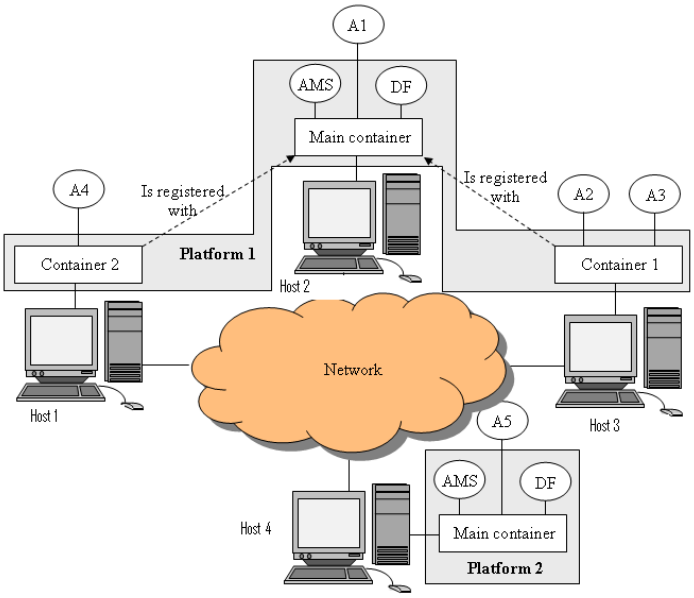
Integração de Sistemas

Relatório do Trabalho 3 – Sistemas Multiagente (JADE) e XML



Autores: João Barata Oliveira – 31559

Pedro Martins - 31501

Docentes Responsáveis:

Regente: José Barata de Oliveira

Responsável Práticas: André Rocha

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2015

Índice

Índice de Figuras

Índice de tabelas

Introdução

Antes de começar importa definir o que é um agente, e aqui neste ponto que parece tão elementar encontramos um obstáculo que ainda hoje divide grande parte da comunidade científica pois encontramos múltiplas definições para o conceito de agente. Na prática todas as definições de agentes tem alguns pontos em comum e um que se encontra em todas é o facto de um agente ter alguma autonomia.

Figura 1 - Exemplo de uma linha de produção baseada em sistemas multiagente

Os sistemas multiagente tem sido alvo de um grande foco de investigação por parte de instituições um pouco por todo o mundo. Neste momento existem sistemas multiagente em linhas de produção que permitem a essa mesma linha de se adaptar e evoluir à medida que produz cada vez mais produtos (1), um exemplo de uma linha de montagem deste tipo pode ser visto na Figura 1.

No entanto existem outras aplicações a este tipo de sistemas, por exemplo existem simuladores de redes elétricas inteligentes baseados em sistemas multiagentes (2) e também simuladores de ecossistemas em que cada elemento do ecossistema é modelado por um agente. É nestes últimos que vamos nos focar neste trabalho pois o trabalho desenvolvido é na prática um ecossistema constituído por Vacas e Lobos em que ambos tem de lutar pela sobrevivência

Objetivos do trabalho

Neste trabalho pretendemos perceber como funciona um sistema multiagente simples e perceber até que ponto o mesmo consegue modelar comportamentos verificados na natureza.

Neste caso em concreto pretende-se implementar um sistema multiagente que modele o comportamento de um grupo de vacas e lobos no interior de um ambiente fechado. Para isso foi decidido que cada elemento presente no mundo simulado seria representado por um agente, sendo que o próprio ambiente teria também um agente responsável por coordenar a simulação e iniciar os pedidos aos outros agentes de forma a atualizar a interface gráfica do utilizador.

Tabela 1 - Agentes e as suas responsabilidades

Agente	**Responsabilidade**
EnvironmentAgent	Este agente é responsável por coordenar a simulação e disparar os pedidos aos outros agentes para atualizarem os seus estados. Este agente também tem a capacidade para remover ou adicionar as restantes entidades e atualizar a interface gráfica.
CowAgent	Este agente é responsável por abstrair uma vaca presente no ambiente simulado. Este agente deve garantir que a entidade se desloca de forma a procurar comida (*Grass*) e fugir dos lobos (*Wolfs*) de forma a garantir a sobrevivência.
WolfAgent	Este agente é responsável por abstrair um lobo presente no ambiente simulado. Este agente deve garantir que a entidade se desloca de forma a procurar comida que neste caso são as vacas (*Cows*).

