetivos já completados: as funções de inicialização, propostas e de aplicação, são:

- *start*leaflet* :: inicializa o leaflet, inserindo a contagem de completados e um controle se foi ou não inicializado/terminado.
- *start*leaflet*jewel* :: inicializa um jewel da lista de jewels do leaflet, inserindo a contagem de capturados e se está completo ou não. Importante ressaltar a necessidade de uma variável de controle chamada ORDER para configurar os critérios

de desempate já que todos os operadores com match (para cada jewel a ser inicializado) mas usamos o ORDER para controlar qual executa primeiro.

- *completed*leaflet*jewel* :: detecta um jewel do leaflet que completou, marca que está completo e atualiza a contagem de completos no leaflet.

Nas preferências dos inicializadores, como esperado, tem qualquer inicializador mais prioritário que as ações da criatura (*preferences*all*leaflet*over*others*) e entre os inicializadores a prioridade é *start*leaflet > start*leaflet*jewel > completed*leaflet*jewel.*

Como os jewels do leaflet podem conflitar, no caso de um impasse de empate, para operadores iguais, eles seguem o campo order, no caso dos *start*leaflet*jewel* e *completed*leaflet*jewel.*

Adicionamos então duas regras para melhorar as decisões do *moveJewel* em que entre duas jewels temos as decisões:

- *moveJewel*moveJewel*leaflet*done*vs*undone ::* se <o> está incompleta e <o2> está completa, prefira <o>
- *moveJewel*moveJewel*leaflet*when*equal* :: se <o> tem o mesmo estado que <o2> ambas completas ou ambas não completas, não prefira, deixe que o controle de decisão por distância atue como usualmente

No passo 3, implementamos o comando *Deliver* para viabilizar o comando para a criatura. Adicionamos a classe *SoarCommandDeliver.java* e editamos o *SoarCommand.java* com uma entrada no enumerável que indica a operação de entrega.

Em *SimulationBridge.java* editamos o método *getReceivedCommands()* e alteramos *SimulationTask.java* para processar o comando em *getResponseCommands()*, a resposta no final somente é escrita no output padrão a mensagem de quantos pontos a criatura completou.

No passo 4, adicionamos nas regras do Soar a proposta e aplicação da operação *deliver*leaflet* que tem o argumento de quanto a criatura recebeu de pagamento. Esse caso acontece quando o número de objetivos existentes e completos são iguais.

Importante ressaltar algumas características:

- A decisão de continuar no mapa atuando, mesmo na ausência de um objetivo não completo, é que a chance de atrapalhar o oponente pode nos ajudar a ganhar o jogo. Esse *approach* talvez não fosse interessante se existisse um punição, por exemplo, quanto mais peso, mais energia gasta por segundo já que não foi descarregada.
- A tentativa de abrir duas criaturas resulta no *crash* das aplicações.
- Não foi possível competir com colegas de turma devido o término tardio do trabalho.