No que diz respeito aos Lobos e às Vacas é também importante de referir que tanto o elemento Lobo como o elemento Vaca tem de ser minimamente inteligente para que o sistema não seja totalmente determinístico e acaba invariavelmente da mesma forma, ou seja com as vacas a morrerem todas por ação dos lobos.

Outro objetivo deste trabalho foi perceber como funciona a serialização de um objeto usando a *framework* JAXB presente no JAVA e até que ponto a mesma pode ser utilizada para simplificar a comunicação entre agentes.

Implementação do sistema

Ambiente gráfico

Uma das alterações que se efetuou no código fornecido pelo corpo docente foi implementada logo na primeira janela que a Interface Gráfica de Utilizador (GUI), de forma a poder incluir nos parâmetros do ambiente o tempo que um lobo pode estar sem comer, ou como nós chamámos no programa TTL de Time To Live. Esta alteração pode ser vista na Figura 2.

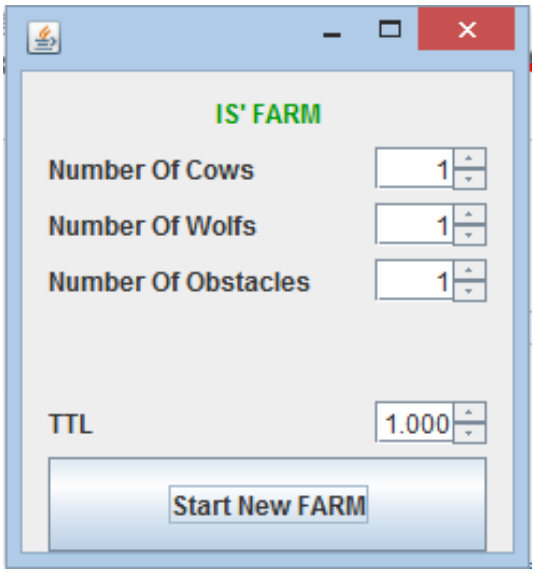


Figura 2 - Primeira janela da GUI do programa implementado

De seguida, após configurados todos os parâmetros será mostrado ao utlizador a interface gráfica da Figura 3.

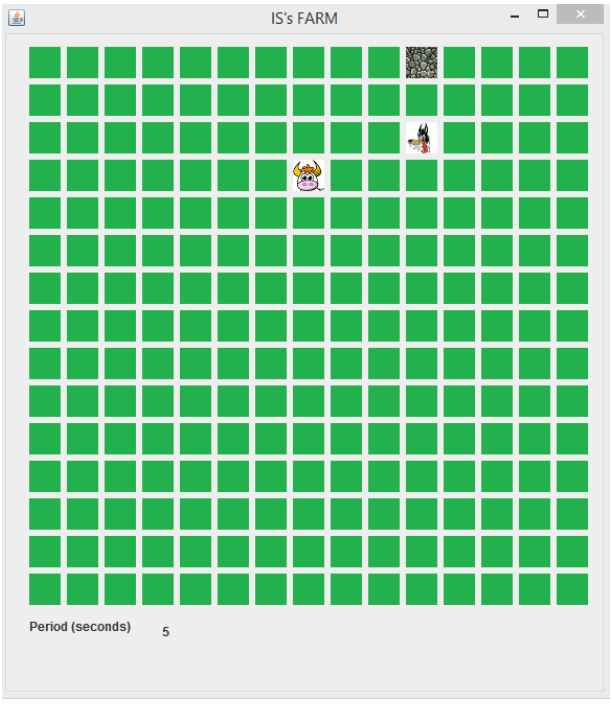


Figura 3 - Interface da simulação

Ontologias definidas

Como ponto de partida para o desenvolvimento do nosso trabalho tínhamos definidas três ontologias que deveríamos de utilizar na realização do nosso trabalho. Sendo elas as seguintes:

- ONTOLOGY_UPDATE_STATE:
  - Ontologia a ser utilizada quando o ambiente pede uma actualização do estado aos outros agentes.
- ONTOLOGY_KILL_COW:
  - Ontologia a ser utilizada quando o ambiente decide matar uma vaca por acção do Lobo ou por a mesma ter ficado sem alimento.
- ONTOLOGY_REMOVE_WOLF:
  - Ontologia a ser utilizada quando o ambiente decide matar o lobo por o mesmo não ter conseguido comer uma vaca durante um número de turnos definido pelo utilizador durante a configuração do ambiente.

Registo, Procura e Remoção do Directory Facilitator

Outro ponto essencial para o bom funcionamento do nosso sistema multiagente é a utilização eficaz dos recursos que o JADE põe ao nosso dispor, um dos recursos que mais utilizamos neste trabalho é sem dúvida o *Directory Facilitator* (DF) que permite a qualquer agente encontrar os serviços disponibilizados por outros agentes desde que eles se tenham registado previamente no DF.

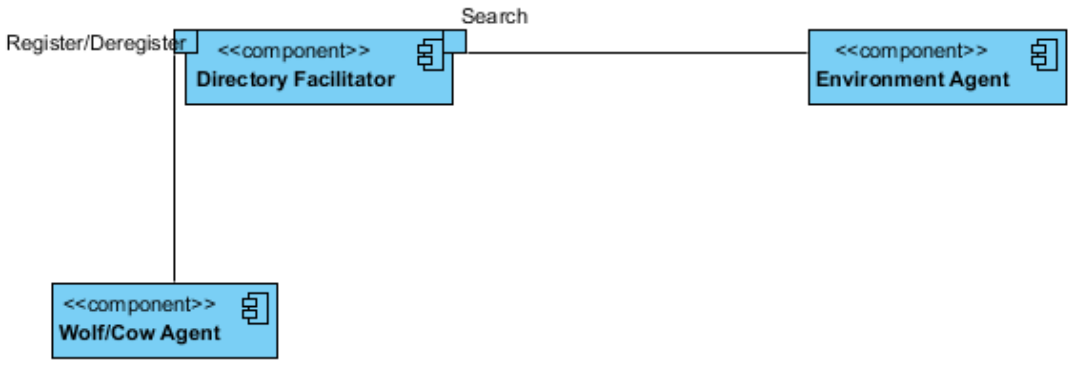


Figura 4 - Diagrama de componentes que utilizam o DF

Como podemos ver na Figura 4, Os lobos e as vacas vão se registar no DF quando são criados e desregistarem-se quando morrem. Durante a simulação o agente de ambiente vai poder pesquisar por eles no DF de forma a poder encontra-los e enviar-lhes mensagens.

Behaviours dos agentes

Agente *Cow* (Vaca)

No agente Cow foram implementados dois *Behaviours*:

- KillYourselfResponder
    - Responsável por matar o agente vaca e responder às mensagens enviadas a partir do *Behaviour KillYourselfInitiator* do agente de ambiente.
- UpdateStateResponder
    - Responsável por decidir qual será a próxima posição para onde a vaca se vai deslocar e que será enviada em resposta a um pedido do *Behaviour UpdateStateInitiator* também ele do agente de ambiente.

Para que estes *behaviours* sejam corretamente realizados pelo agente *Cow* é necessário adicionar uma instância dos mesmos quando o agente nasce, ou seja adicionar dentro da função *setup()*. De outra forma o agente seria incapaz de receber as mensagens provenientes do agente de ambiente. O diagrama de classes deste agente pode ser consultado na Figura 5.

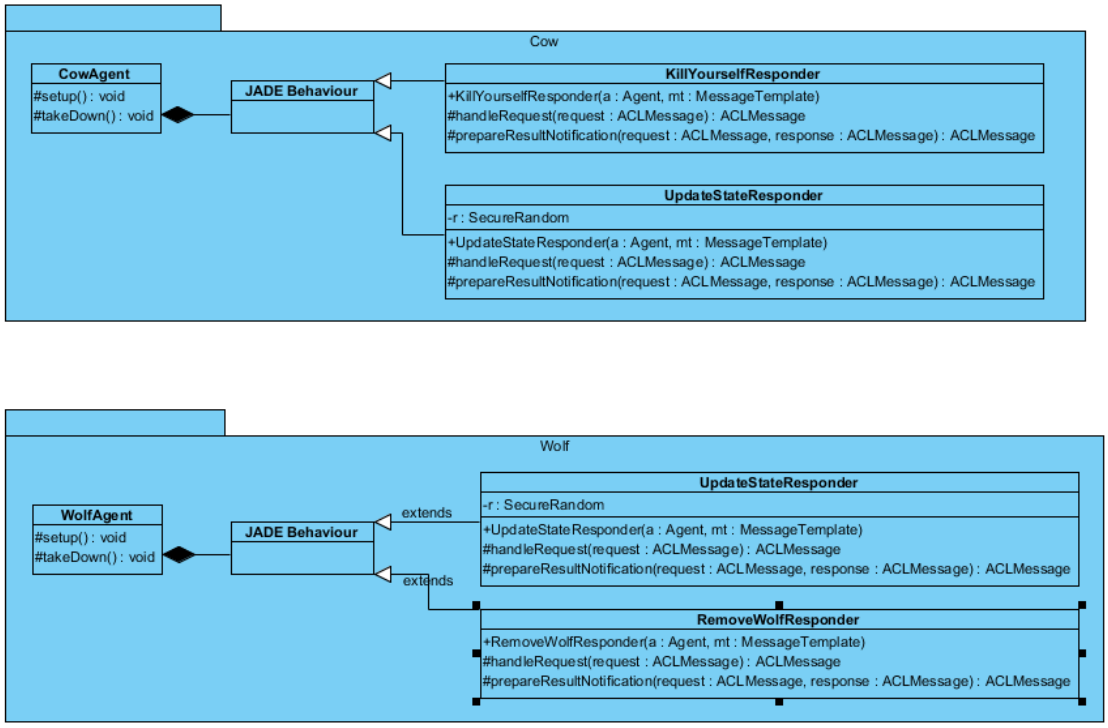


Figura 5 - Diagrama de classes dos Agentes Wolf e Cow

Agente *Wolf* (Lobo)

No agente Wolf foram implementados dois *Behaviours*:

- RemoveWolfResponder
    - Responsável por matar o agente lobo e responder às mensagens enviadas a partir do *Behaviour RemoveWolfInitiator* do agente de ambiente.
- UpdateStateResponder
    - Responsável por decidir qual será a próxima posição para onde o lobo se vai deslocar e que será enviada em resposta a um pedido do *Behaviour UpdateStateInitiator* também ele do agente de ambiente.

Tal como anteriormente para o agente vaca, para que estes *behaviours* sejam corretamente realizados pelo agente *Wolf* é necessário adicionar uma instância dos mesmos quando o agente nasce, ou seja adicionar dentro da função *setup()*.O diagrama de classes deste agente pode ser consultado na Figura 5.

Comunicação entre agentes

Inicialmente no código fornecido a comunicação entre agentes vinha feita de forma simplificada sem o envio do AGREE que é parte integrante do protocolo FIPA-ACL. Portanto como funcionalidade extra foi implementado no código de todos os *behaviours* métodos com a finalidade de reagir à receção de mensagens com esta performativa.

Portanto podemos ver na Figura 6 e Figura 7 os protocolos completos implementados no trabalho realizado.

Se quisermos ser extremamente rigorosos na implementação do protocolo, importa realçar que caso haja uma falha, em vez de um *inform*, os *Responders* irão responder com *failure*.

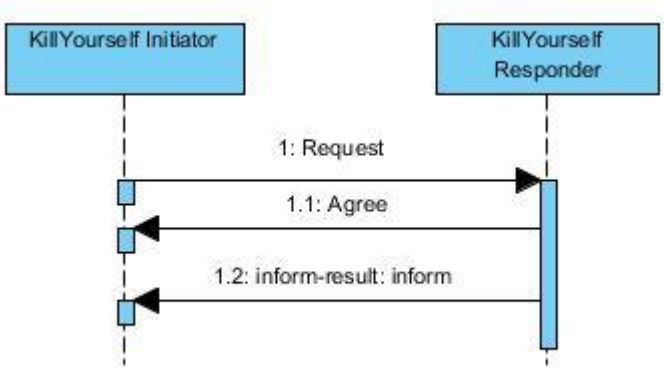


Figura 6 - Comunicação entre KillYourselfInitiator e KillYourselfResponder

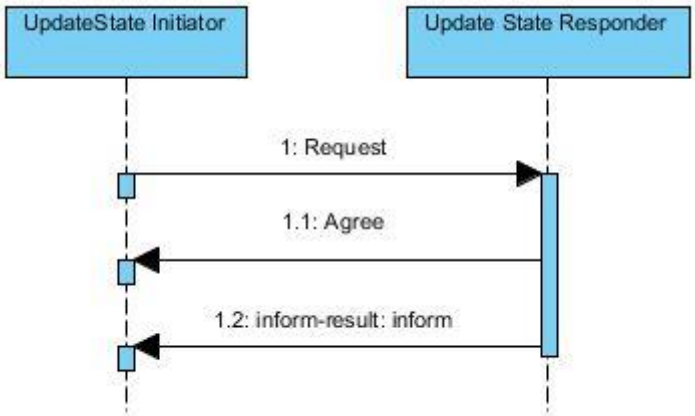


Figura 7 - Comunicação entre UpdateStateInitiator e UpdateStateResponder

Serialização dos objetos enviados

Uma das necessidades que tivemos foi a serialização dos objectos através da utilização da framework JAXB. Esta permite-nos facilmente criar um XML com o conteúdo serializado de uma classe, também ela gerada a partir de um XML com os atributos dessa mesma classe.

O funcionamento genérico do JAXB pode ser visto na Figura 8.

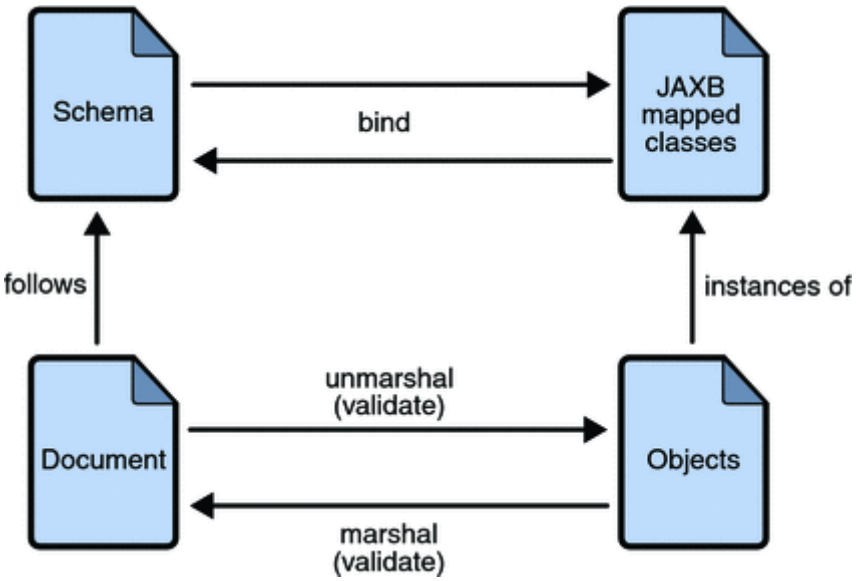


Figura 8 - Funcionamento dos bindings no JAXB (3)

No caso particular do nosso trabalho todo o processo de serialização e de-serialização está implementado na classe *MessageManagement*.

Inteligência das entidades

Vacas

No caso das vacas nós pretendemos que a Vaca fuja do lobo e portanto verificamos para cada posição do vector TMyPlace recebido se essa posição não tem lobo, e se as posições adjacentes também estão livres de lobo, caso estas condições sejam cumpridas a posição avaliada é adicionada a uma lista de posições possíveis.

Ao termos percorrido todas as posições recebidas, caso haja mais do que uma posição possível, o agente escolhe uma aleatoriamente, caso não haja nenhuma solução que responda aos critérios expostos, a Vaca fica no mesmo local.

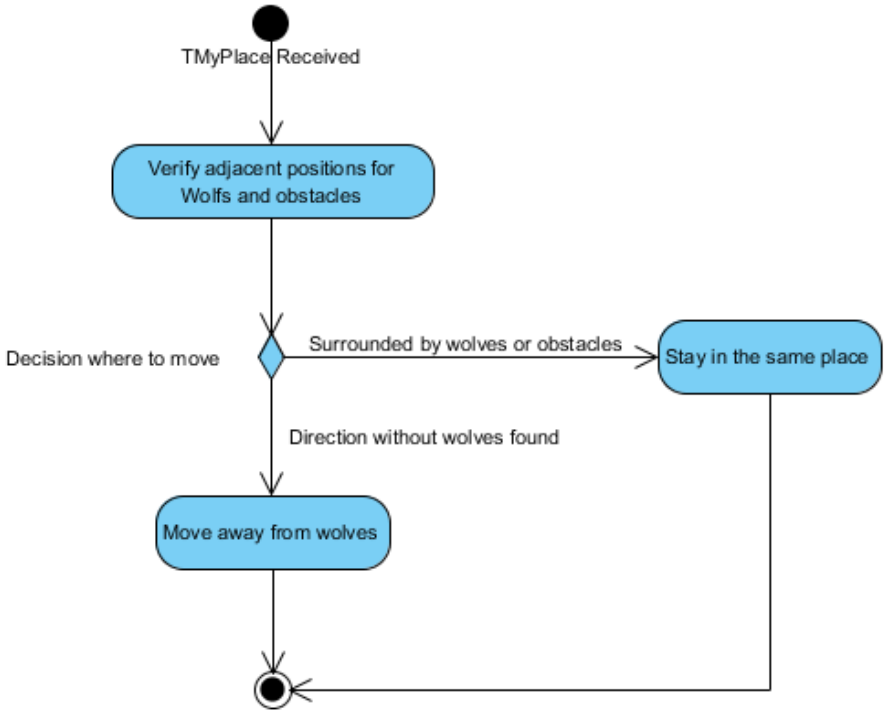


Figura 9 - Diagrama de atividade que representa a inteligência da vaca

Lobos

No caso dos Lobos, o algoritmo de inteligência implementado é significativamente mais simples pois o mesmo limita-se a procurar uma vaca nas posições adjacentes à sua posição atual. Caso não haja vacas, desloca-se para uma posição aleatória. Caso haja mais do que uma, vai atrás de uma delas aleatoriamente.

Morte do Lobo

Um dos requisitos do trabalho era implementar a morte do Lobo que implicou a criação de novas classes que implementavam esses *behaviours*. Na prática estes *behaviours* são muito semelhantes ao *KillYourself* presente nas vacas e portanto o código da morte das vacas é muito semelhante ao código da morte dos lobos. Também foi implementado no ficheiro myPlaceSchema.xml.xsd mais um elemento dentro do tipo tPlace para guardar o TTL do lobo. Quando o ambiente muda o lobo de posição, o ambiente faz com que o TTL do lobo acompanhe o mesmo e seja reiniciado caso o Lobo "encontre" uma vaca.

No entanto é importante voltar a referir que no nosso trabalho o tempo que um Lobo pode viver é uma escolha do utilizador. Caso o valor seja muito pequeno, as vacas ficam em vantagem e acabam por dominar a simulação, se ao contrário o TTL for muito grande os Lobos vão ficar em franca vantagem.

Conclusões

Através deste trabalho podemos concluir que um sistema multiagente é intrinsecamente um sistema ao mesmo tempo inteligente, não determinístico, distribuído e flexível e portanto não é de estranhar que cada vez seja mais utilizados em diferentes domínios da robótica tanto em ambiente fabril como ao ar livre em aplicações como *swarms* de UAVs.

No caso que estudamos, os agentes permitiram de uma forma muito simples simular um ecossistema à partida não determinístico a não ser que se altere os parâmetros de forma a reduzir a probabilidade de um determinado tipo de agentes sobreviver. Sem a utilização de agentes a simulação de um ambiente como o que foi implementado tornar-se-ia muito mais difícil. O facto também de cada agente ter os seus próprios comportamentos permite que cada elemento reaja de forma individual ao que o rodeia e permite que o sistema facilmente se adapte à entrada ou saída de agentes.

Referencias

1. *A multiagent based control approach for evolvable assembly systems.* **Barata, José, Camarinha-Matos, Luís e Onori, Mauro.** s.l. : IEEE, 2005. 2005 3rd IEEE International Conference on Industrial Informatics (INDIN). pp. 478-483.

2. *Interfacing Issues in Multiagent Simulation for Smart Grid Applications.* **Wang, X, et al.** 3, s.l. : IEEE, 2013, IEEE Transaction on Power Delivery, Vol. 28. 0885-8977.

3. **Oracle Corporation.** Part III Web Services (The Java EE 5 Tutorial). *Oracle Help Center.* [Online] [Citação: 14 de Junho de 2015.] https://docs.oracle.com/cd/E19879-01/819-3669/6n5sg7bi9/index.html.